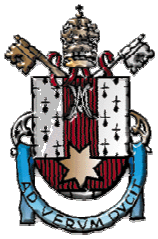
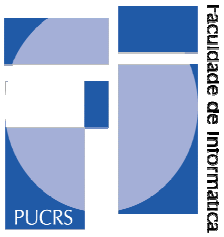


Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Curso de Bacharelado em Ciência da Computação



Arquitetura Multiprocessada em SoCs: Estudo de Diferentes Topologias de Conexão

Trabalho de Conclusão II

Autores:

Aline Vieira de Mello

Leandro Heleno Möller

Orientador:

Fernando Gehm Moraes

Porto Alegre, Junho de 2003.

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

A maior parte do desenvolvimento de métodos e ferramentas para o projeto de sistemas digitais das últimas quatro décadas (60 a 90) derivaram de necessidades relacionadas com o projeto de computadores e sistemas periféricos. Contudo, a ênfase de pesquisa tem se deslocado gradativamente para um mercado muito mais vasto, o de sistemas eletrônicos que entram na composição de produtos de uso específico. Exemplos são automóveis, aeronaves, eletrodomésticos e dispositivos de comunicação pessoal tais como telefones celulares e *pagers*. Sistemas embarcados ou sistemas embutidos são para todos os efeitos, sistemas computacionais que executam uma função específica. Eles possuem a mesma estrutura geral de um computador, mas a especificidade de suas tarefas faz com que não sejam nem usados nem percebidos como um computador [DEM95] [LAV95].

De acordo com vários autores [HAM97] [PAT97], em menos de 7 anos já existirão no mercado circuitos integrados compostos por mais de um bilhão de transistores. Com esta capacidade de integração, pode-se imaginar a inclusão de um sistema computacional completo em um mesmo chip, o que cria o conceito de SoC (System On a Chip). A vantagem na utilização de SoCs está na interface de comunicação, pois nos sistemas atuais o maior gargalo é a perda de desempenho causada pela troca de informações entre o hardware e o software executados em CIs (Circuitos Integrados) distintos. Caso os componentes de hardware e software estejam integrados em um único CI, o desempenho global do sistema tende a ser muito maior. Além do mais, a possibilidade de realizar um SoC pode reduzir o tempo de chegado do produto ao mercado (*time-to-market*) e criar novas relações de desempenho entre o hardware e o software.

É neste contexto que entram em cena os núcleos de hardware (*cores*), módulos de hardware pré-caracterizados, usados no desenvolvimento de sistemas computacionais integrados. Neste trabalho, os núcleos de hardware são denominados *núcleos*. Através do reuso, de componentes complexos, é possível atender às demandas do mercado atual e reduzir a distância entre a quantidade de recursos disponíveis e a produtividade das equipes de projeto.

O *primeiro objetivo* deste trabalho é estudar a possibilidade de implementar um sistema multiprocessado em dispositivos programáveis FPGA, com conexão por barramento. Estes processadores embarcados podem tanto operar em paralelo para resolver um problema complexo, quanto operarem independentemente. A motivação para estudar este tipo de sistema deve-se ao fato que tradicionalmente multiprocessamento resulta em ganho de desempenho. Entretanto, estes sistemas são tradicionalmente implementados com múltiplos circuitos. Implementando-se todo o sistema em único dispositivo pode-se obter um incremento de desempenho, dada a proximidade física dos componentes.

Para atingir este objetivo núcleos do tipo processador, memória e serial são desenvolvidos. O módulo serial é responsável pela troca de dados do computador hospedeiro com o SoC. Sua tarefa é alimentar a memória com um programa a ser executado pelo processador. Com a implementação do sistema multiprocessado é possível avaliar o desempenho quando múltiplos núcleos requisitam acesso ao barramento.

O *segundo objetivo* do trabalho é explorar outras formas de comunicação entre processadores (ou núcleos), que não seja a de barramento. Para atingir este objetivo é implementado uma rede intra-chip (NoCs - *Network on Chip*) para interconexão de núcleos. Esta estrutura de interconexão troca mensagens entre núcleos de hardware como uma rede de computadores o faz. Com a implementação desta arquitetura obtêm-se ganho de desempenho, já que mais de dois núcleos de hardware podem se comunicar simultaneamente.

Este Capítulo apresenta os conceitos básicos relacionados a multiprocessamento e interconexão de núcleos. A Seção 1.1 aborda o estado-da-arte em processadores embarcados. A Seção 1.2 apresenta sistemas com mais de um processador e modelos de comunicação destes com as memórias. A Seção 1.3 discute diferentes formas de interconexão de núcleos de hardware.

O presente Trabalho de Conclusão está organizado da seguinte forma:

- O Capítulo 2 apresenta propostas de arquiteturas de sistemas interconectados por barramento e por rede intra-chip empregadas neste trabalho.
- O Capítulo 3 mostra como é feita a arbitragem para acesso ao barramento.
- O Capítulo 4 descreve o protocolo de comunicação entre o sistema e o computador hospedeiro.
- O Capítulo 5 apresenta o sistema de memória e os modos de acesso a mesma.
- O Capítulo 6 mostra uma breve descrição do processador e a comunicação deste com a memória.
- O Capítulo 7 apresenta a validação e a prototipação dos sistemas monoprocessado e biprocessado interconectados por barramento.
- O Capítulo 8 aborda os conceitos e as decisões de projeto para a implementação de uma rede de interconexão intra-chip.
- O Capítulo 9 apresenta a implementação e validação da chave, responsável por transferir mensagens entre núcleos da rede.
- O Capítulo 10 apresenta a rede intra-chip obtida a partir da integração das chaves, a validação funcional da mesma e a validação por prototipação do sistema biprocessado com interconexão por NoC.
- O Capítulo 11 apresenta três estudos de caso utilizados para avaliar funcionalmente a NoC desenvolvida.
- O Capítulo 12 apresenta as conclusões desse trabalho e direções para trabalho futuros.

A Figura 1 ilustra a organização do texto, conforme os módulos de hardware desenvolvidos.

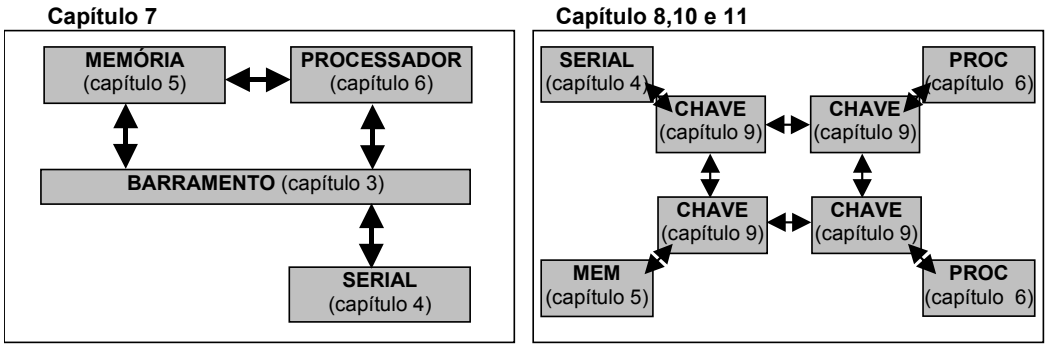


Figura 1 – Organização do texto.

1.1 Processadores Embarcados

Processadores embarcados estão presentes em diversos lugares. Uma estimativa realizada nos lares de classe média americanos indicou que existem, em média, 40 a 50 processadores em cada uma delas. Os processadores estão presentes em aparelhos de TV, videocassete, máquinas de lavar e secar, controles remotos, aparelhos de som, forno microondas, lavadora de louça, cafeteiras, geladeiras, videogames e muitos outros. Para se ter uma idéia, os novos carros tem em média 12 processadores chegando a 65 em uma Mercedes, e os computadores pessoais contêm aproximadamente 10 processadores [EXT03].

Processadores embarcados são componentes que permitem integrar em um mesmo dispositivo a parte de software com a parte de hardware do sistema, conduzindo a redução de custo e melhor desempenho. Processadores embarcados, como MIPS e ARM, não são mais utilizados apenas em ASICs. A evolução na tecnologia de semicondutores permite hoje integrar estes processadores em FPGA, e ao mesmo tempo disponibilizar milhares de portas lógicas para o usuário.

A iniciativa Excalibur, da Altera, disponibiliza dois processadores embarcados: NIOS e ARM . O NIOS é um *firm core*, que é integrado à lógica do usuário, descrita em VHDL ou Verilog. Já o processador ARM é disponibilizado na forma de um *hard core*, apresentando um desempenho superior ao NIOS.

O dispositivo Altera EPXA10 contém um processador ARM 922T, de arquitetura RISC 32 bits, que opera a 200MHz. Além do processador ARM há disponível para o usuário 256Kbytes de RAM porta simples, 128Kbytes de RAM dupla porta, 1 milhão de portas lógicas para implementar lógica configurável pelo usuário e mais de 1000 pinos de entrada e saída.

O Nios é um processador RISC de propósito geral, configurável, e que pode ser combinado com a lógica do usuário em um dispositivo lógico programável da Altera, de família selecionada. Algumas características do processador Nios são: (*i*) conjunto de instruções de 16 bits; (*ii*) largura de dados de 16 ou 32 bits; (*iii*) execução de uma instrução por ciclo de clock; (*iv*) suporte para memória *on-chip* e *off-chip*; (*v*) desempenho superior a 50 MIPS.

A iniciativa Empower, da Xilinx, também disponibiliza dois processadores embarcados: MICROBLAZE (*firm core*) e POWERPC (*hard core*) [XIL02]. O Microblaze é um processador RISC de 32 bits. Possui 32 registradores e segue a arquitetura Harvard. Suporta

acesso à memória do próprio FPGA onde reside (BlockRAMs) ou memória externa.

Para o desenvolvimento deste trabalho, utilizaremos um processador embarcado simples, desenvolvido localmente no grupo GAPH. O objetivo do trabalho é explorar e mostrar que multiprocessamento é factível em FPGAs. Assim sendo, não há razão para utilizar processadores complexos, o que desviaria a atenção do trabalho para o problema de estudar tais processadores, e não a rede de interconexão.

1.2 Multiprocessadores

Baseado nos conceitos descritos acima surge a idéia de um sistema multiprocessado, na qual cada núcleo de hardware pode ser um processador de propósito geral (General Purpose Processor - GPP). Teoricamente, a junção de *N* processadores pode conduzir a uma melhoria do desempenho em *N* vezes, atingido uma capacidade de processamento superior a qualquer sistema monoprocessado. São definidas duas classes gerais de multiprocessadores: memória compartilhada e troca de mensagem (ou passagem de mensagem) [HWA93] [KUM94] [PAT96], descritas a seguir:

1.2.1 Multiprocessadores de memória compartilhada

Nos multiprocessadores de memória compartilhada, todos os processadores compartilham um espaço de endereçamento global, o qual pode ser fisicamente centralizado ou distribuído. A sincronização e a comunicação entre os processos de uma mesma aplicação ocorrem através do acesso a variáveis compartilhadas em memória. Em função de como o compartilhamento de memória é realizado, podem ser definidas as subclasses descritas a seguir.

1.2.1.1 Modelo UMA

No modelo UMA *(Uniform Memory Acces*s), os múltiplos processadores são ligados por meio de uma rede de interconexão a uma memória global centralizada. Essa memória central é formada por módulos disjuntos, os quais podem ser acessados independentemente um do outro por diferentes processadores da máquina. Para reduzir o tráfego na rede, a cada processador pode ser associada uma memória local privativa para o armazenamento de dados locais e instruções de programa. Essas máquinas recebem esse nome porque o tempo de acesso à memória compartilhada é igual para todos os processadores.

1.2.1.2 Modelo NUMA

No modelo NUMA *(Non-Uniform Memory Acces*s), cada processador possui uma memória local, a qual é agregada ao espaço de endereçamento global da máquina. Dessa forma, podem existir até três padrões de acesso à memória compartilhada. O primeiro, e o mais rápido, é aquele onde a variável compartilhada está localizada na memória local do processador. O segundo padrão refere-se ao acesso a um endereço na memória central. Já o terceiro, e o mais lento, diz respeito ao acesso a uma posição localizada em uma memória local de outro processador. Dois modelos alternativos de máquina NUMA são mostrados na

Figura 2.

O modelo mostrado na Figura 2b é também chamado de multiprocessador de memória compartilhada distribuída (DSM - *Distributed Shared Memory*), pois toda a memória do sistema é distribuída entre os processadores da máquina, não havendo uma memória central.

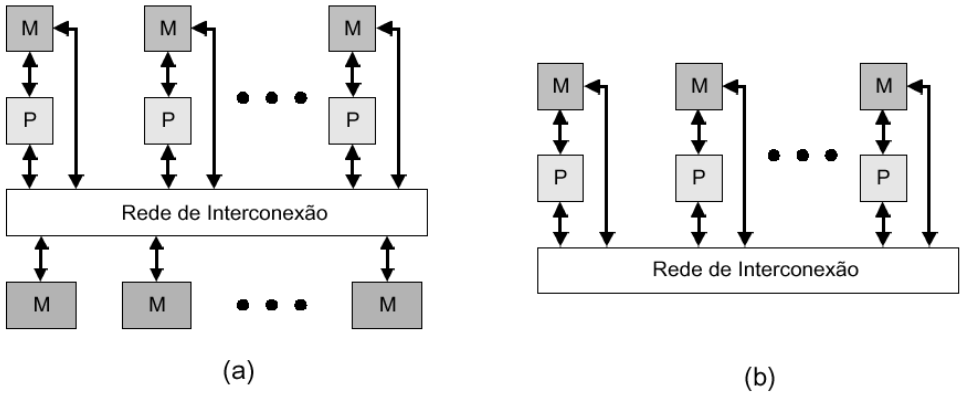


Figura 2 – Modelo NUMA: (a) com memória central; (b) sem memória central.

1.2.1.3 Modelos COMA e CC-NUMA

As máquinas COMA *(Cache-Only Memory Access)* são um caso particular das arquiteturas NUMA onde as memórias locais dos processadores são convertidas em caches. Todas as caches formam o espaço de endereçamento global e o acesso as caches remotas é auxiliado por meio de diretórios distribuídos. Já as máquinas CC-NUMA *(Cache-Coherent Non-Uniform Memory Access*) utilizam memória compartilhada distribuída e diretórios de cache, como, por exemplo, a máquina DASH de Stanford [LEN90]. Essas máquinas têm sido outra alternativa para a construção de máquinas escaláveis com memória compartilhada.

1.2.2 Multiprocessadores de troca de mensagem

Um multiprocessador de troca de mensagem é constituído por múltiplos processadores com memória local privativa e sem acesso à memória remota. Não existe compartilhamento de memória e a comunicação entre os processadores ocorre exclusivamente pela troca de mensagens através da rede de interconexão, normalmente através do uso de bibliotecas de comunicação como PVM ou MPI. Por esses motivos, essa arquitetura também recebe o nome de modelo NORMA *(NO-Remote Memory Access*), ou seja, sem acesso à memória remota. Um exemplo de arquitetura de troca de mensagem é ilustrado na Figura 3.

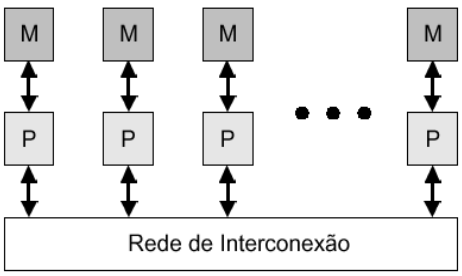


Figura 3 – Modelo NORMA.

1.3 Redes de Interconexão

Em uma arquitetura genérica de um SoC, os núcleos de hardware são integrados através de uma rede de interconexão comercial ou adaptada, com um controlador e funções de interface com o meio externo [PAL02] [MAD97]. Caso os núcleos de hardware sejam obtidos de diferentes fontes, a integração dos módulos e o teste do sistema podem ser difíceis, podendo até haver necessidade de que os núcleos de hardware sejam reprojetados, para adequá-los a um protocolo de interface comum. Para que o núcleo de hardware não seja reprojetado é construído um módulo (w*rapper* - [BRA02] [IYE02] [KOR02]) que encapsula o núcleo de hardware e o conecta à rede de interconexão.

A forma mais usual de interconectar núcleos é através de um barramento. Esta forma de conexão é discutida na Seção 1.3.1. Devido aos problemas de escalabilidade e desempenho, observa-se que esta forma de conexão está migrando para uma rede intra-chip, a qual denomina-se NoC (network-on-chip). NoCs são discutidas na Seção 1.3.2.

1.3.1 Barramento

A forma usual de interconexão entre os núcleos de hardware é a de barramento, classificado em três diferentes tipos [PAT96]:

- Processador-Memória: são normalmente curtos e extremamente velozes.
- Entrada/Saída: longos, geralmente conectam muitos dispositivos e atendem uma ampla faixa de banda passante.
- Backplane: permite que processadores, memórias e dispositivos de entrada/saída possam coexistir em um único barramento físico.

Existem diversas arquiteturas de barramentos propostos pela indústria, como o *CoreConnect* [IBM99] da *IBM, AMBA* [ARM02] da *ARM* e *WISHBONE* [SIL02] da *Silicore*. Estas arquiteturas de barramento são geralmente vinculadas à arquitetura de um processador, tal como o *PowerPC* ou o *ARM* [BER00].

A Figura 4 [BER00] ilustra um SoC baseado na arquitetura de barramento *CoreConnect*. A arquitetura *CoreConnect* da *IBM* disponibiliza três barramentos para interconectar núcleos de hardware e lógica personalizável:

- Barramento Local do Processador (*Processor Local Bus - PLB*): usado para interconectar núcleos de hardware com alto desempenho, grande largura de banda, tais como o PowerPC, controladores DMA e interfaces de memória externa.
- Barramento Periférico (*On-Chip Peripheral Bus - OPB*): usado para interconectar periféricos que trabalham com baixas taxas de dados, tais como portas seriais, portas paralelas, UARTs (Universal Asynchronous Receiver Transmitter), e outros núcleos de hardware com pequena largura de banda.
- Barramento de Registradores de Controle de Dispositivos (*Device Control Register Bus - DCR*): caminho de baixa velocidade, usado para passar configuração e informações de estado entre o processador e outros núcleos de hardware.

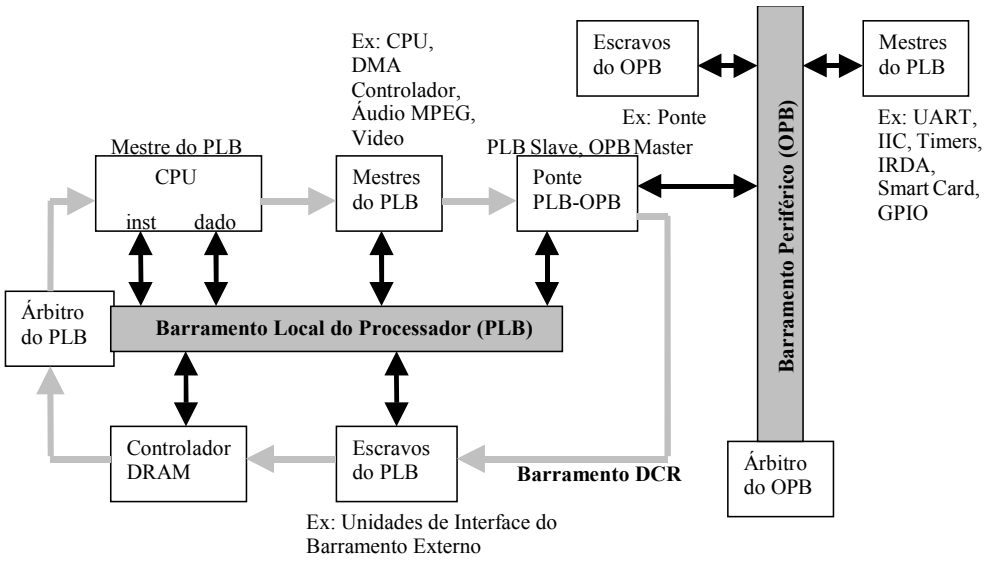


Figura 4 – SoC que usa a arquitetura de barramento *CoreConnect.*

Outra arquitetura que deve ser citada é a *WISHBONE*. Esta especificação pode ser utilizada para *soft*, *firm* e *hard cores*, já que *firm* e *hard cores* são geralmente concebidos a partir de *soft cores*. A especificação não requer o uso de ferramentas de desenvolvimento ou dispositivos-alvo (hardware) específicos [SIL02].

Os desenvolvedores do *WISHBONE* foram influenciados por três fatores principais. Primeiro, havia a necessidade de uma solução boa e confiável para a integração de núcleos de hardware em SoCs. Segundo, havia a necessidade de uma especificação de interface comum para facilitar as metodologias de projeto estruturadas para grandes equipes de projeto. Terceiro, eles foram influenciados pelas soluções de integração de sistemas tradicionais, fornecidos por barramentos de microcomputador como o PCI, por exemplo.

De fato, a arquitetura do *WISHBONE* é análoga a um barramento de microcomputador, sendo que: (*i*) oferece uma solução flexível para integração que pode ser facilmente adaptada à uma aplicação específica; (*ii*) oferece uma variedade de ciclos de acesso ao barramento e de larguras de caminhos de dados para atender a diferentes requisitos; e (*iii*) permite que os núcleos de hardware sejam projetados por vários fornecedores.

Os projetistas do *WISHBONE* criaram uma especificação robusta o suficiente para assegurar a compatibilidade entre IP *cores* e, ao mesmo tempo, sem restringir a criatividade do projetista e do usuário final. Em Janeiro de 2001, a organização *OpenCores* adotou o *WISHBONE* como um padrão de conectividade entre seus núcleos de hardware. Ele foi escolhido por ser o único a atender os requisitos adotados por esta instituição. É um barramento flexível, simples e aberto ao domínio público [OPE01].

Ao contrário de arquiteturas como a *CoreConnect*, o *WISHBONE* tem uma estrutura simples, composta de um único barramento, como mostra a Figura 5. Um sistema com muitos componentes pode incluir duas interfaces *WISHBONE*: uma para blocos que exigem alto desempenho e outro para periféricos de baixo desempenho.

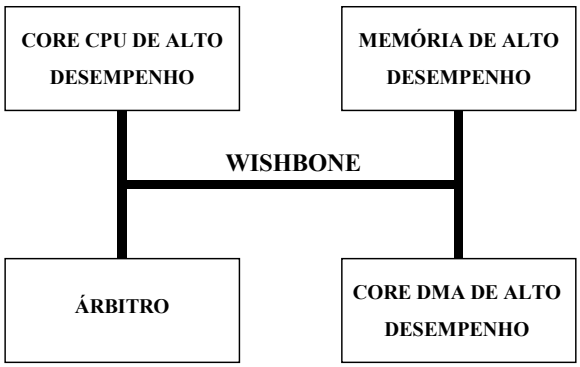


Figura 5 – Estrutura *WISHBONE.*

A interconexão por barramento é simples, sob o ponto de vista de implementação, apresentando entretanto diversas desvantagens [BEN02]: (*i*) apenas uma troca de dados pode ser realizada por vez, pois o meio físico é compartilhado por todos os núcleos de hardware, reduzindo o desempenho global do sistema; (*ii*) existe necessidade de desenvolver mecanismos inteligentes de arbitragem do meio físico para evitar desperdício de largura de banda; (*iii*) a escalabilidade é limitada, ou seja, o número de núcleos de hardware que podem ser conectados ao barramento é muito baixo, tipicamente na ordem das dezenas; (*iv*) o uso de linhas globais em um circuito integrado com tecnologia submicrônica impõe sérias restrições ao desempenho do sistema devido às altas capacitâncias e resistências parasitas inerentes aos fios longos. Estas desvantagens podem ser parcialmente contornadas através do uso de, por exemplo, hierarquia de barramentos, onde o problema continua existindo, sendo apenas minimizado.

1.3.2 Network on Chip - NoC

Uma maneira de solucionar os problemas oriundos da arquitetura de barramentos é através da utilização de redes internas ao circuito integrado [DAL01] [WIN01], mecanismo denominado de NoC – *Network on Chip*. Estas redes herdam conceitos utilizados em telecomunicações, que é a organização da transferência de informação em camadas e protocolos [SGR01].

Uma rede é composta por nodos de processamento e nodos de chaveamento. Por exemplo, a rede em anel (Figura 6) é uma topologia bastante simples e econômica. Cada nodo de chaveamento possui ligações para dois nodos de chaveamento vizinhos e para um nodo de processamento local.

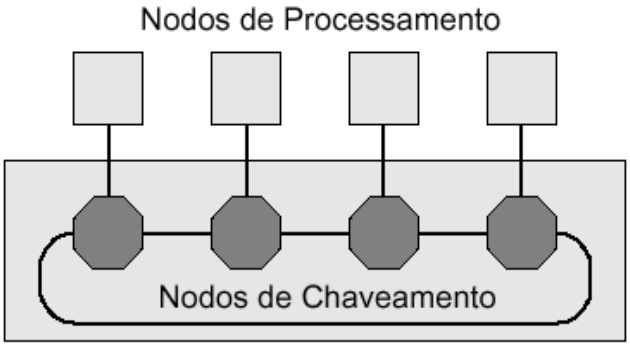


Figura 6 – Topologia em anel.

Os nodos de processamento (Figura 7a) possuem pelo menos um processador e uma interface para a rede de interconexão (chamada *interface de rede*), podendo ter, ainda, memória local, discos e outros periféricos.

Os nodos de chaveamento (Figura 7b) realizam a transferência de mensagens entre os nodos de processamento. Em geral, eles possuem um núcleo de chaveamento (ou chave), uma lógica para roteamento e arbitragem (abreviado por R&A) e portas de comunicação para outros nodos de chaveamento e, dependendo da topologia, para um nodo de processamento local. As portas de comunicação incluem canais de entrada e de saída, os quais podem possuir, ou não, *buffers* para o armazenamento temporário de informações. As portas possuem ainda um controlador de enlace para a implementação do protocolo físico de comunicação. Na Figura 7b, a porta de interface com o nodo de processamento não inclui controlador de enlace, pois, nesse caso, é sugerido que a ligação entre os nodos é implementada por meio de trilhas de circuito. Alternativamente, ela poderia ser realizada por meio de um enlace.

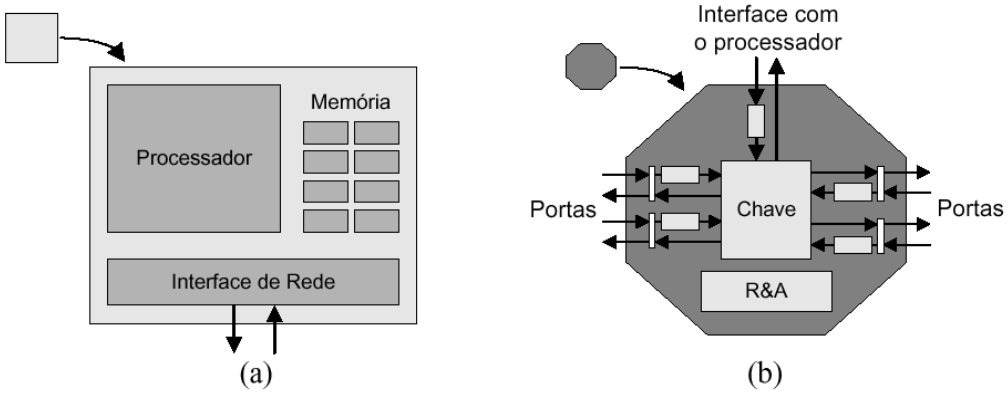


Figura 7 – Nodos: (a) de processamento; (b) de chaveamento.

A ligação física entre dois nodos de chaveamento é denominada enlace (*link*). Um enlace possui um ou dois canais físicos de comunicação e é implementado sob a forma de um cabo elétrico ou óptico. Dependendo da topologia da rede, um nodo de processamento também pode ser ligado a um nodo de chaveamento através de um enlace.

Em SoCs os enlaces são implementados através de fios físicos entre os módulos. Nas topologias NoC, as conexões são locais, entre módulos de chaveamento próximos, o que reduz o comprimento total dos fios físicos e por conseqüência pode potencialmente aumentar o desempenho elétrico. Já nas topologias de barramento, as conexões são globais, o que acarreta perda de desempenho devido aos fios longos.

As informações trocadas pelos nodos fonte e destino de uma comunicação são organizadas sob a forma de mensagens. Em geral, uma mensagem possui três partes: um cabeçalho (*header*), um corpo de dados (*payload*) e um terminador (*trailer*), sendo que o cabeçalho e o terminador formam um envelope ao redor do corpo de dados da mensagem. No cabeçalho são incluídas informações de roteamento e controle utilizadas pelos nodos de chaveamento para propagar a mensagem em direção ao nodo destino da comunicação. O terminador, por sua vez, inclui informações usadas para a detecção de erros e para a sinalização do fim da mensagem.

Tipicamente, as mensagens são quebradas em pacotes para transmissão. Um pacote é a menor unidade de informação que contém detalhes sobre o roteamento e seqüenciamento dos dados e mantém uma estrutura semelhante a de uma mensagem, com um cabeçalho, um corpo de dados e um terminador. Um pacote é constituído por uma seqüência de *flits*[1], cuja largura depende da largura física do canal.

Uma rede de interconexão pode ser caracterizada pela sua topologia e pelas estratégias utilizadas para roteamento, controle de fluxo, chaveamento e arbitragem que ela utiliza. Essas características são definidas brevemente abaixo:

- *Topologia*: é o arranjo dos nodos e canais sob a forma de um grafo.
- *Roteamento*: determina um caminho entre uma origem e um destino de uma mensagem.
- *Controle de fluxo*: lida com a alocação de canais e *buffers* para o tráfego de uma mensagem na rede.
- *Chaveamento*: define como e quando um canal de entrada é conectado a um canal de saída selecionado pelo algoritmo de roteamento.
- *Arbitragem*: determina qual canal de entrada pode utilizar um determinado canal de saída.

As redes de interconexão para multiprocessadores adotam muitos dos conceitos empregados em redes de computadores. Por exemplo, muitas redes de interconexão são estruturadas em camadas que encapsulam funções equivalentes àquelas definidas para os níveis hierárquicos do modelo de referência OSI (*Open System Interconnection*), um padrão internacional de organização de redes de computadores proposto pela ISO (*International Organization for Standardization*) [DAY83]. O objetivo de uma estrutura de protocolos em níveis é delimitar e isolar funções de comunicações a camadas. Cada nível deve ser pensado como um programa ou processo, quer implementado por hardware ou software, que se comunica com o processo correspondente na outra máquina. As regras que governam a conversação de um nível "K" qualquer são chamadas de protocolo de nível "K". O modelo OSI possui sete níveis de protocolos, como pode ser observado na Figura 8.

Figura 8 – Camadas do modelo OSI.

A arquitetura de uma rede que segue o modelo OSI é formada por níveis, interfaces e protocolos. Cada nível oferece um conjunto de serviços ao nível superior, usando funções

[1] Flit é a menor unidade de transferência de dados.

realizadas no próprio nível e serviços disponíveis nos níveis inferiores. Os nodos de chaveamento de redes de interconexão que são estruturados em camadas hierárquicas implementam algumas das funções dos níveis inferiores (físico, enlace, rede) do modelo OSI, descritas abaixo:

- Nível físico: realiza a transferência de dados em nível de bits através de um enlace.
- Nível de enlace: efetua a comunicação em nível de quadros (grupos de bits). Preocupa-se com o enquadramento dos dados e com a transferência desses quadros de forma confiável, realizando o tratamento de erros e o controle do fluxo de transferência de quadros.
- Nível de rede: faz a comunicação em nível de pacotes (grupos de quadros). Responsável pelo empacotamento das mensagens, roteamento dos pacotes entre a origem e o destino da mensagem, controle de congestionamento e contabilização de pacotes transferidos.

1.3.3 Topologias de Redes de Interconexão

Uma rede de interconexão pode ser caracterizada pela estrutura de interligação de seus nodos. Essa estrutura pode ser estruturalmente representada por um grafo G(N,C) onde *N* representa o conjunto de nodos (de processamento e/ou de chaveamento) da rede e *C* representa o conjunto de canais de comunicação. Quanto à topologia, as redes de interconexão para multiprocessadores podem ser agrupadas em duas classes principais, as redes diretas e as redes indiretas [ZEF03].

Nas redes *diretas*, cada nodo de chaveamento possui um nodo de processamento associado, e esse par pode ser visto como um elemento único dentro da máquina, tipicamente referenciado pela palavra nodo, como ilustra a Figura 9. Pelo fato de utilizarem nodos de chaveamento tipo roteador, as redes diretas são também chamadas de redes baseadas em roteadores [DUA97]. Uma outra denominação utilizada é a de redes estáticas, pois as ligações entre os nodos não mudam durante a execução do programa paralelo [HWA93].

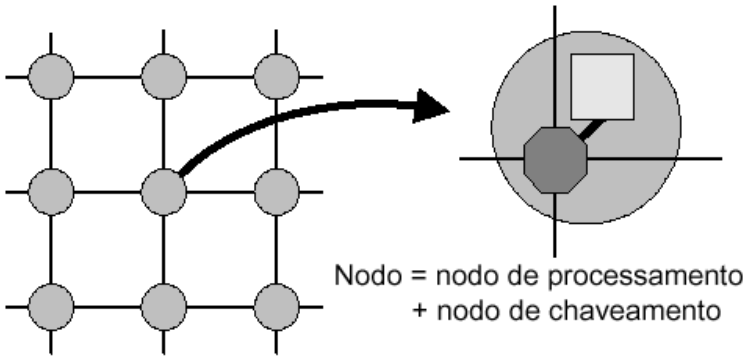


Figura 9 – Nodos de redes diretas.

As topologias de redes diretas estritamente ortogonais mais utilizadas são a grelha (ou malha) n-dimensional (Figura 10a), o toróide (Figura 10b) e o hipercubo (Figura 10c).

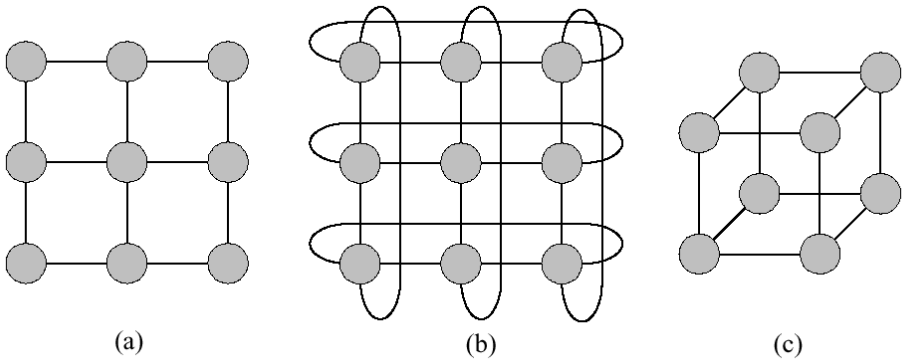


Figura 10 – (a) Grelha 2D 3x3; (b) Toróide 2D 3x3; (c) Hipercubo 3D.

Nas redes *indiretas*, o acoplamento entre os nodos de processamento e os nodos de chaveamento não ocorre no mesmo nível das redes diretas. Na visão unificada cada par nodo de processamento - nodo de chaveamento é visto como um elemento. Os nodos de processamento possuem uma interface para uma rede de nodos de chaveamento baseados em chaves. Cada chave possui um conjunto de portas bidirecionais para ligações com outras chaves e/ou com os nodos de processamento. Somente algumas chaves possuem conexões para nodos de processamento e apenas essas podem servir de fonte ou destino de uma mensagem. A topologia da rede é definida pela estrutura de interconexão dessas chaves.

Duas topologias clássicas de redes indiretas se destacam: o crossbar e as redes multiestágio. Para conexão indireta de N nodos de processamento, o crossbar (Figura 11) é a topologia ideal, pois consiste de uma única chave $N \times N$ capaz de ligar qualquer entrada com qualquer saída. Embora seja mais econômico que uma rede direta completamente conectada (a qual necessitaria de N roteadores, cada um com um crossbar $N \times N$ interno), o crossbar possui uma complexidade $O(N^2)$, o que torna o seu custo proibitivo para redes grandes.

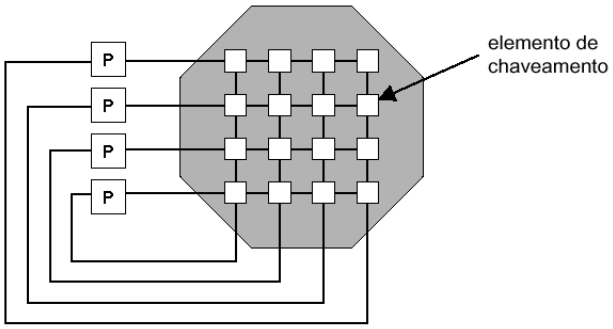


Figura 11 – Crossbar 4 x 4.

Além das redes vistas acima, existem inúmeras outras topologias de redes diretas e indiretas propostas com objetivos específicos, como por exemplo, minimizar o diâmetro da rede para um determinado número de nodos e grau do nodo [FEN81] [HWA93] [CUL98]. Entre essas topologias, podem ser citadas o cubo conectado por ciclos, a árvore, a árvore gorda, a estrela, a rede banyan, a banyan-hipercúbica, a pirâmide e a rede De Brujin, entre outras.

2 Arquitetura do Sistema

Este Capítulo apresenta de forma simplificada a interconexão de núcleos de hardware a partir das topologia de barramento e NoC. O barramento, como explicado na Seção 1.3.1, é uma forma simples de interconexão de núcleos de hardware sob o ponto de vista de implementação. Das arquiteturas de barramento apresentadas na Seção 1.3.1, a que mais se aproxima à arquitetura de barramento adotada neste trabalho é a Wishbone, que é composta por um único barramento que interliga todos os núcleos, provendo flexibilidade e simplicidade. Embora a conexão de núcleos de hardware por barramento possua as desvantagens já apresentadas, ela foi a escolhida pois é o primeiro passo para o desenvolvimento de uma forma de conexão mais avançada, além de servir como base de referência para desempenho. NoCs, como mencionado na Seção 1.3.2, são uma maneira de minimizar os problemas oriundos da arquitetura de barramentos através da utilização de redes internas ao circuito integrado. A topologia NoC adotada foi a Mesh (Malha), pois esta simplifica o algoritmo de chaveamento e é facilmente escalável.

Neste trabalho implementou-se quatro arquiteturas alvo de forma incremental. A Seção 2.1 apresenta o projeto do sistema inicial com barramento, contendo apenas uma memória para armazenamento de dados e um núcleo Serial responsável por fazer a comunicação com o computador hospedeiro. A conexão de um processador ao barramento, assim como sua comunicação com outros núcleos são detalhados na Seção 2.2. Na Seção 2.3 é apresentada a arquitetura de um sistema multiprocessado com múltiplas memórias, interconectado por um barramento. O sistema multiprocessado com interconexão por NoC é apresentado na Seção 2.4. A Seção 2.5 apresenta a plataforma de prototipação utilizada para teste em hardware das arquiteturas propostas.

2.1 Sistema Inicial

O primeiro passo para o desenvolvimento de uma arquitetura SoC baseada em barramento é definir a forma de controle de acesso ao barramento. O sistema inicial (Figura 12) tem por objetivo prover uma estrutura simples, utilizando para teste um núcleo Serial que transfere dados entre um computador hospedeiro e uma memória que armazena estes dados.

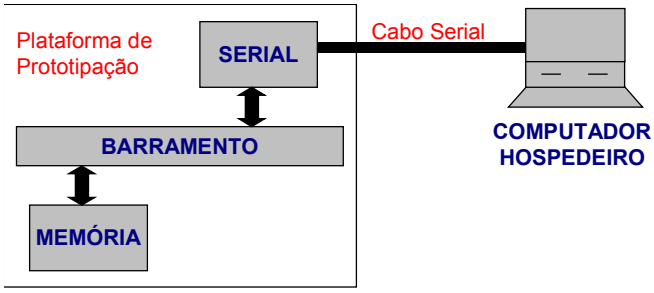


Figura 12 – Arquitetura do sistema inicial.

Neste modelo, um usuário pode enviar dados para o sistema utilizando um cabo serial que conecta de um lado o computador hospedeiro e do outro lado a plataforma de prototipação. No computador hospedeiro utiliza-se um software, implementado pelo grupo, para acessar a porta serial do computador enviando comandos e dados, de forma que o núcleo Serial possa

tratá-los. O núcleo Serial trata a mensagem recebida do mundo externo e verifica se é um comando válido. Caso seja, ele executa a ação correspondente ao comando. Um exemplo de comando recebido pelo núcleo Serial é a leitura do primeiro valor da memória. Para fazer uma leitura da memória o núcleo Serial requisita acesso ao barramento. Quando o acesso é liberado, o núcleo Serial monta um pacote com o destino (núcleo Memória), a origem (núcleo Serial), o comando a ser efetuado pela memória (leitura), o número de palavras a serem lidas da memória (01) e a posição inicial de acesso à memória (0000). Após o envio do pacote o núcleo Serial libera o barramento. O núcleo Memória ao receber o pacote, verifica a primeira palavra que identifica o núcleo destino. Como o núcleo Memória é o destino do pacote, ele verifica a segunda palavra que corresponde ao comando que deve ser executado. Neste caso, o comando indica a leitura de uma posição de memória. O núcleo Memória monta um novo pacote que tem como destino o núcleo Serial, como origem o núcleo Memória, como comando uma leitura e a seguir vem o dado lido do endereço requisitado. O núcleo Memória requisita o acesso ao barramento e o envia quando a requisição é atendida. O núcleo Serial reconhece que o pacote é destinado a ele e o trata, extraindo as informações necessárias e as enviando para o computador. O software que está executando no computador recebe os dados enviados e os mostra na tela.

A operação realizada acima é dita simples, pois acessa uma única posição de memória. O sistema também permite o acesso em modo rajada, no qual com um único pedido do computador hospedeiro, o núcleo Serial executa diversos acessos ao núcleo Memória. A única modificação em relação à operação de acesso simples é referente ao número de palavras informadas na operação.

O projeto deste sistema inicial permitiu ao grupo definir os protocolos básicos dos comandos de leitura e escrita, de acesso à memória e comunicação com o computador hospedeiro. A partir deste projeto é possível adicionar um processador ao sistema, como será visto na próxima Seção.

2.2 Sistema Monoprocessado com Interconexão por Barramento

Para transformar o sistema apresentado na Seção anterior em um sistema computacional programável é necessário inserir um processador ao barramento. Isto permite que um programa seja armazenado na memória e executado no processador. O núcleo Serial, a partir deste momento, será utilizado também para a carga de programas na memória e para troca de comandos entre a plataforma de prototipação e o computador hospedeiro. A Figura 13 ilustra este sistema.

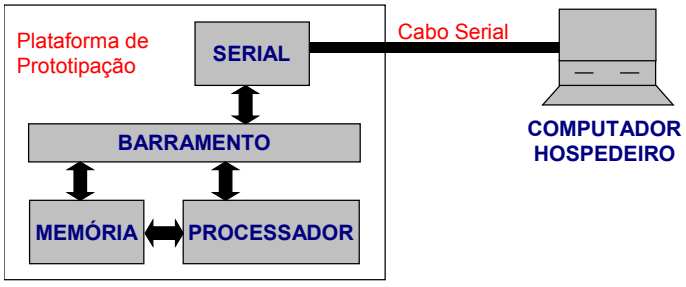


Figura 13 – Arquitetura do sistema monoprocessado.

Com a arquitetura do sistema monoprocessado apresentado na Figura 13 é possível

realizar as seguintes operações:

1. Utilizando o software Serial o usuário pode carregar um programa em código objeto na memória do sistema.
2. Utilizando o software Serial o usuário pode habilitar o processador, para que o mesmo comece a executar um programa.
3. O processador, após a habilitação, executa a ação correspondente a cada instrução até que alcance a instrução de "halt", responsável por finalizar a execução do processador. As ações podem ser operações de entrada e saída e utilizarem o barramento para acessar o núcleo Serial.
4. A verificação da correta execução do programa pelo processador pode ser feita através de instruções de entrada/saída inseridos no programa ou através da leitura dos dados, via software Serial, armazenados em memória.

A inclusão do processador exige as seguintes tomadas de decisão quanto ao projeto da arquitetura:

- Política de acesso ao barramento: um único mestre ou vários mestres.
- Política de acesso do processador à memória: pelo barramento compartilhado por todos os núcleos do sistema ou utilização de um novo barramento processador-memória.

A política de acesso ao barramento é necessária porque sem a mesma os vários núcleos que desejassem se comunicar poderiam escrever no barramento, ocasionando colisões entre os acessos. Este acesso simultâneo pode ser evitado com a introdução no sistema de um ou mais mestres do barramento. O mestre do barramento tem a finalidade de iniciar e monitorar todas as requisições feitas ao barramento [PAT98]. Em um sistema com um único mestre, todas as solicitações de uso do barramento precisam ser controladas pelo mestre. O maior problema desse esquema está no fato deste mestre estar necessariamente envolvido em todas as transações ocorridas com o barramento. Optou-se então por adotar um esquema alternativo que envolve diversos mestres do barramento e um esquema de arbitragem. Neste esquema cada um dos diversos mestres pode iniciar uma transferência, logo deve-se estabelecer um mecanismo de arbitragem para disciplinar o acesso ao barramento, de forma que ele possa ser usado de modo cooperativo. O mecanismo de arbitragem adotado é detalhado no Capítulo 3.

Para otimizar os acessos do processador ao barramento propõem-se que este tenha acesso direto a uma memória local. Desta forma, o processador fará uso do barramento apenas nas instruções de entrada/saída, proporcionando um ganho significativo de desempenho para o sistema. A comunicação direta entre processador e memória requer a implementação de um barramento do tipo processador-memória, como caracterizado na Seção 1.3.1. Para esta modificação é necessário introduzir uma lógica adicional para controlar o acesso a memória, que agora pode ser acessada através do barramento processador-memória ou através do barramento do sistema.

O projeto do sistema monoprocessado permitiu ao grupo validar o processador em hardware, a comunicação do mesmo com a memória, a sua comunicação com o módulo serial e as novas políticas de acesso ao barramento mais complexas e mais flexíveis.

2.3 Sistema Multiprocessado com Interconexão por Barramento

Após o projeto do sistema monoprocessado é possível estendeu-se a capacidade do sistema, adicionando novos processadores e memórias. A Figura 14 ilustra um sistema multiprocessado com quatro processadores e quatro memórias.

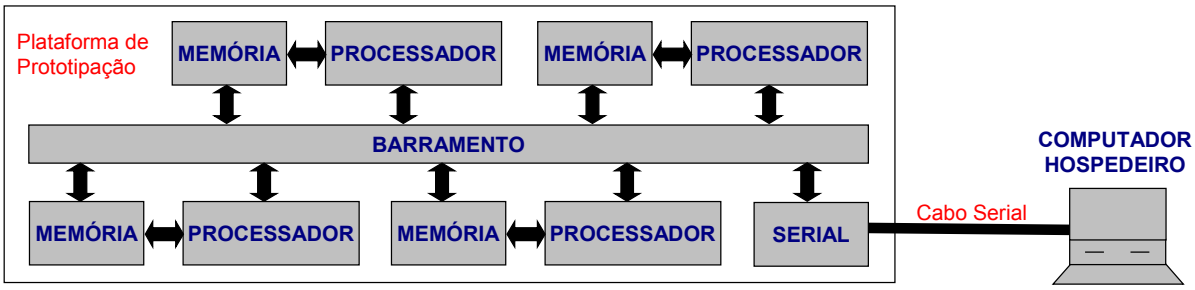


Figura 14 – Arquitetura do sistema multiprocessado.

A inclusão de diversos processadores ao barramento implicará nas seguintes tomadas de decisão quanto ao projeto da arquitetura multiprocessada:

- Sincronização entre processadores que executam em paralelo.
- Compartilhamento de dados entre processadores que executam em paralelo.
- Arquitetura de memória: UMA, NUMA ou NORMA.

A sincronização entre os processadores pode ser obtida através de memória compartilhada ou troca de mensagens, apresentadas nas seções 1.2.1 e 1.2.2, respectivamente. Optou-se por adotar a técnica de troca de mensagens, onde rotinas de *send* e *receive* são utilizadas para envio e recepção de mensagens. A sincronização é obtida em um sistema de troca de mensagens considerando que um processador transmissor sabe quando uma mensagem é enviada, e um outro processador receptor sabe quando uma mensagem chega [PAT98].

Para o compartilhamento de dados entre os processadores optou-se por uma solução DSM. DSM permite que os processadores possam acessar as diversas memórias do sistema, de forma que as posições das mesmas sejam referenciadas por endereços únicos. Deste modo, o processador deve saber o intervalo de endereçamento referente a cada memória do sistema, para que ao requisitar o acesso a uma posição da memória, analise se a posição de memória requisitada pertence à memória local ou a remota. Se pertencer à memória remota o endereço deve ser convertido para o endereço global e o acesso deve ser feito através do barramento do sistema. Caso contrário, o endereço não necessita de conversão e o acesso é através do barramento processador-memória.

Baseado na Figura 14, alguns acessos à memória compartilhada são mais rápidos do que outros, caracterizando uma máquina NUMA, sem memória centralizada, como apresentado na Seção 1.2.1.2. A diferença entre os tempos de acesso à memória ocorre quando o processador deseja acessar uma posição da DSM que pertence à memória local. Deste modo, a memória local é vista como uma cache, pois apresenta vantagens devido à sua localidade de acesso. Cabe lembrar que o sistema é parametrizável, atendendo a modelagem dada pelo seu projetista. Por exemplo, é possível a adição de memórias globais centralizadas ao barramento padrão, como apresentado na Seção 1.2.1.1 que trata sobre máquina UMA.

2.4 Sistema Biprocessado com Interconexão por NoC

O primeiro passo para o desenvolvimento da rede intra-chip (NoC) é o estudo das diferentes topologias de interconexão de chaves. A topologia é definida pela forma com que as chaves são ligadas, conforme visto na Seção 1.3.3. Dependendo da topologia, as chaves possuem um número definido de portas e podem executar diferentes classes de algoritmos de chaveamento. Os algoritmos de chaveamento determinam qual o caminho para a transmissão de pacotes na rede. As chaves, independentemente da topologia, podem possuir filas para o armazenamento temporário de pacotes. Essas filas tem o objetivo de reduzir a contenção dos pacotes na rede.

Dentre as topologias estudadas optou-se por utilizar a Malha. Essa escolha é justificada pela sua regularidade física desta e a conseqüente facilidade de implementação do roteamento no FPGA. As demais decisões de projeto que envolvem a implementação de uma rede são apresentadas no Capítulo 8.

O sistema interconectado por NoC utilizou o mesmo conjunto de núcleos apresentados nos sistemas interconectados por barramento, conforme ilustrado na Figura 15. É importante salientar que embora os núcleos de hardware sejam reutilizados, é necessário realizar a adaptação dos mesmos ao protocolo de comunicação adotado pela rede.

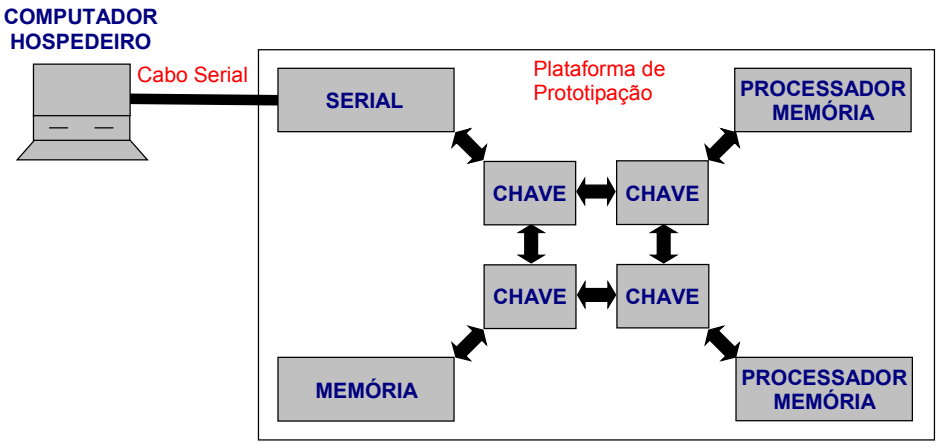


Figura 15 – Arquitetura do sistema interconectado por rede intra-chip.

O projeto do sistema biprocessado com interconexão por NoC permitiu validar a chave e a rede intra-chip desenvolvida, através do correto envio e recebimento de mensagens verificados pelo computador hospedeiro. O projeto também permitiu a comparação do desempenho da NoC em relação ao barramento.

2.5 Plataforma de Prototipação

A prototipação dos sistemas projetados em hardware é um dos requisitos iniciais deste trabalho. Para prototipar um sistema digital utiliza-se FPGAs de alta densidade de integração. Para isto, escolheu-se como plataforma de prototipação a V2MB1000 da empresa Memec Insight [MEM03]. As razões para a escolha desta plataforma foram: FPGA com grande capacidade em termos de portas lógicas, comunicação com o computador hospedeiro através da porta serial (o que simplifica o desenvolvimento do software de comunicação) e principalmente a sua disponibilidade no grupo de pesquisa GAPH.

A plataforma da fabricante Memec (Figura 16), contém o FPGA (Field-Programmable Gate Array) XC2V1000 fabricado pela Xilinx [XIL02]. Este dispositivo permite reconfiguração parcial e dinâmica, possui capacidade para implementar circuitos com dimensões de cerca de 1 milhão de portas lógicas equivalentes e 40 blocos de memória RAM (BlockRAMs) de 18 Kbits cada.

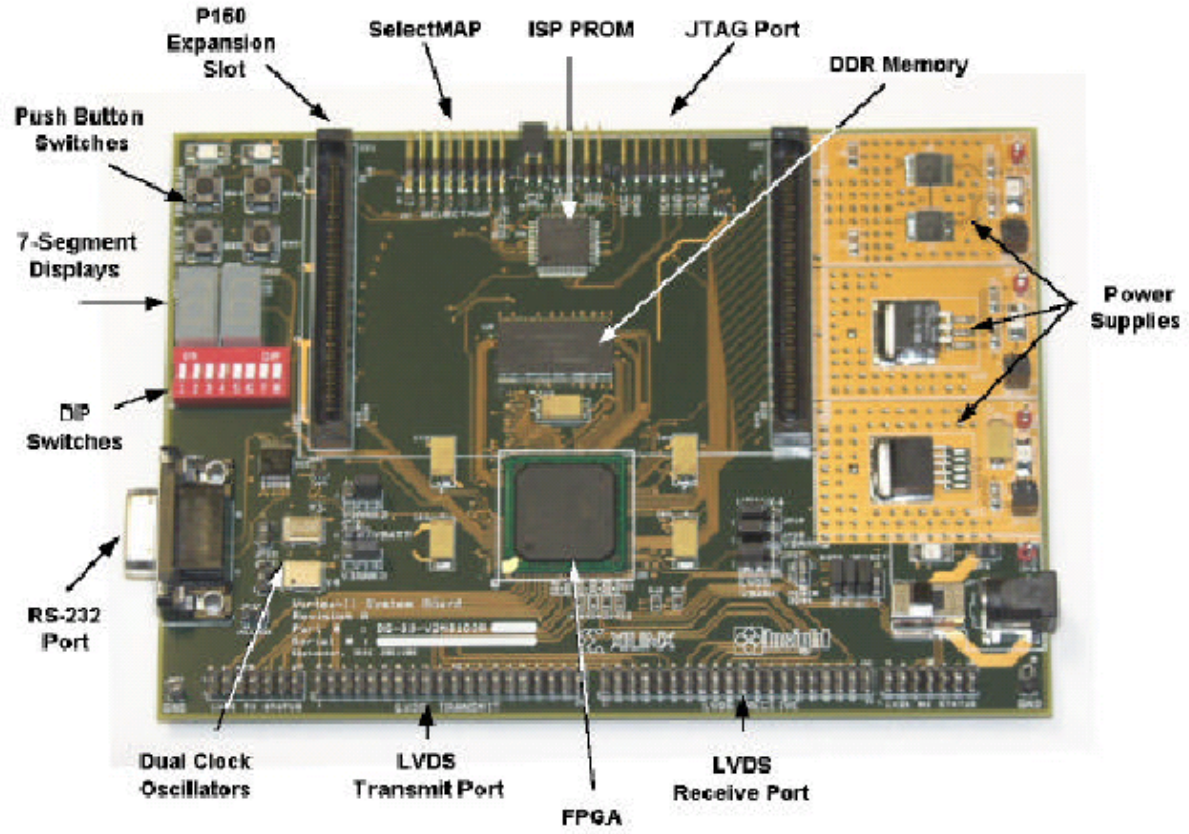


Figura 16 – Plataforma de prototipacão V2MB1000 Memec-Insight, com FPGA XC2V1000.

3 Arbitragem e Arquitetura do Barramento

Denomina-se arbitragem de barramento a decisão sobre qual dos dispositivos vai obter o controle do barramento. Os esquemas de arbitragem tentam balancear dois fatores na escolha do próximo dispositivo a usar o barramento. Primeiramente, cada dispositivo tem a sua prioridade de acesso ao barramento, sendo que o dispositivo de mais alta prioridade deve ser servido em primeiro lugar. Segundo, deve ser garantido que todo o dispositivo, mesmo aquele com a prioridade mais baixa, tenha acesso ao barramento. A observância desses princípios garante que todo dispositivo que deseje usar o barramento terá acesso a ele [PAT98].

A tarefa neste capítulo é apresentar o mecanismo de arbitragem adotado para que os dados trafeguem corretamente entre os núcleos. A Seção 3.1 apresenta a estrutura do barramento que interconecta os núcleos de hardware. Dois protocolos de acesso ao barramento são discutidos na Seção 3.2 e a Seção 3.3 mostra o esquema de recebimento e envio de pacotes pelo barramento.

3.1 Barramento

O barramento interconecta todos os núcleos do sistema. Esta interconexão se dá por um conjunto de fios globais que cruzam o sistema levando informações de dados e controle aos núcleos. O conjunto de sinais utilizado pelo barramento no presente projeto é ilustrado na Figura 17 e comentado a seguir.

Figura 17 – Exemplo de dois núcleos de hardware conectados ao barramento.

- *request*: sinal ativado quando núcleo deseja acesso ao barramento.
- *grant*: sinal ativado quando o uso do barramento é concedido para o núcleo.
- *release*: sinal ativado quando núcleo terminar o uso do barramento.
- *busy*: sinal que permanecerá ativado enquanto o barramento estiver sendo utilizado.
- *abort*: sinal ativado quando um núcleo excede o tempo máximo de acesso ao barramento.
- *dataIn*: sinal de entrada de dados/endereços no barramento de 16 bits.
- *dataOut*: sinal de saída de dados/endereços do barramento de 16 bits.

Os sinais *busy*, *abort*, *dataIn* e *dataOut* são compartilhados por todos os módulos, enquanto os sinais *request*, *grant* e release são individuais.

A Figura 18 apresenta um diagrama de tempos exemplificando o funcionamento do acesso ao barramento via simulação funcional. Para que um núcleo (transmissor) possa se comunicar com outro (receptor), o mesmo ativa o sinal de *request* e espera que o sinal *grant* seja ativado (1). O árbitro do barramento ao detectar o *request* do módulo transmissor ativo,

ativa os sinais *grant* do transmissor e *busy* (2). Quando o transmissor detecta o sinal de *grant* ativo, o mesmo desativa o sinal de *request* (3) e transmite os dados pelo sinal *dataOut* (4). Ao final da transferência dos dados o transmissor ativa o sinal *release*(5), o árbitro detecta o mesmo e desativa o sinal *busy* (6). Enquanto o sinal *busy* estiver ativo, o receptor captura os dados do barramento no sinal *dataIn*. É importante salientar que o barramento é síncrono, pois toda a transmissão de um dado dura um ciclo de relógio. O sinal *abort* será visto na próxima Seção.

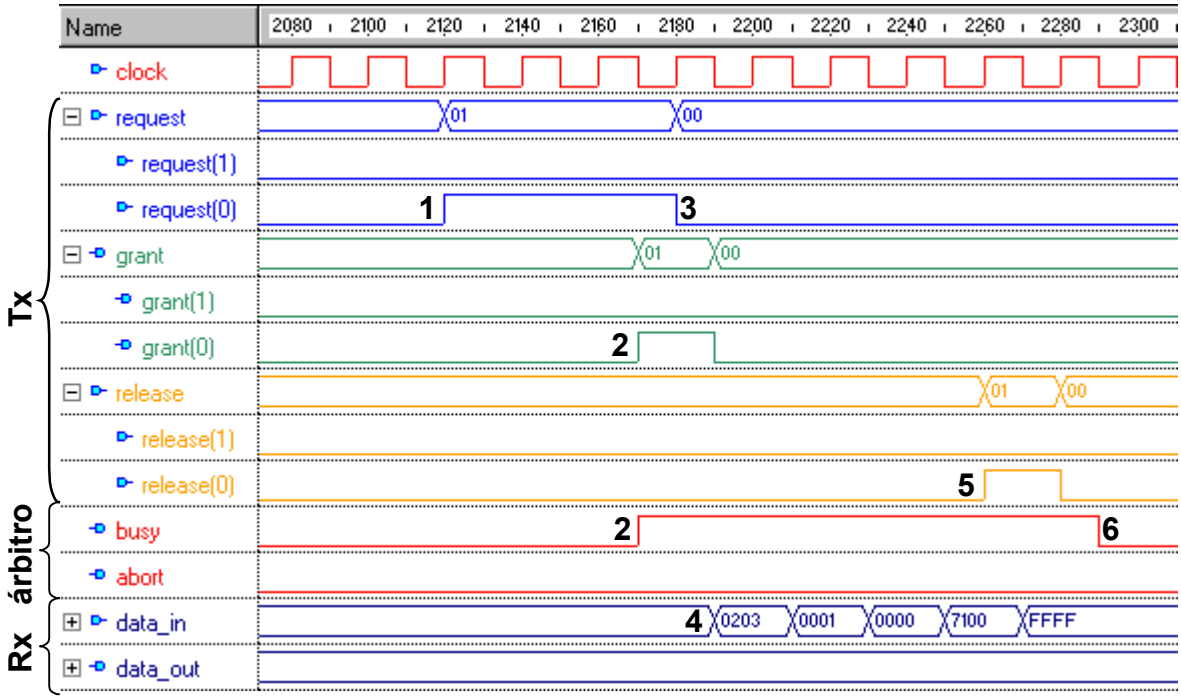


Figura 18 – Diagrama de tempos dos eventos para acesso ao barramento.

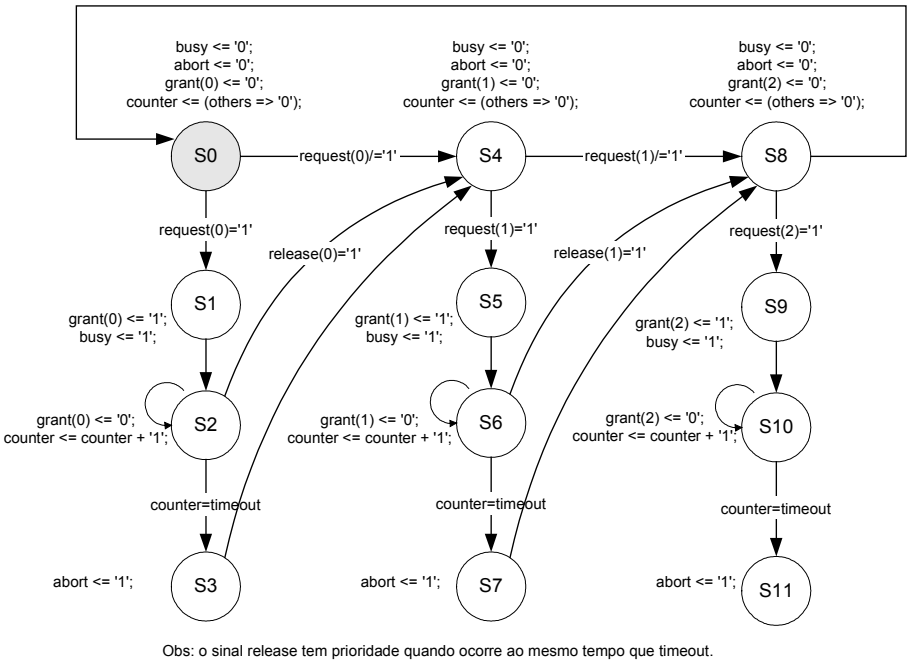
3.2 Árbitro do barramento

Conforme visto na Seção anterior é necessário um protocolo de acesso ao canal de dados. Esta Seção apresenta quatro protocolos de arbitragem para acesso a um barramento [PAT98]:

- Daisy chain: neste esquema, a linha com a informação da garantia de uso do barramento está ligada a todos os dispositivos conectados ao barramento, a partir do dispositivo de mais alta prioridade até chegar ao de mais baixa (as prioridades são determinadas pela posição dos dispositivos no barramento). Um dispositivo de alta prioridade que deseje acesso ao barramento simplesmente intercepta o sinal de garantia de acesso, não permitindo que os dispositivos de mais baixa prioridade enxerguem tal sinal.
- Arbitragem centralizada com requisição em paralelo: cada dispositivo tem uma linha direta de solicitação do barramento ao árbitro central. O árbitro central escolhe um entre os dispositivos solicitantes e autoriza o acesso a este.
- Arbitragem distribuída com acesso por auto-seleção: este esquema também utiliza múltiplas linhas de solicitação. Os dispositivos colocam o seu código no barramento e através da análise dos códigos presentes no mesmo, o dispositivo de maior prioridade toma o barramento.
- Arbitragem distribuída via detecção de colisão: neste esquema, cada dispositivo pode

solicitar o barramento independentemente. Múltiplos dispositivos solicitando o barramento ao mesmo tempo causam uma colisão. Colisões são detectadas pelos próprios dispositivos e estes usam um esquema para saber qual deles tomará o barramento.

Assumindo o exemplo ilustrado na Figura 17, no qual o núcleo 0 deseja comunicar-se com o núcleo 1, pode-se utilizar uma arbitragem centralizada com requisição em paralelo e prioridade rotativa, de forma a garantir que não haja colisão de dados no barramento compartilhado. Um esquema simples para não permitir que essa colisão ocorra é implementar um árbitro cuja máquina de estados é ilustrada na Figura 19 e explicada a seguir.



Obs: o sinal release tem prioridade quando ocorre ao mesmo tempo que timeout.

Figura 19 – Máquina de estados do árbitro do barramento, para atender a requisições de 3 núcleos.

Na Figura 19, os estados S0 à S3 correspondem ao núcleo 0. Analogamente S4 à S7 são referentes ao núcleo 1. É relevante observar que para inserir novos núcleos no sistema necessita-se replicar a mesma estrutura de estados, como pode ser visto nos estados S8 à S11. Enquanto nenhum dos núcleos requisitar acesso ao barramento, seus sinais de *request* permanecem desativados. Portanto, o sistema fica alternando entre os estados S0, S4 e S8. Quando o núcleo 0 requisita acesso para transmissão de dados, ele ativa o sinal de *request(0)*. Quando a máquina de estados do árbitro chega ao estado S0 e verifica que *request(0)* está ativo, avança para o estado S1. No estado S1 os sinais *busy* e *grant(0)* são ativados, indicando que o barramento está sendo utilizado e que é o núcleo 0 que tem permissão para escrever no mesmo. Do estado S1 a máquina avança automaticamente para o estado S2, onde o sinal *grant(0)* é desativado e onde a máquina de estados permanece até que o sinal de *release(0)* seja ativado ou até que expire o tempo máximo de acesso de um núcleo ao barramento. Quando ocorre a primeira condição, ou seja, o núcleo 0 libera o barramento através da ativação do *release(0)*, a máquina avança do estado S2 automaticamente para o estado S4. Caso contrário, é necessária a comparação entre o tempo máximo de acesso ao barramento (*timeout*), que é dado em número de ciclos de relógio, com o tempo de permanência da

máquina de estados no estado S2, que também é contabilizado em número de ciclos de relógio. Quando estes valores se igualam a máquina de estados avança do estado S2 para S3. No estado S3 o sinal *abort* é ativado, retirando a permissão de acesso que o núcleo detinha, e a máquina avança automaticamente para o estado S4.

A Figura 20 mostra o diagrama de tempos dos eventos ocorridos no parágrafo anterior. Observar que a requisição do núcleo 0, simultânea a requisição do núcleo 2, só é atendida quando o núcleo 2 entregar o barramento (*release*(2)). O sinal ES indica o estado da máquina de arbitragem, conforme a Figura 20.

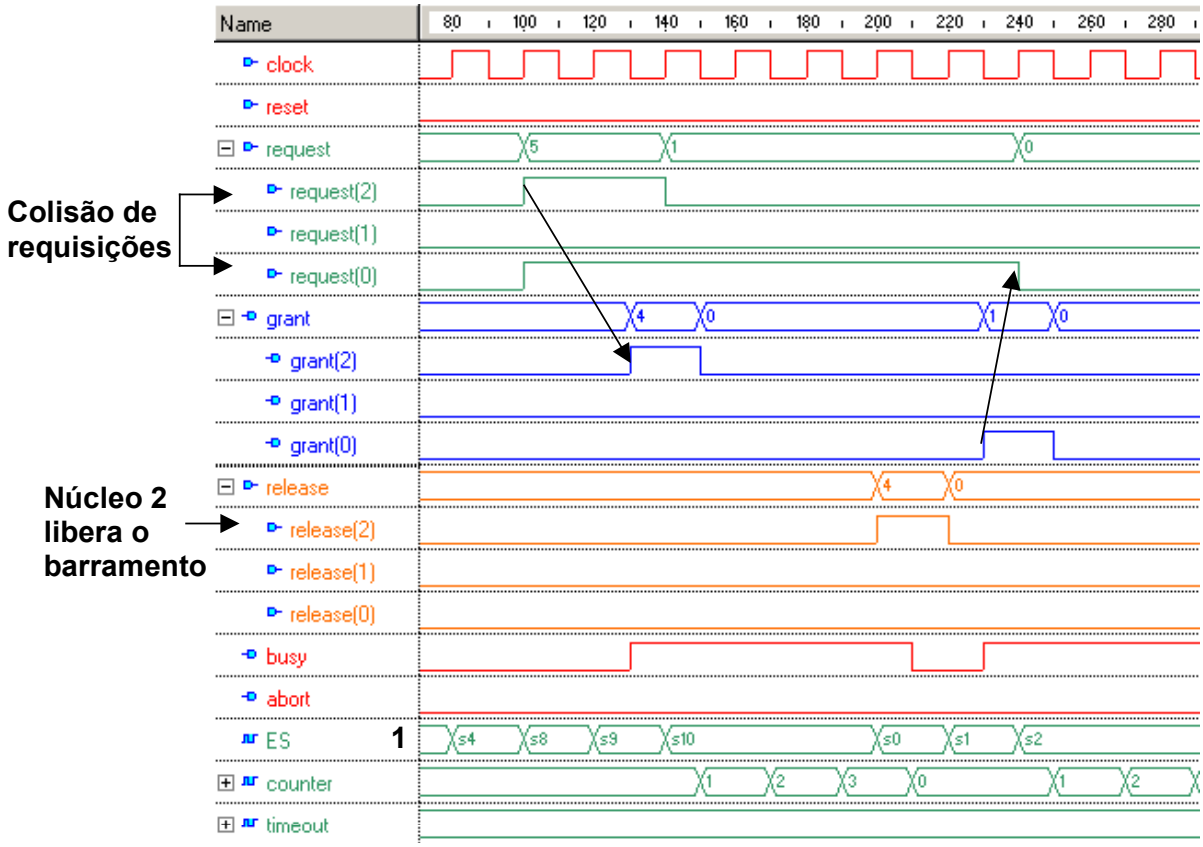


Figura 20 – Diagrama de tempos dos eventos de uma transmissão.

É possível observar na Figura 20(1), que enquanto nenhum núcleo requisita acesso ao barramento, o árbitro executa um *polling*, verificando o sinal de *request* de cada um dos núcleos conectados ao barramento. Isto significa que se existirem 64 núcleos conectados ao barramento e o sinal de *request* do núcleo 64 for ativado imediatamente após o mesmo ser testado, serão necessários no mínimo 63 ciclos de relógio para que o núcleo seja atendido.

Uma otimização do árbitro pode ser implementada a partir da idéia de arbitragem dinâmica rotativa. Considere quatro núcleos conectados ao barramento e os seguintes sinais:

- *sel*: número do último núcleo que obteve a permissão de escrita no barramento.
- *prox*: número do próximo núcleo a receber permissão de escrita no barramento.

A utilização da arbitragem dinâmica rotativa permite que a requisição de acesso de um núcleo seja atendida no mesmo ciclo de relógio. Caso ocorram requisições em paralelo, o núcleo que possuir a maior prioridade terá sua requisição atendida. É importante observar que a prioridade dos núcleos é variável e dependente do último núcleo que teve a permissão de

acesso. Isto significa que todos os núcleos serão atendidos, mesmo os de menor prioridade.

Partindo do pressuposto que o núcleo 1 foi o último a possuir permissão de escrita no barramento, enquanto ninguém requisitar acesso ao barramento o sinal *sel* será igual a 1 e o sinal *prox* será igual à 1. Quando, por exemplo, os núcleos 0 e 3 requisitarem, ao mesmo tempo, o acesso ao barramento, ativando os sinais *request(0)* e *request(3)*, respectivamente, então o sinal *sel* será analisado para determinar qual dos núcleos terá o acesso. Podemos observar na Figura 21, a partir da linha 13, que quando o sinal *sel* é igual a 1, a prioridade dos núcleos em ordem decrescente é 2, 3, 0 e por fim 1. Desta forma, a requisição de acesso do núcleo 3 será atendida antes da requisição do núcleo 0.

```
01 if( request[0]==´1´ || request[1]==´1´ || request[2]==´1´ || request[3]==´1´){
02        grant[prox]=´1´;
03        sel = prox;
04 }
05
06 switch(sel){
07        case 0:
08               if(request[1]==´1´) prox=1;
09               else if(request[2]==´1´) prox=2;
10               else if(request[3]==´1´) prox=3;
11               else prox=0;
12               break;
13        case 1:
14               if(request[2]==´1´) prox=2;
15               else if(request[3]==´1´) prox=3;
16               else if(request[0]==´1´) prox=1;
17               else prox=1;
18               break;
19        case 2:
20               if(request[3]==´1´) prox=3;
21               else if(request[0]==´1´) prox=0;
22               else if(request[1]==´1´) prox=1;
23               else prox=2;
24               break;
25        case 3:
26               if(request[0]==´1´) prox=0;
27               else if(request[1]==´1´) prox=1;
28               else if(request[2]==´1´) prox=2;
29               else prox=3;
30               break;
31 }
```

Figura 21 – Exemplo de código fonte em linguagem C para definição do próximo núcleo a acessar o barramento.

3.3 Módulo Send/Receive

O módulo Send/Receive é um módulo genérico que conectado a outro módulo do sistema (como processador, memória) compõe um núcleo. Ele contém a lógica de envio e recebimento de pacotes, assim como a montagem, o envio e o recebimento de confirmações *(ack)*. A Figura 22 ilustra este módulo e seus respectivos sinais são detalhados a seguir.

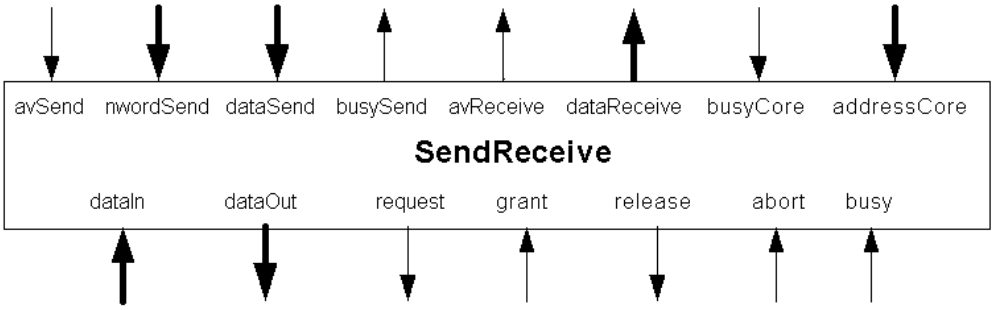


Figura 22 – Módulo Send/Receive.

- *avSend*: sinaliza que o núcleo tem um pacote a ser enviado através do módulo Send.
- *nWordSend*: número de palavras do pacote a ser enviado.
- *dataSend*: vetor correspondente ao pacote a ser enviado. Este vetor pode conter até 8 palavras de 16 bits.
- *busySend*: sinaliza que o módulo Send está transmitindo pacote.
- *avReceive*: indica que o módulo Receive recebeu pacote destinado ao núcleo.
- *dataReceive*: vetor correspondente ao pacote recebido.
- *busyCore*: indica que o núcleo não pode receber pacotes no momento.
- *addressCore*: endereço do núcleo ao qual o módulo Send/Receive está vinculado.

O módulo Send/Receive possui três máquinas de estados: a máquina de estados Send, a máquina de estados Receive e a máquina de estados Ack.

A máquina de estados Send é responsável por transmitir um pacote do núcleo origem a outro núcleo do sistema e realizar retransmissões do mesmo, até que o núcleo destino transmita um pacote de confirmação de recebimento.

A máquina de estados Ack é responsável por enviar pacotes de confirmação de recebimento quando o núcleo vinculado ao módulo Send/Receive recebe um pacote, desde que este pacote não seja de confirmação.

A máquina de estados Receive é responsável por receber os pacotes destinados ao núcleo vinculado ao módulo Send/Receive. Esta máquina verifica se o pacote é uma confirmação de recebimento ou não. Quando o pacote é uma confirmação de recebimento a máquina de estados Send é avisada, pois esta deve estar esperando por esta confirmação. Quando o pacote não é de confirmação é verificado se o núcleo pode receber o pacote no momento, porque somente quando este puder receber o pacote é avisada a máquina de estados Ack que envie a confirmação de recebimento do mesmo. Se o núcleo não puder receber o pacote, o pacote é desprezado pela máquina de estados Receive.

A máquina de estados Send (Figura 23) e seu funcionamento (Figura 24) são comentados a seguir.

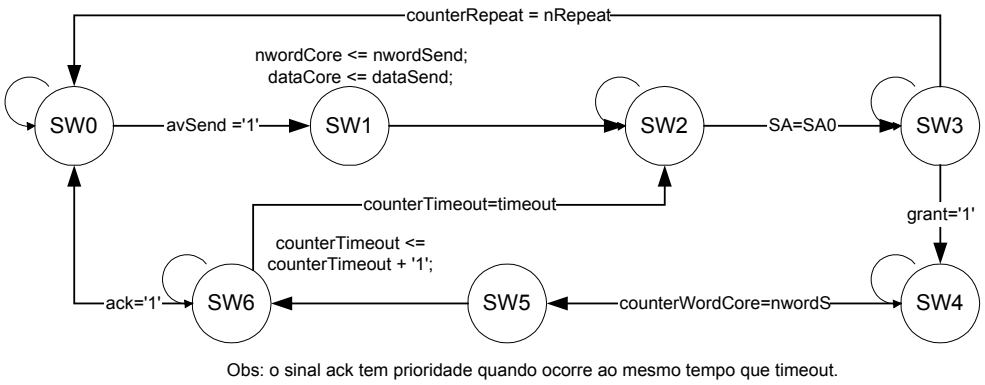


Figura 23 – Máquina de estados Send.

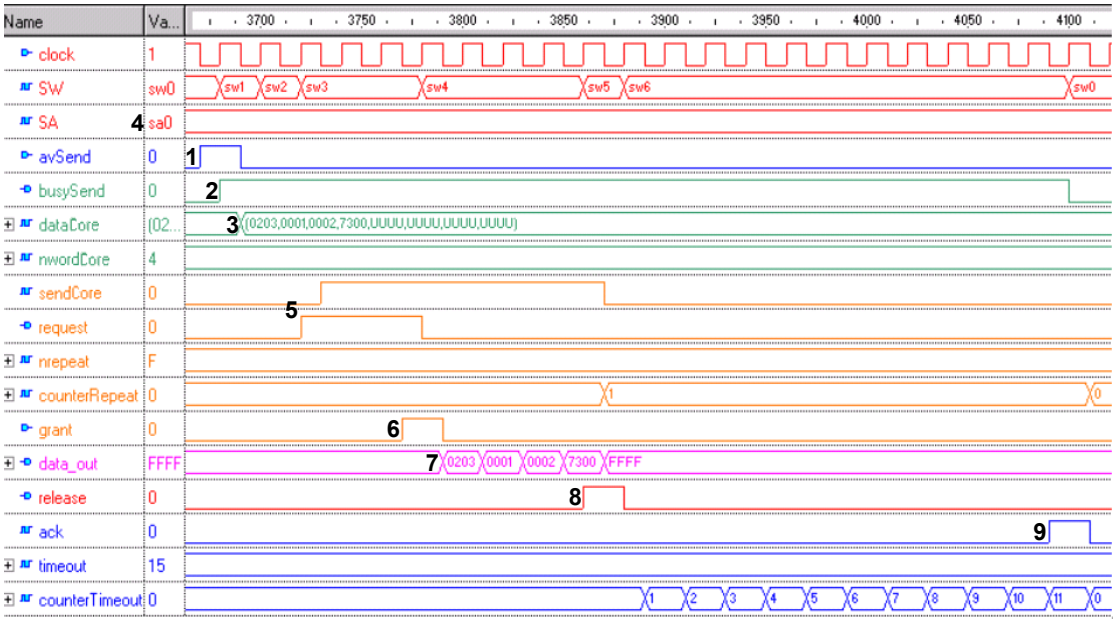


Figura 24 – Diagrama de tempos dos eventos da máquina Send.

1. O estado SW0 é o estado de repouso da máquina de estados Send (SW). SW deixa o estado de repouso avançando para SW1 quando o núcleo conectado ativa o sinal *avSend* (1 da Figura 24), indicando que o mesmo possui um pacote para transmitir.

2. No estado SW1 o sinal *busySend* é ativado (2) e são capturados o pacote e o número de palavras do mesmo (3). Deste estado a máquina avança automaticamente para o estado SW2.

3. No estado SW2 é verificado se a máquina de estados SA (Ack) (4), responsável pelo envio de confirmações, encontra-se no estado de repouso (SA0). Esta verificação é necessária, porque não podem ocorrer transmissões simultâneas. Logo a máquina de estados SW somente avançará do estado SW2 para SW3, quando esta condição for respeitada.

4. No estado SW3 os sinais *request* e *sendCore* são ativados (5), informando que o núcleo solicita acesso ao barramento e que a máquina de estados SA não poderá realizar transmissões neste momento. Neste estado é controlado o número de retransmissões do pacote, que não deve exceder ao valor armazenado em *nrepeat***.** Caso o número de

retransmissões exceda este valor, a máquina de estados SW retorna ao estado de repouso. Caso contrário a máquina de estados avança para SW4, quando receber a permissão de acesso ao barramento, ou seja, quando o sinal *grant* for ativado (6).

5. No estado SW4 ocorre a transmissão do pacote (7). A máquina SW permanecerá neste estado **n** ciclos de relógio, onde **n** é o número de palavras do pacote a ser transmitido. Isto porque, a cada ciclo de relógio uma posição do vetor, que contem o pacote, é enviada pelo barramento. Ao final dos **n** ciclos de relógio a máquina avança para SW5.
6. No estado SW5 o módulo libera o barramento ativando o sinal *release* (8), desativa o sinal *SendCore*, incrementa o número de transmissões e avança para o estado SW6.
7. SW6 é o estado de espera pelo recebimento da confirmação (9). A máquina de estados permanece em SW6 até que o *ack* seja recebido ou até que expire o tempo máximo para recepção do mesmo. Portanto, é necessária a comparação entre o tempo máximo de espera pela confirmação (*timeout*), que é dado em número de ciclos de relógio, com o tempo de permanência da máquina de estados em SW6, que também é contabilizado em número de ciclos de relógio. Quando estes valores se igualam, a máquina de estados retorna ao estado SW2 para realizar a retransmissão do pacote. Caso o *ack* seja recebido antes deste valor ser alcançado, a máquina de estados SW retorna ao estado de repouso (SW0).

A máquina de estados Receive e seu funcionamento podem ser visualizados na Figura 25 e na Figura 26, respectivamente, e é comentada a seguir.

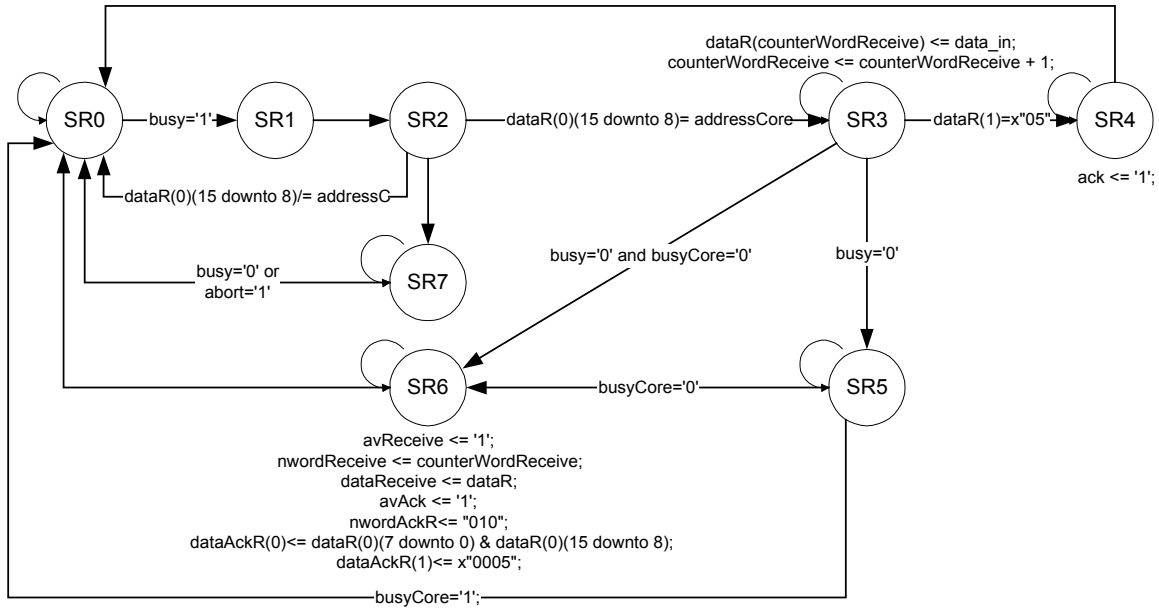


Figura 25 – Máquina de estados receive.

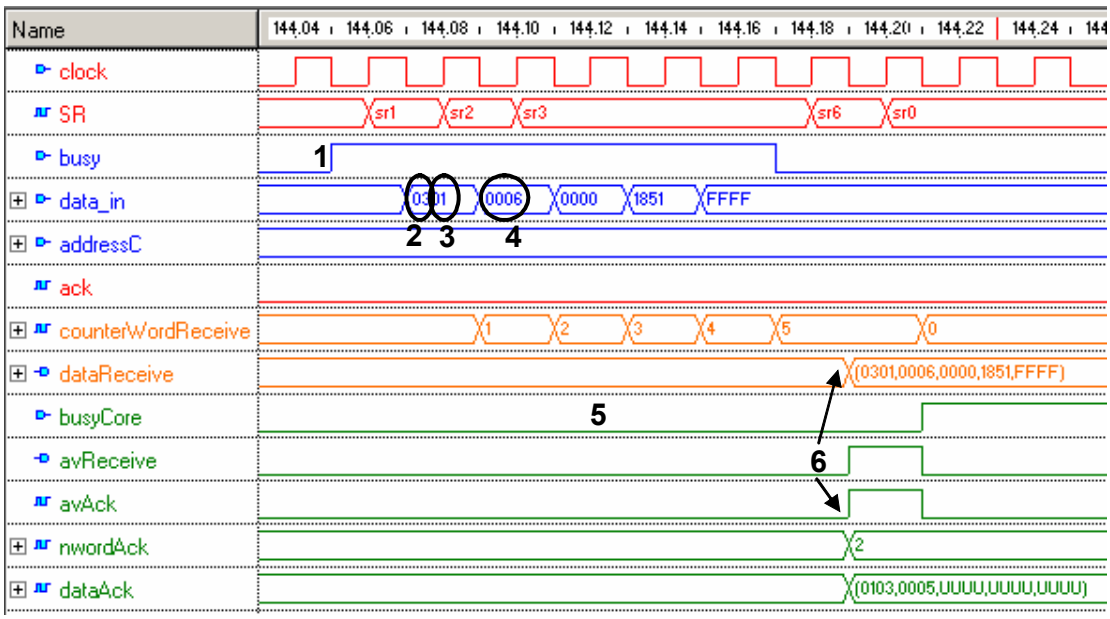


Figura 26 – Diagrama de tempos dos eventos da máquina Receive.

1. A máquina de estados Receive (Figura 25) inicia no estado SR0, onde os sinais estão desativados. Quando o sinal *busy* é ativado (1 da Figura 26), significa que o barramento está sendo utilizado e por isso a máquina de Receive avança para o estado SR1.
2. Do estado SR1 avança-se automaticamente para o estado SR2. A passagem pelo estado SR1 é necessária porque somente um ciclo de relógio após o sinal *busy* ter sido ativado, o núcleo que possui acesso de escrita no barramento, envia a primeira palavra do pacote.
3. Em SR2 é avaliada a primeira palavra do pacote recebido. Esta palavra contém nos 8 bits mais significativos o endereço do núcleo destino do pacote (2) e nos 8 bits menos significativos o endereço do núcleo origem do pacote (3). Se o endereço do núcleo destino do pacote for diferente do endereço do núcleo (*addressCore*), o pacote deve ser desprezado, avançando para SR7, onde a máquina Receive permanece até que o barramento seja liberado (*busy* desativado) ou o sinal *abort* ativado. Se o endereço de destino for igual ao endereço do núcleo, então a máquina avança para o estado SR3.
4. No estado SR3, é verificada a segunda palavra do pacote (4), que contem o comando a ser tratado. Caso o comando seja igual a 5, indicando que é um pacote de confirmação (*ack*), a máquina de estados SR avança para SR4. Caso o comando não corresponda a um pacote de confirmação, as palavras recebidas através do barramento são montadas em um pacote (*dataReceive*) até que o sinal *busy* seja desativado. Quando isto ocorre, a máquina avança para o estado SR5.
5. No estado SR4, a máquina de estados SW (responsável pelo envio de pacotes) que aguarda esta confirmação é avisada do seu recebimento, através da ativação do sinal *ack*. Do estado SR4 a máquina de estados SR avança automaticamente para o estado de repouso (SR0).
6. Em SR5 é verificado se o núcleo associado ao Send/Receive pode receber pacotes no momento (5). Esta verificação é realizada através do sinal *busyCore*. Quando *busyCore* estiver ativo, indicando que o núcleo não pode receber pacotes no momento, a máquina SR despreza o pacote recebido, não envia o *ack* e retorna ao estado de repouso. Caso o

núcleo possa receber pacotes a máquina de estados avança para SR6.

7. No estado SR6 o sinal *avReceive* é ativado e o pacote recebido é disponibilizado ao núcleo (6), juntamente com o número de palavras que o compõe. É também no estado SR6 que é montado o pacote de confirmação do recebimento e é solicitado à máquina de estados SA que envie esta confirmação através da ativação do sinal *avAck*.

A Figura 27 ilustra a máquina de estados Ack, responsável pelo envio de confirmações de recebimento, seu funcionamento é apresentado na Figura 28, e é comentado a seguir.

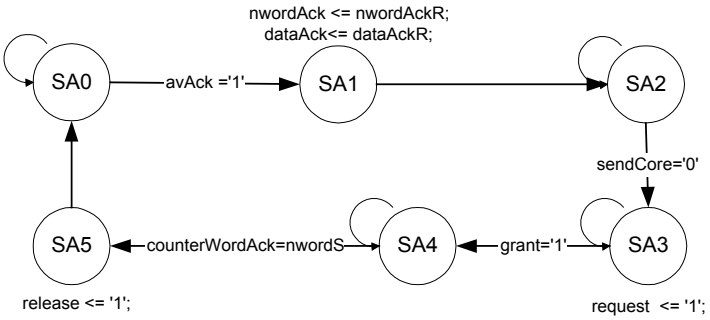


Figura 27 – Máquina de estados Ack.

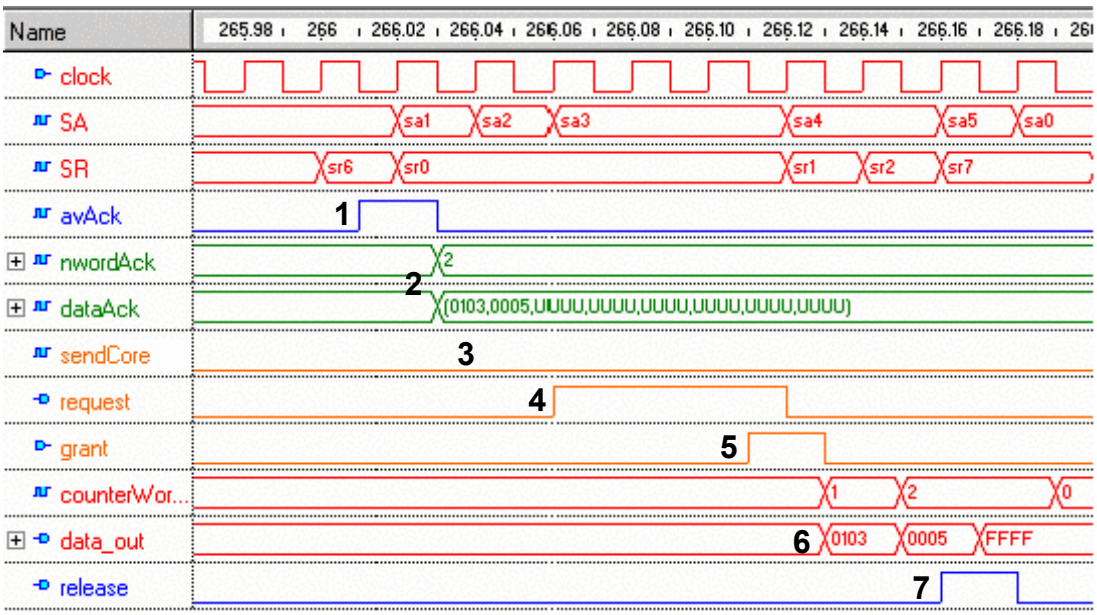


Figura 28 – Diagrama de tempo dos eventos da máquina Ack.

1. O estado SA0 é o estado de repouso da máquina de estados SA, onde esta permanece até que o sinal *avAck* seja ativado pela máquina Receive. Quando avAck é ativado (1), indicando que existe o pedido de envio de confirmação feito pelo estado SR6 da máquina de Receive, a máquina de estados SA avança para o estado SA1.
2. No estado SA1 o pacote montado pela máquina de Receive é capturado, juntamente com o número de palavras do mesmo (2) e a máquina avança automaticamente para o estado SA2.
3. Do estado SA2 a máquina somente avança para o estado SA3 quando a máquina Send não

estiver usando o barramento, ou seja, quando o sinal *sendCore*(3) estiver desativado.

4. No estado SA3 é realizada a requisição de acesso ao barramento (4) e a máquina de estados SA avança para SA4 quando recebe a liberação de acesso ao mesmo (5).
5. O estado SA4 envia o pacote, palavra a palavra, pelo barramento (6). Quando o número de palavras do pacote for igual ao número de ciclos de relógio de permanência da máquina em SA4, o pacote foi inteiramente transmitido e a máquina de estados avança para SA5.
6. Em SA5 é liberado o barramento (7) e a máquina retorna ao estado de repouso (SA0).

4 Comunicação com o Computador Hospedeiro

Foi visto a partir da Seção 2.1 que o sistema implementado na plataforma de prototipação tem comunicação com um computador hospedeiro. Esta comunicação é responsável por fornecer um meio de interação do usuário com os elementos internos ao sistema, tais como processadores, memórias ou qualquer outro módulo que tenha sido conectado ao barramento. A Figura 29 ilustra a bancada de trabalho para utilização do sistema assim como as conexões do mesmo com um computador hospedeiro. Esta figura mostra o software (1) implementado para comunicação com a interface serial padrão RS-232, o cabo (2) que liga a interface serial do computador com a interface serial da plataforma de prototipação (3) e o dispositivo FPGA (4) onde o sistema é executado.

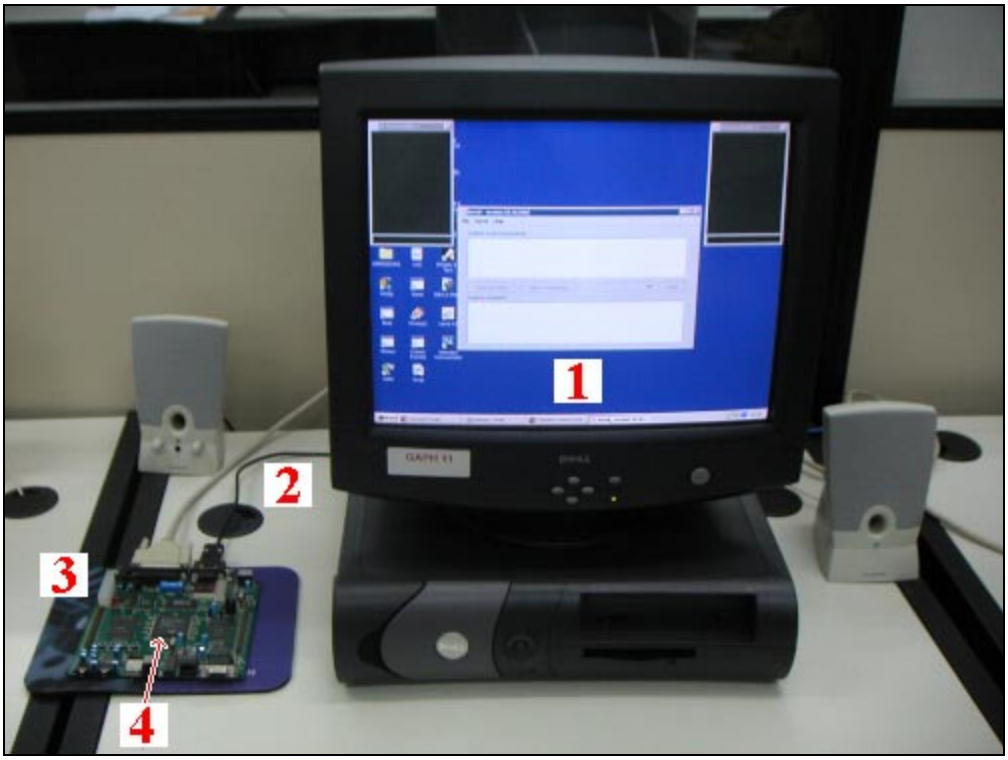


Figura 29 – Bancada de desenvolvimento do sistema.

A Seção 4.1 mostra o funcionamento do software que implementa a comunicação serial com a plataforma de prototipação. A Seção 4.2 apresenta o módulo Serial que recebe dados do computador hospedeiro e também envia dados ao mesmo. A Seção 4.3 apresenta o núcleo Serial que trata os dados recebidos do módulo Serial e através do barramento.

4.1 Sistema Hospedeiro

O sistema hospedeiro é um software elaborado pelo grupo para enviar e receber dados pela interface serial do computador. A Figura 30 mostra a tela principal do programa.

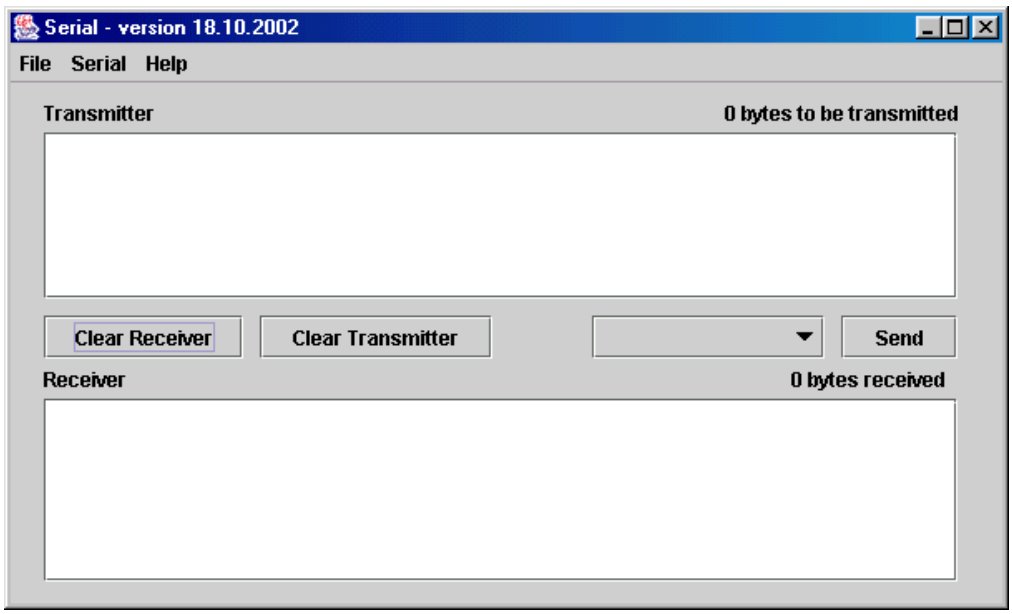


Figura 30 – Tela principal do programa.

Ao iniciar, o programa automaticamente carrega o arquivo *processor.cfg*. Este arquivo deve ser configurado pelo usuário para identificar quais são os módulos processadores presentes no sistema. A Figura 31 mostra um exemplo de arquivo de configuração que contêm dois processadores. O primeiro tem endereço 01 e é chamado de "Processador 1". O segundo tem endereço 04 e é chamado de "Processador 2". Após ler este arquivo, dois terminais janelas (Figura 32) são abertos para comunicação direta com o processador.

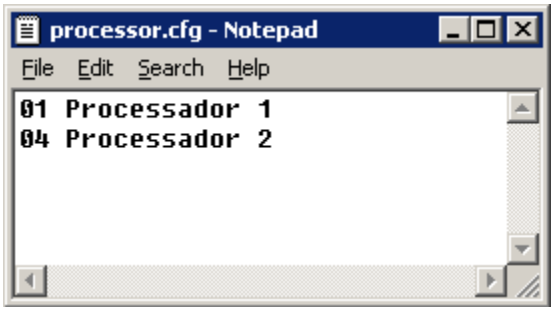


Figura 31 – Arquivo de configuração dos endereços dos processadores no sistema.

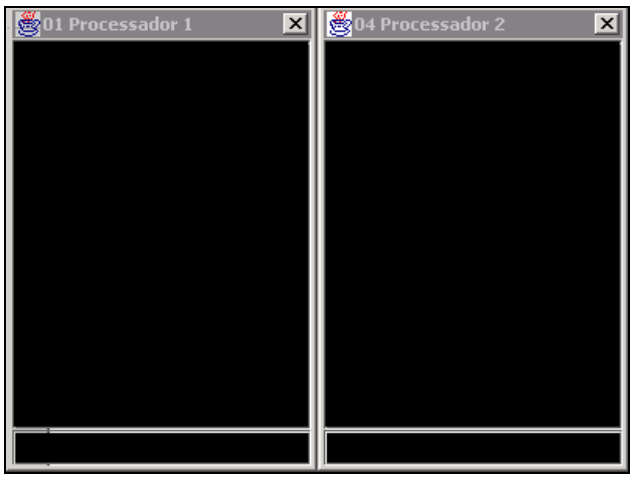


Figura 32 – Terminais de entrada/saída dos processadores do sistema.

Para conectar o software ao sistema é necessário primeiro informar as propriedades da conexão pelo menu *Serial – Configure*. Após isso, a conexão pode ser estabelecida pelo menu *Serial – Connect*. Caso nenhum erro tenha ocorrido, a conexão é estabelecida e os botões da janela principal do programa são habilitados. A partir deste momento qualquer informação pode ser enviada pela janela *transmitter* e qualquer informação recebida será automaticamente apresentada na janela *receiver*. Se algum dos processadores do sistema executar uma operação de Entrada/Saída, os dados também serão mostrados no terminal específico do processador.

4.2 Módulo Serial

A Seção 4.1 apresentou as características do software que envia e recebe dados da interface serial. Esta Seção mostra como os dados são recebidos e enviados pela plataforma de prototipação. A Figura 33 mostra os sinais deste módulo e como os mesmos fazem a interface com o mundo externo e o sistema.

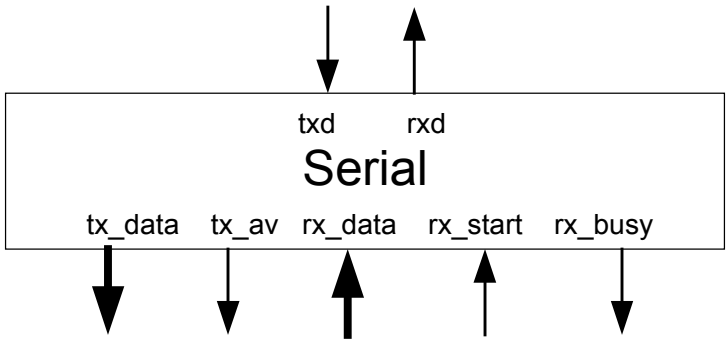


Figura 33 – Módulo Serial.

Os sinais dispostos na parte superior da Figura 33 fazem interface com o computador hospedeiro, que envia e recebe dados bit a bit. Os sinais dispostos na parte inferior da figura fazem interface com o sistema, que envia e recebe dados byte a byte. Abaixo segue uma breve explicação sobre cada um dos sinais vistos na Figura 33.

- *rxd*: envia dados bit a bit pela serial.
- *txd*: recebe dados bit a bit pela serial.
- *rx_data*: byte a ser convertido e transmitido para o computador hospedeiro.
- *rx_start*: indica que existe um byte disponível a ser transmitido em *rx_data*.
- *rx_busy*: indica se está sendo efetuado uma transmissão.
- *tx_data*: byte recebido do computador hospedeiro.
- *tx_av*: indica que existe um byte disponível em *tx_data*.

A primeira tarefa deste módulo é ajustar a velocidade do sistema interno à velocidade da porta serial. Esta sincronização é feita através do envio do byte com valor 55 (01010101 em binário) e deve ser realizada cada vez que a plataforma for inicializada. Quando este valor é recebido pelo módulo serial é efetuado um "*autobaud*", ou seja, uma contagem de quantos pulsos de relógio da plataforma de prototipação ocorrem enquanto o valor recebido pela serial está em nível lógico alto (valor binário 1). A seguir, a quantidade de ciclos de relógio contabilizados é dividido por 8 para determinar o semi-período. A partir deste momento, têm-se o ciclo de relógio no qual a serial opera.

Depois que este módulo é inicializado ele fica em laço recebendo dados da interface serial. Para isto ele detecta um start bit e armazena os 8 bits subseqüentes. Após isso, ele ativa o sinal *tx_av* indicando que há um byte recebido do computador hospedeiro disponível no sinal *tx_data*. Quando o sistema deseja enviar dados para o computador hospedeiro, deve-se colocar o byte a ser transmitido no sinal *rx_data* e ativar o sinal *rx_start*, avisando que existe um novo byte. A partir deste momento é enviado o start bit mais o byte a ser transmitido.

4.3 Núcleo Serial

O núcleo Serial encapsula o módulo Send/Receive, apresentado na Seção 3.3, e o módulo Serial, apresentado na Seção 4.2.

A função do núcleo Serial é receber os dados enviados pelo software Serial, através do módulo Serial, selecionar o comando e montar adequadamente o pacote, para que este seja enviado pelo módulo Send/Receive. Da mesma forma, receber os pacotes destinados ao núcleo Serial, selecionar o comando e enviar para o Software Serial.

A Figura 34 ilustra o núcleo Serial com seus módulos e respectivos sinais.

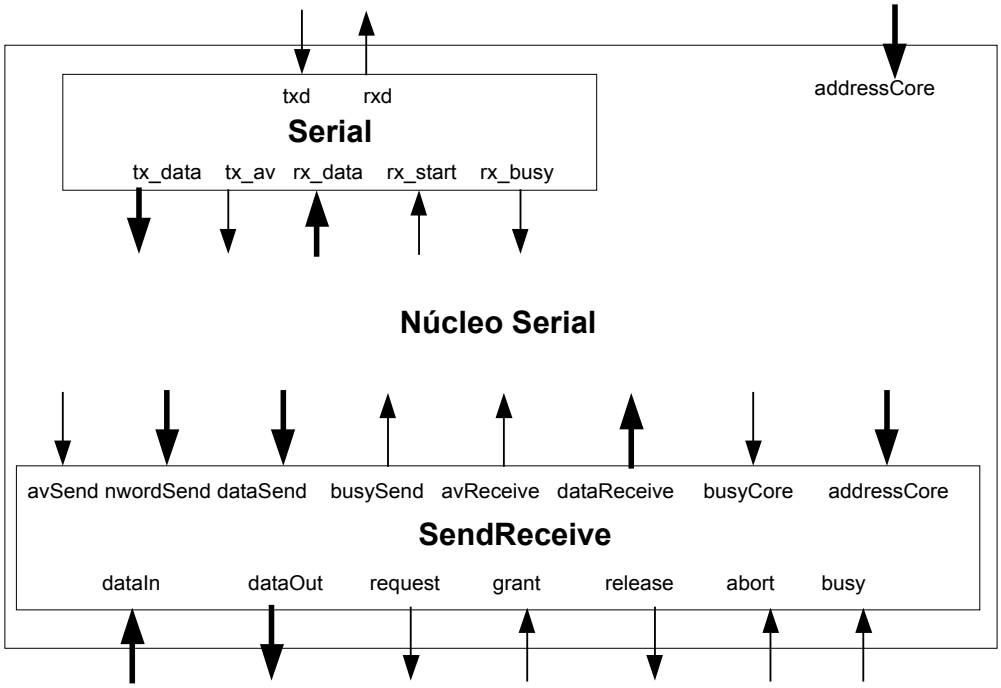


Figura 34 – Diagrama de blocos do núcleo Serial.

É importante destacar que os sinais que compõem a interface externa do núcleo Serial fazem parte das interfaces dos módulos Serial e Send/Receive, sendo *txd* e *rxd* sinais do módulo Serial e *request*, *grant*, *release*, *busy*, *abort*, *dataIn* e *dataOut* sinais do módulo Send/Receive. O sinal *addressCore* é o endereço do núcleo Serial que deve ser atribuído pelo sistema.

O sistema possui um conjunto de comandos que identificam qual o processo a ser executado e qual o formato do pacote a ser montado ou interpretado pelos núcleos. Estes comandos são apresentados na Tabela 1.

Tabela 1 – Comandos do sistema.

Identificador	Comando	Descrito na Seção:
0	Leitura da memória	4.3.1
1	Escrita na memória	4.3.2
2	Habilitação do processador	4.3.3
3	Saída	4.3.4
4	Entrada	4.3.4
5	Confirmação de recebimento	3.3
6	*Execution time*	6.2
7	*Wait*	6.2
8	*Notify*	6.2

O núcleo da Serial utiliza seis dos comandos do sistema: leitura da memória, escrita na memória, habilitação do processador, entrada, saída e *execution time* (tempo de execução do algoritmo do processador). A Figura 35 mostra como a máquina de estados Serial realiza a seleção dos estados que tratam cada comando.

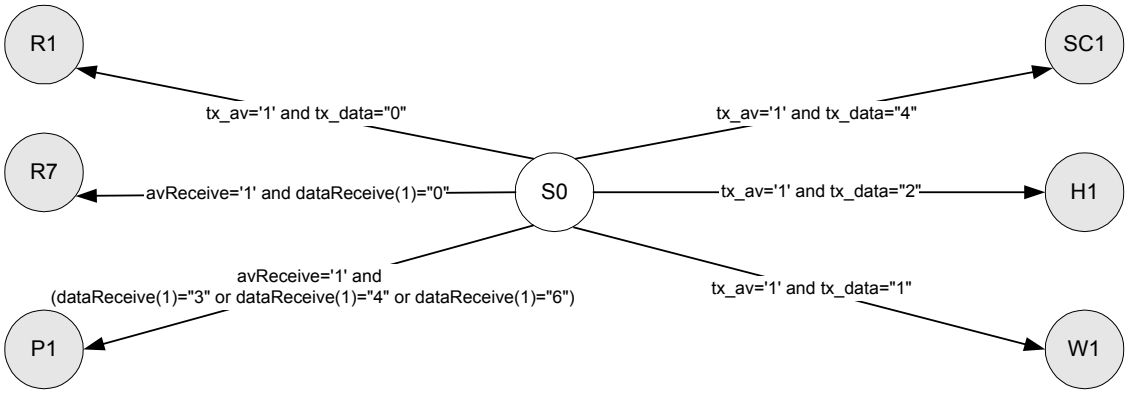


Figura 35 – Máquina de estados parcial do núcleo Serial.

O estado S0 é o estado de repouso da máquina de estados Serial. É neste estado que a máquina permanece até receber transmissões do computador hospedeiro ou pacotes do sistema.

Quando a máquina Serial encontra-se no estado S0 e percebe o sinal *tx_av* ativo, indicando que possui dado disponível oriundo do computador hospedeiro, o núcleo Serial verifica qual é o valor do sinal *tx_data*, pois este indicará o comando a ser executado. Caso *tx_data* seja igual a 0, o comando a ser executado é de leitura da memória e a máquina de estados avança para o estado R1. O estados R1 a R6 correspondem à transmissão do pedido de leitura da memória e os estados R7 a R13 correspondem à recepção de dado com o barramento pela máquina Serial. A recepção do dado lido do barramento é identificado quando o sinal *avReceive* está ativo, indicando que há um pacote do sistema disponível em *dataReceive* e a segunda palavra do mesmo é igual a 0. Todos os estados do comando de leitura encontram-se detalhado na Seção 4.3.1. Caso *tx_data* seja igual a 1, o comando a ser executado é o de escrita na memória e a máquina de estados deve avançar de S0 para W1. O funcionamento do comando de escrita na memória encontra-se detalhado na Seção 4.3.2. Caso *tx_data* seja igual a 2, o comando a ser executado é o de habilitação do processador e a

máquina Serial avança para H1. O comando de habilitação do processador encontra-se detalhado na Seção 4.3.3. A Seção 4.3.5 apresenta o detalhamento do comando de entrada recebido do computador hospedeiro (estados SC1 a SC6), que é identificado por *tx_data* igual a 4.

Quando a máquina Serial está no estado S0 e detecta *avReceive* ativo, indicando que há um pacote do sistema disponível em *dataReceive*, o núcleo Serial captura o pacote e verifica qual o valor da segunda palavra do mesmo, que indica o comando a ser executado. Caso o comando do pacote seja igual a 3, 4 ou 6, ou seja, o comando a ser executado seja o de saída, o de entrada ou o *execution time*, respectivamente, a máquina Serial avança para o estado P1. Os comandos de entrada, saída e *execution time* abrangem os estados P1 a P22. A Seção 4.3.4 apresenta o detalhamento destes comandos.

4.3.1 Comando de leitura da memória

O comando de leitura é enviado pelo computador hospedeiro para obter dados ou instruções armazenados em uma ou várias posições de memória do sistema. A parte da máquina de estados Serial que trata este comando é apresentada na Figura 36 e o seu funcionamento detalhado é descrito a seguir.

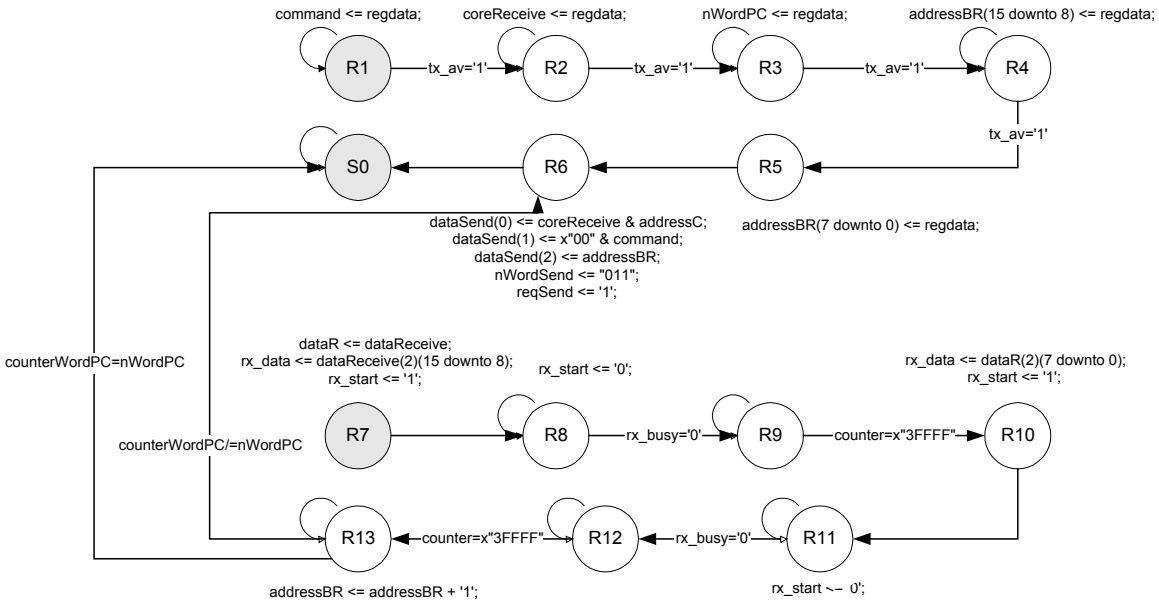


Figura 36 – Máquina de estados da leitura da memória.

Como relatado na Seção 4.3 o comando de leitura da memória é identificado quando a máquina Serial encontra-se no estado S0, o sinal *tx_av* é ativado e o sinal *tx_data* é igual a 0. A ocorrência destes três eventos faz com que a máquina de estados avance do estado S0 para R1.

1. No estado R1 o valor de *tx_data*, que corresponde ao comando, é armazenado e se espera até que o sinal *tx_av* seja ativado novamente para que se avance para o estado R2.
2. Em R2 o valor de *tx_data* é armazenado novamente. Neste estado, o valor do sinal *tx_data* corresponde ao endereço do núcleo, ou seja, o endereço do núcleo do sistema que identifica a memória em que se deseja realizar a leitura. Do estado R2 a máquina avança

para R3 quando o sinal *tx_av* é ativado.

3. No estado R3 é armazenado o valor de *tx_data* corresponde ao número de palavras que se deseja ler da memória. Da mesma forma que os estados anteriores, a máquina permanece em R3 até que *tx_av* seja ativado indicando que há outro dado disponível.
4. Nos estados R4 e R5 é armazenado o valor do sinal *tx_data* que corresponde à posição da memória de onde se deseja realizar a leitura da primeira palavra. No estado R4 são armazenados os 8 bits mais significativos da posição e no estado R5 os 8 bits menos significativos. Do estado R5 a máquina avança automaticamente para o estado R6.
5. O estado R6 é responsável por montar o pacote (apresentado na Figura 37) a ser enviado para o núcleo de memória informado e requisitar à máquina Send sua transmissão. Deste estado a máquina Serial avança automaticamente para o estado inicial, S0.
6. Após ter enviado o pacote com o pedido de leitura ao núcleo de memória, o núcleo Serial espera da mesma o pacote com o valor lido (apresentado na Figura 38). Logo, ao detectar o sinal *avReceive* ativo, que indica que o núcleo recebeu um pacote, é verificado se a segunda palavra do pacote é composta pelo comando de leitura da memória (0). Caso a verificação seja verdadeira a máquina avança para R7 (transição ilustrada na Figura 35).
7. No estado R7 é capturado o pacote disponível em *dataReceive* e os 8 bits mais significativos da terceira palavra, que indica o conteúdo da posição lida, são disponibilizados para o computador hospedeiro, através do armazenamento dos mesmos em *rx_data* e da ativação do sinal *rx_start*.
8. No estado R8 o módulo serial ativa o sinal *rx_busy*, indicando que o mesmo está capturando o dado em *rx_data*. Do estado R8, avança-se para R9 quando o sinal *rx_busy* for desativado.
9. A máquina Serial permanece no estado R9 enquanto o módulo serial transmite o dado lido para o computador hospedeiro. Observar que há uma espera de 3FFFFh ciclos de relógio para a transição do estado R9 para R10. Este valor foi determinado empiricamente, correspondendo ao tempo necessário para transmitir um byte para o computador hospedeiro e este processá-lo.
10. No estado R10 são disponibilizados para o computador hospedeiro, através do armazenamento dos mesmos em *rx_data* e da ativação do sinal *rx_start*, os 8 bits menos significativos da terceira palavra do pacote *dataReceive*, que indica o conteúdo da posição lida.
11. No estado R11, como no estado R8, o módulo serial ativa o sinal *rx_busy*, indicando que o mesmo está capturando o dado em *rx_data*. Do estado R11 avança-se para R12 quando o sinal *rx_busy* for desativado.
12. A máquina Serial permanece no estado R12 enquanto o módulo serial transmite o dado lido para o computador hospedeiro.
13. No estado R13 é verificado se o número de palavras que se deseja ler, informado no estado R3, é igual ao número de palavra lidas. Caso seja igual retorna-se para o estado de repouso (S0). Caso contrário, incrementa-se a atual posição de memória e retorna-se para o estado R6 para transmitir um novo pacote de pedido de leitura da memória.

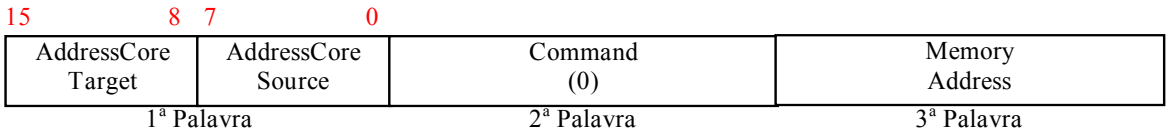


Figura 37 – Pacote de pedido de leitura da memória.

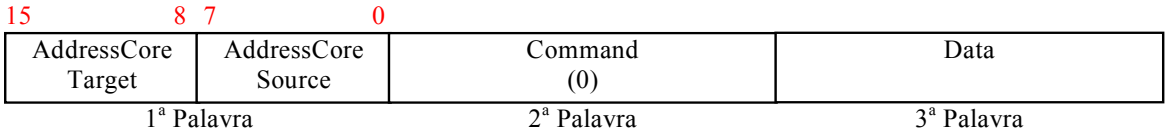


Figura 38 – Pacote que contém o dado lido da memória.

4.3.2 Comando de escrita na memória

O comando de escrita é enviado pelo computador hospedeiro para armazenar dados ou instruções em uma ou várias posições de memória do sistema. A parte da máquina de estados Serial que trata este comando é apresentada na Figura 39 e o seu funcionamento detalhado a seguir.

Como relatado na Seção 4.3 o comando de escrita na memória é identificado quando a máquina Serial encontra-se no estado S0, o sinal *tx_av* é ativado e o sinal *tx_data* é igual a 1. A ocorrência destes três eventos faz a máquina de estados avançar para W1.

1. No estado W1 o valor de *tx_data*, que corresponde ao comando, é armazenado e se espera até que o sinal *tx_av* seja ativado novamente para que se avance para o estado W2.
2. Em W2 o valor de *tx_data* é armazenado novamente. Neste estado o valor do sinal *tx_data* corresponde ao endereço do núcleo, ou seja, o endereço do núcleo do sistema que identifica a memória em que se deseja realizar a escrita. Do estado W2 a máquina avança para W3 quando o sinal *tx_av* é ativado.
3. No estado W3 é armazenado o valor de *tx_data* correspondente ao número de palavras que se deseja escrever na memória. O número de palavras (*nWordPC*) indica o número de palavras a enviar. Da mesma forma que os estados anteriores, a máquina permanece em W3 até que *tx_av* seja ativado indicando que há outro dado disponível.
4. Nos estados W4 e W5 é armazenado o valor do sinal *tx_data* que corresponde à posição da memória onde se deseja realizar a escrita da primeira palavra. No estado W4 é armazenado os 8 bits mais significativos do endereço e no estado W5 os 8 bits menos significativos, formando assim o endereçamento da memória que é de 16 bits. Do estado W4 a máquina avança para o estado W5 quando *tx_av* é ativado. Com a mesma condição avança do estado W5 para o estado W6.
5. Nos estados W6 e W7 é armazenado o valor do sinal *tx_data* que corresponde ao valor que se deseja escrever na posição de memória informada nos estados W4 e W5. No estado W6 são armazenados os 8 bits mais significativos do valor e no estado W7 os 8 bits menos significativos, formando assim a palavra de memória que é de 16 bits. Do estado W6 a máquina avança para o estado W7 quando *tx_av* é ativado. Do estado W7 a máquina Serial avança automaticamente para W8.

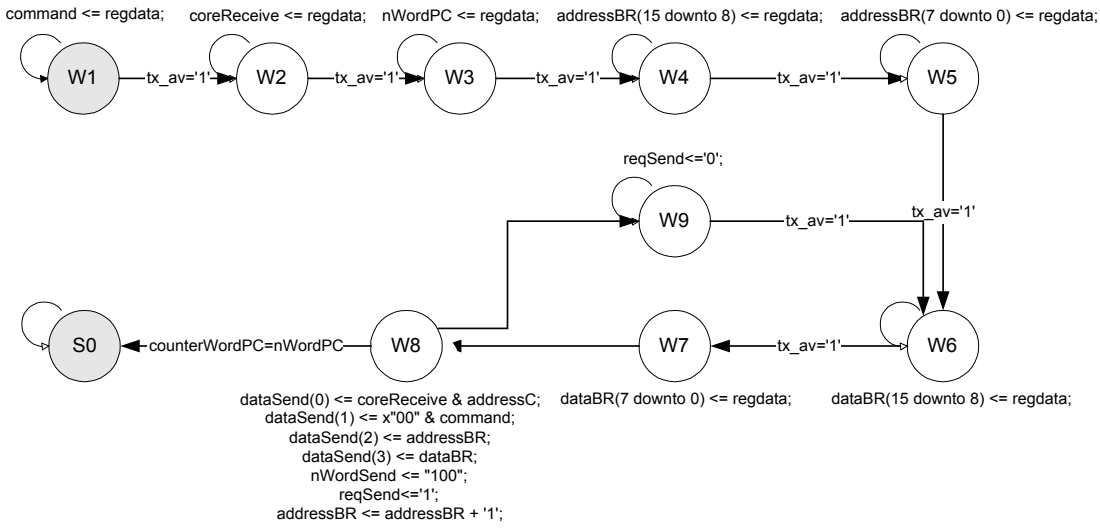


Figura 39 – Máquina de estados da escrita na memória.

6. O estado W8 é responsável por montar o pacote (apresentado na Figura 40) a ser enviado para o núcleo de memória informado e requisitar à máquina de Send a sua transmissão. Ainda no estado W8 é verificado se o número de palavras que se deseja escrever, informado no estado W3, é igual ao número de palavras escritas. Caso seja igual retorna-se para o estado de repouso (S0). Caso contrário, incrementa-se a atual posição de memória e avança-se para o estado W9.
7. No estado W9 é desativada a requisição de transmissão da máquina Send e retorna-se para o estado W6, quando *tx_av* é ativado, para que o próximo valor a ser escrito seja capturado e transmitido.

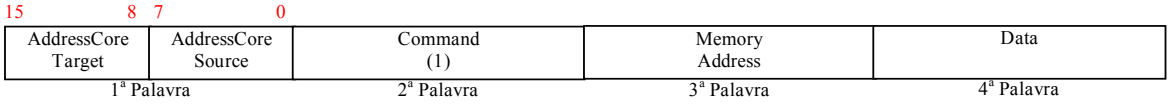


Figura 40 – Pacote de escrita na memória.

4.3.3 Comando de habilitação do processador

O comando de habilitação do processador é enviado pelo computador hospedeiro para permitir que o processador comece a executar instruções. A parte da máquina de estados Serial que trata este comando é apresentada na Figura 41 e o seu funcionamento é descrito a seguir.

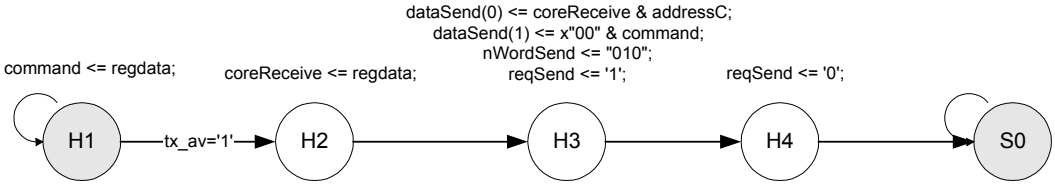


Figura 41 – Máquina de estados da habilitação do processador.

Como relatado na Seção 4.3 o comando de habilitação do processador é identificado quando a máquina Serial encontra-se no estado S0, o sinal *tx_av* é ativado e o sinal *tx_data* é igual a 2. A ocorrência destes três eventos faz com que a máquina de estados avance do estado S0 para H1.

1. No estado H1 o valor de *tx_data*, que corresponde ao comando, é armazenado e se espera até que o sinal *tx_av* seja ativado novamente para que se avance para o estado H2.
2. Em H2 o valor de *tx_data* é armazenado novamente. Neste estado o valor do sinal *tx_data* corresponde ao endereço do núcleo do sistema que identifica o processador que se deseja habilitar. Do estado H2 a máquina avança automaticamente para H3.
3. O estado H3 é responsável por montar o pacote (apresentado na Figura 42) a ser enviado para o núcleo Processador informado e requisitar a máquina de Send a sua transmissão. Do estado H3 avança-se automaticamente para H4.
4. No estado H4 é desativada a requisição de transmissão da máquina Send e retorna-se para o estado de repouso da máquina Serial (S0).

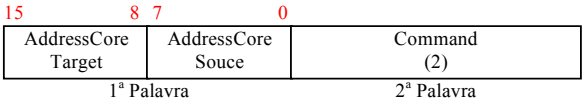


Figura 42 – Pacote de habilitação do processador.

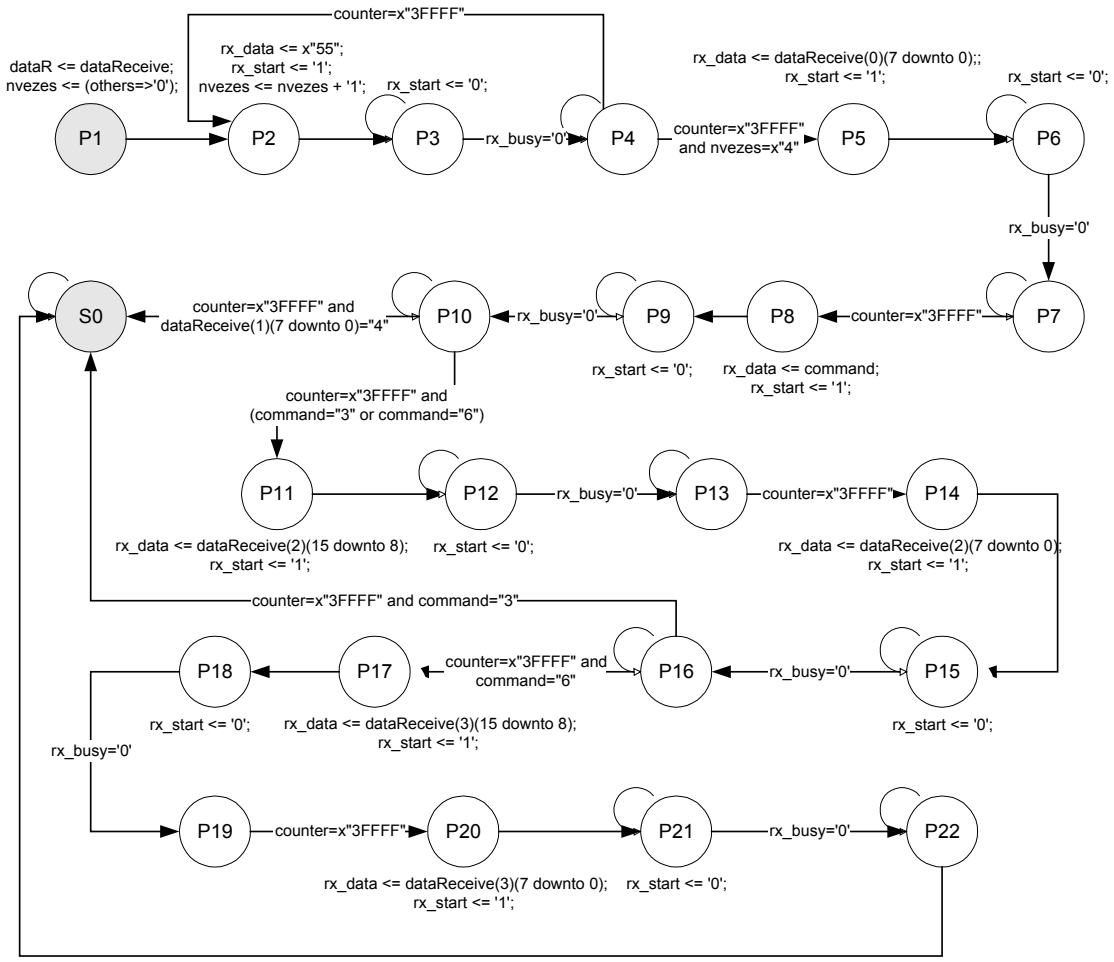
4.3.4 Comando de entrada, saída e *execution time* recebido do processador

Quando uma instrução de entrada, saída ou *halt* é executada pelo processador, o mesmo transmite um pacote com um comando associado para o núcleo Serial. O núcleo Serial captura este pacote, interpreta o comando e monta uma mensagem a ser transmitida ao computador hospedeiro. O software Serial recebe a mensagem e, caso seja o comando de saída ou o comando *execution time*, exibe-a no terminal do processador no hospedeiro, ou, caso contrário, ativa a requisição de entrada no terminal do processador.

Como relatado na Seção 4.3 os comandos de entrada, saída e *execution time* são identificados quando a máquina Serial encontra-se no estado S0, o sinal *avReceive* é ativado e a segunda palavra do pacote *dataReceive* é igual a 3, 4 ou 6. A ocorrência destes três eventos faz com que a máquina de estados apresentada na Figura 43 avance do estado S0 para P1.

1. No estado P1 é capturado o pacote *dataReceive* e decodificado o comando.
2. Os estados P2 a P4 são responsáveis por transmitir ao módulo Serial uma seqüência de quatro bytes 55H. Essa seqüência foi escolhida empiricamente para indicar que um comando, e não um dado, está sendo transmitido ao software Serial. Em P2 é disponibilizada a seqüência no sinal *rx_data*, através da ativação do sinal *rx_start*. Em P3 o módulo Serial ativa *rx_busy*, indicando que o módulo está capturando o dado. Do estado P3 avança-se para P4 quando o sinal *rx_busy* for desativado. Em P4 a máquina Serial espera o módulo Serial transmitir o dado e verifica se a seqüência completa foi transmitida. Se a seqüência foi completamente transmitida, a máquina Serial avança de P4 para P5, caso contrário, retorna a P2.

3. Do estado P5 a P7 são transmitidos ao módulo serial os 8 bits menos significativos da primeira palavra do pacote, que correspondem ao endereço do núcleo Processador originador do pacote. Da mesma forma que os estados P2 a P4, no estado P5 é disponibilizado o dado em *rx_data* e ativado o sinal *rx_start*, no estado P6 o módulo serial ativa *rx_busy* e após *rx_busy* ser desativado é esperada a transmissão do módulo Serial no estado P7.



3FFFFh: valor determinado empiricamente, correspondendo ao tempo necessário para transmitir um byte para o computador hospedeiro e este processá-lo.

Figura 43 – Máquina de estados dos comandos de entrada, saída e *execution time* recebidos do processador.

4. A seqüência de estados de P8 a P10 transmite ao módulo serial a segunda palavra do pacote, que corresponde ao comando. O procedimento nestes estados é igual ao dos estados P2 a P4 e P5 a P7. Ao final do estado P10, ou seja, quando o módulo Serial terminou a transmissão, verifica-se qual o comando está sendo executado. Se o comando em execução for o de entrada, a máquina Serial retorna a estado inicial (S0), porque a mensagem correspondente ao comando de entrada, apresentada na Figura 44, foi finalizada. Caso contrário, a máquina Serial avança do estado P10 para P11.

55H	55H	55H	55H	AdressCore Source	Command (4)
1º byte	2º byte	3º byte	4º byte	5º byte	6º byte

Figura 44 – Mensagem do comando de entrada (equivalente a um comando *scanf*).

5. Do estado P11 a P16 é transmitido ao módulo serial a terceira palavra do pacote. Nos estados P11 a P13 são transmitidos os 8 bits mais significativos e nos estados P14 a P16 são transmitidos os 8 bits menos significativos. No estado P16 é verificado qual o comando está sendo executado. Se o comando em execução for o de saída, a mensagem, apresentada na Figura 45, foi completamente enviada e a máquina Serial retorna ao estado inicial (S0). Caso o comando seja o *execution time*, que possui 32 bits de dados, a máquina Serial avança para o estado P17.

55H	55H	55H	55H	AdressCore Source	Command (3)	Data (15:8)	Data (7:0)
1º byte	2º byte	3º byte	4º byte	5º byte	6º byte	7º byte	8º byte

Figura 45 – Mensagem do comando saída.

6. Do estado P17 a P22 é transmitido ao módulo serial a quarta palavra do pacote. O procedimento destes estados é igual aos estados P11 a P16. No estado P22, quando o módulo Serial terminou a transmissão da mensagem (apresentada na Figura 46), a máquina Serial retorna a estado inicial (S0).

55H	55H	55H	55H	AdressCore Source
1ºbyte	2ºbyte	3ºbyte	4ºbyte	5ºbyte

Command (3)	Data (31:24)	Data (23:16)	Data (15:8)	Data (7:0)
6ºbyte	7ºbyte	8ºbyte	9ºbyte	10ºbyte

Figura 46 – Mensagem do comando *execution time*.

4.3.5 Comando de entrada recebido do computador hospedeiro

Quando uma instrução de entrada é executada pelo processador, o mesmo transmite um pacote com o comando para o núcleo Serial. O núcleo Serial captura este pacote, interpreta o comando e monta uma mensagem a ser transmitida ao computador hospedeiro (rever a Figura 44). O software Serial, do lado do hospedeiro, recebe a mensagem e ativa a requisição de entrada do terminal do processador originador do comando. Quando o usuário atende a esta requisição, o software Serial transmite ao núcleo Serial uma seqüência de bytes, seqüência esta que identifica o comando a ser tratado nesta Seção. A Figura 47 apresenta a parte da máquina de estados Serial que trata o comando de entrada recebido do computador hospedeiro e o funcionamento desta é descrito a seguir.

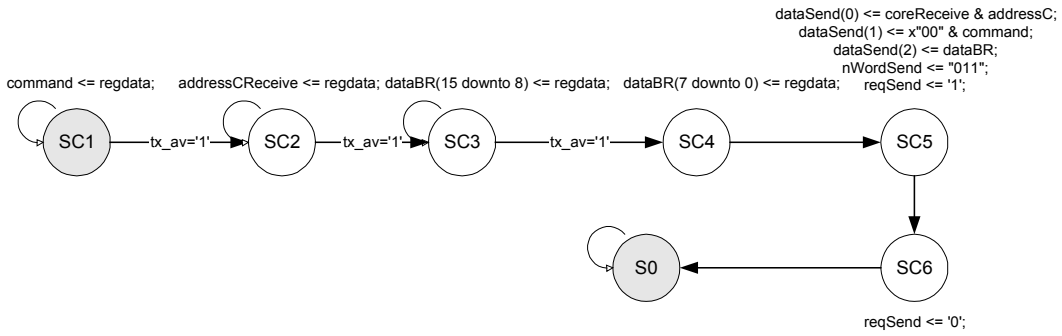


Figura 47 – Máquina de estados do comando de entrada recebido do computador hospedeiro.

Como relatado na Seção 4.3 o comando de entrada recebido do computador hospedeiro é identificado quando a máquina Serial encontra-se no estado S0, o sinal *tx_av* é ativado e o sinal *tx_data* é igual a 4. A ocorrência destes três eventos faz com que a máquina de estados avance do estado S0 para SC1.

1. No estado SC1 o valor de *tx_data*, que corresponde ao comando, é armazenado e se espera até que o sinal *tx_av* seja ativado novamente para que se avance para o estado SC2.
2. Em SC2 o valor de *tx_data* é armazenado novamente. Neste estado o valor do sinal *tx_data* corresponde ao endereço do núcleo, ou seja, o endereço do núcleo do sistema que identifica o processador que requisitou a entrada. Do estado SC2 a máquina avança para SC3 quando o sinal *tx_av* é ativado.
3. Nos estados SC3 e SC4 é armazenado o valor do sinal *tx_data* que corresponde ao valor fornecido pelo usuário conforme requisitado. No estado SC3 são armazenados os 8 bits mais significativos do valor e no estado SC4 os 8 bits menos significativos, formando assim o valor que é de 16 bits. Do estado SC3 a máquina avança para o estado SC4 quando *tx_av* é ativado. Do estado SC4 a máquina de estados Serial avança automaticamente para o estado SC5.
4. O estado SC5 é responsável por montar o pacote (apresentado na Figura 48) a ser enviado para o núcleo Processador, informado no estado SC2, e requisitar a máquina de Send a sua transmissão. Deste estado a máquina Serial avança automaticamente para o estado SC6.
5. No estado SC6 é desativada a requisição de transmissão e automaticamente retorna-se ao estado inicial da máquina Serial (S0).

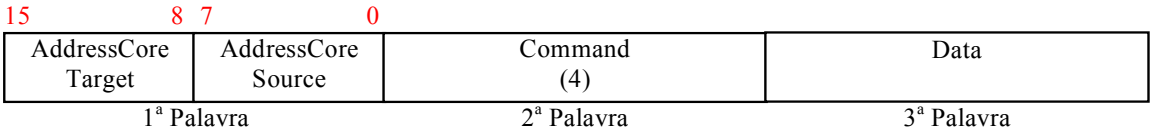


Figura 48 – Pacote que contém o dado do comando de entrada.

5 Sistema de Memória

As memórias do sistema provêem armazenamento de dados ou instruções e podem ser acessadas tanto pelo barramento local processador-memória quanto pelo barramento do sistema, conforme mencionado no Capítulo 2. As instruções e os dados do núcleo Memória são acessados através do barramento processador-memória quando este corresponder à memória local do processador. Quando o núcleo Memória corresponder a uma memória remota, as instruções e os dados são acessados através do barramento do sistema. O núcleo Serial também realiza acessos ao núcleo Memória através do barramento do sistema para escrever ou ler dados e/ou instruções.

A Seção 5.1 apresenta o funcionamento do módulo de memória disponível no FPGA utilizado para o desenvolvimento deste trabalho. A Seção 5.2 apresenta o conjunto de módulos Memória que compõem o núcleo Memória, como estes são acessados e como são tratados os comandos de leitura e escrita neste núcleo.

5.1 Módulo Memória

O bloco básico de memória nos FPGAs da família Virtex II é denominado blockRAM, composto por 18k bits. O dispositivo XC2V1000 utilizado neste trabalho possui 40 blocos de memória, totalizando 720k bits de memória síncrona on-chip para lógica do usuário. As blockRAMs podem ser configuradas como memórias de porta simples ou de porta dupla e possuem 1 bit para cada byte da memória para armazenamento de paridade ou correção de erro. Esta memória pode ter a sua estrutura de acesso configurada conforme a Tabela abaixo:

Tabela 2 – Formatos de endereçamento da blockRAM .
A linha em negrito apresenta a configuração utilizada neste trabalho.

Largura	Profundidade	Addr	Data
1	16384	Addr<13:0>	Data<0>
2	8192	Addr<12:0>	Data<1:0>
4	4096	Addr<11:0>	Data<3:0>
9	**2048**	**Addr<10:0>**	**Data<7:0>**
18	1024	Addr<9:0>	Data<15:0>
36	512	Addr<8:0>	Data<31:0>

A Figura 49 apresenta uma blockRAM porta simples com seus respectivos sinais descritos abaixo.

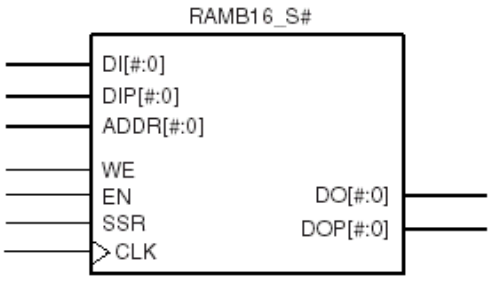


Figura 49 – Blockram porta simples.

- *clk*: relógio da memória.
- *ssr*: quando ativo a saída da memória é zerada.
- e*n*: quando ativo habilita a blockram para leitura ou escrita.
- *we*: quando ativo habilita a escrita na blockram.
- *addr*: informa o endereço de leitura ou escrita.
- *di*: informa o valor a ser escrito no endereço *addr* em uma operação de escrita.
- *dip*: informa o valor do bit de paridade a ser escrito no endereço *addr*.
- *do*: informa o valor lido do endereço *addr* em uma operação de leitura.
- *dop*: informa o valor do bit de paridade do endereço *addr*.

A blockRAM utilizada no projeto é porta simples (Figura 49) e possui endereçamento de 11 bits com palavra de 8 bits.

5.2 Núcleo Memória

O núcleo Memória encapsula o módulo Send/Receive, apresentado na Seção 3.3, e duas blockRAMs, apresentados na Seção 5.1. Os dois módulos de memória são organizados como mostra a Figura 50.

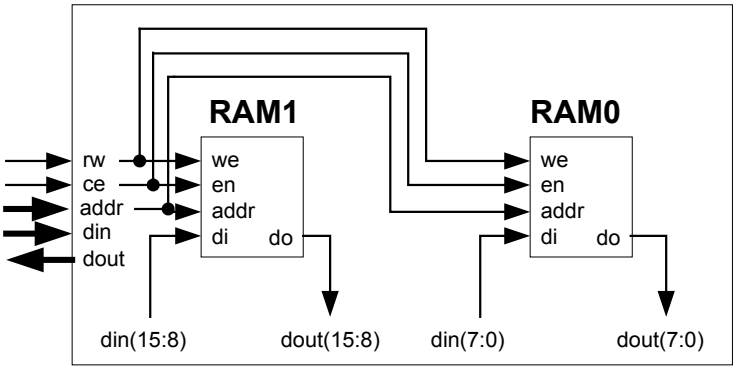


Figura 50 – Organização das blockRAMs no núcleo Memória.

Esta organização de memória foi adotada para prover ao processador um endereçamento local de 2048 palavras. Desta forma quando um dado (16 bits) for escrito na memória, a posição de memória informada endereça os dois módulos e o dado a ser escrito é particionado, sendo que os bits 15 a 8 são armazenados na blockram1 e os bits 7 a 0 na blockram0. Quando se deseja ler um dado da memória, a posição de memória, como no comando de escrita, endereça os dois módulos e o dado lido é constituído pela concatenação do valor do sinal *do* de cada módulo de memória.

Uma das funções do núcleo Memória é determinar qual dos barramentos, processador-memória ou barramento do sistema, terá acesso à memória. O barramento processador-memória possui privilégio de acesso. Esse privilégio é dado para que o acesso realizado pelo processador à sua memória local seja o mais veloz possível.

A Figura 51 ilustra o núcleo Memória com seus módulos e respectivos sinais, que serão descritos a seguir.

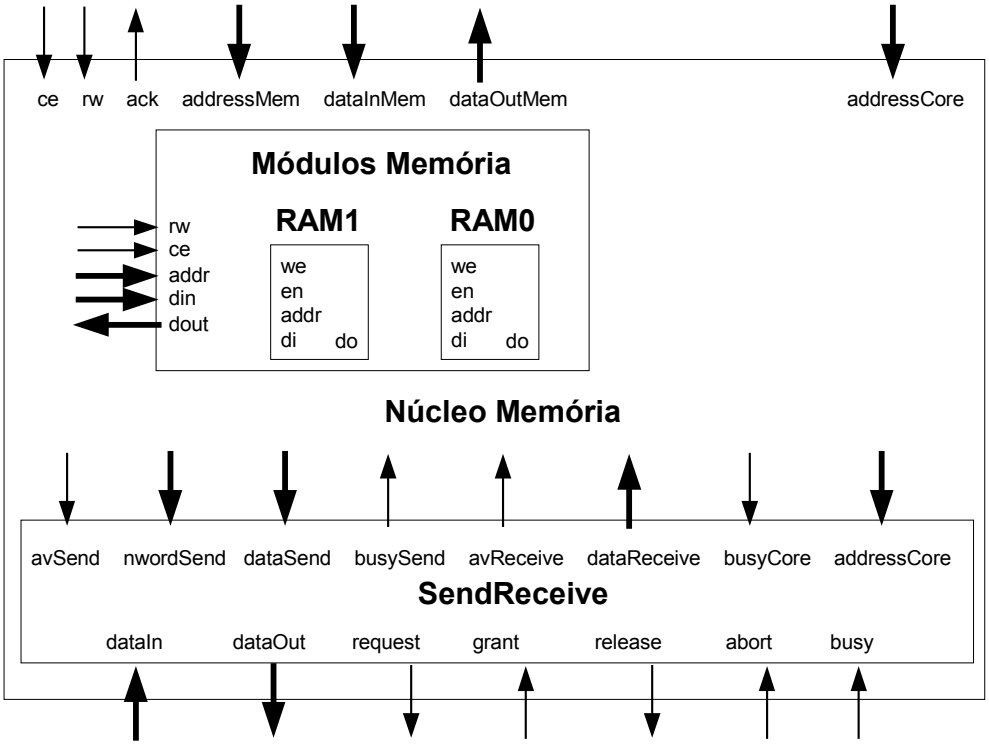


Figura 51 – Núcleo Memória.

- *clock*: relógio do sistema (não indicado na figura).
- *reset*: quando ativo inicializa as posições de memória com zeros (não indicado na figura).
- *addressCore*: endereço do núcleo no sistema.
- **Sinais do barramento:** *request*, *grant*, *release*, *abort*, *busy*, *dataIn* e *dataOut*.
- *ce*: sinal oriundo do processador para habilitar a memória.
- *rw*: sinal oriundo do processador que habilita uma escrita na memória.
- *ack*: sinal ativado quando uma operação de leitura ou escrita foi finalizada.
- *addressMem*: sinal oriundo do processador para endereçar a memória.
- *dataInMem*: sinal de entrada de dados/instruções do barramento processador-memória de 16 bits.
- *dataOutMem*: sinal de saída de dados/instruções do barramento processador-memória de 16 bits.

Devido ao privilégio que o barramento processador-memória possui, quando o sinal *ce* é ativado e a memória não estiver sendo acessada (sinal *accessBR*=0), o acesso é imediatamente liberado ao processador e *accessBR* passa a 1, indicando que o processador está utilizando a memória.

O acesso através do barramento do sistema é apresentado no diagrama de estados do Núcleo Memória (Figura 52) e o seu funcionamento detalhado é descrito a seguir.

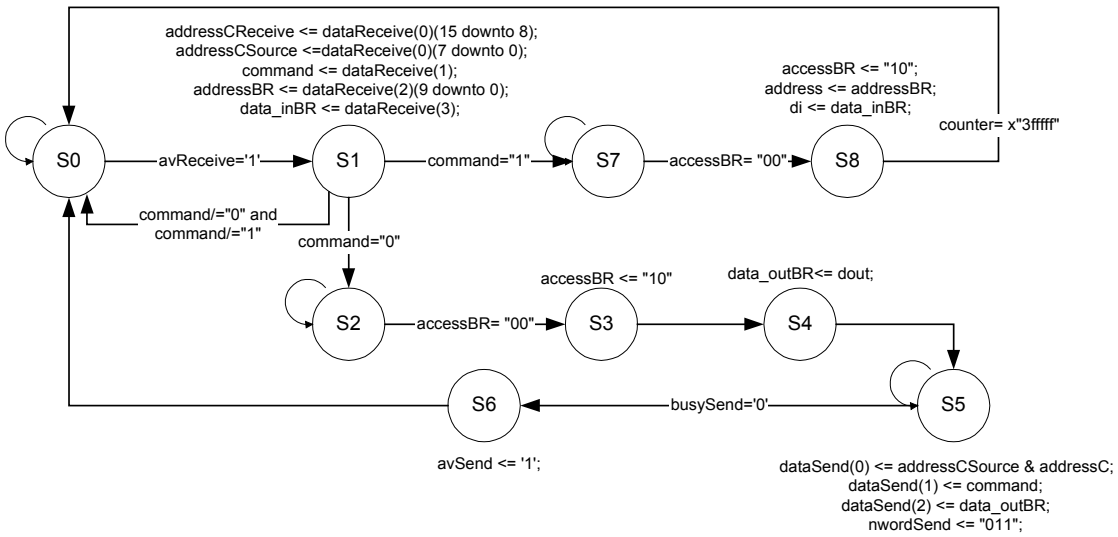


Figura 52 – Máquina de estados do núcleo Memória.

1. O estado S0 é o estado de repouso da máquina de estados do núcleo Memória. A máquina avança para S1 somente quando o sinal *avReceive* é ativado, indicando que existe um pacote disponível no sinal *dataReceive*.
2. No estado S1 o pacote disponível em *dataReceive* é capturado e é verificado o comando do pacote. O comando do pacote é identificado através da segunda palavra do mesmo. Se a segunda palavra do pacote for igual a 0, indicando comando de leitura da memória, a máquina avança para S2. Se for igual a 1, indicando comando de escrita na memória, a máquina avança para S7. Caso contrário, o comando é inválido e a máquina retorna ao estado de repouso (S0).
3. Os estados S2 a S6 compreendem os estados que tratam o comando de leitura da memória. O estado S2 é um estado de espera, onde a máquina permanece até que não exista acesso à memória pelo processador. No estado S3 o acesso à memória é liberado ao barramento do sistema e o endereçamento dos módulos de memória é realizado pelos 10 bits menos significativos da terceira palavra do pacote recebido. No estado S4 o sinal *dout* recebe o dado lido. No estado S5 é montado o pacote, conforme a Figura 38, e é requisitado à máquina Send o envio do mesmo. A máquina de estados Memória permanece em S5 enquanto a máquina Send estiver ocupada com outra transmissão. No estado S6 é transmitido o pacote à máquina Send e retorna-se ao estado S0.
4. Os estados S7 e S8 compreendem os estados que tratam o comando de escrita na memória. O estado S7 é um estado de espera, onde a máquina permanece até que não exista acesso à memória pelo processador. No estado S8 o acesso à memória é liberado ao barramento do sistema, o endereçamento dos módulos de memória é realizado pelos 10 bits menos significativos da terceira palavra do pacote recebido e o sinal *di* dos módulos Memória recebem a quarta palavra do pacote. Do estado S8 a máquina de estados Memória retorna automaticamente para o estado de repouso (S0).

6 Processador

Este Capítulo apresenta o processador escolhido para o desenvolvimento do projeto, o detalhamento das instruções de leitura e escrita em memória local e remota, o detalhamento das instruções de entrada e saída, o detalhamento da sincronização dos processadores que executam em paralelo e as interações do processador com o barramento do sistema e com o barramento processador-memória.

A arquitetura do processador R8, suas principais características e os motivos que levaram à escolha do mesmo são apresentados na Seção 6.1. A Seção 6.2 apresenta o núcleo do processador responsável pelo controle das instruções de leitura, escrita, entrada e saída, pela sincronização entre os processadores e pelas interações do processador com os barramentos.

6.1 Processador R8

O processador R8 foi desenvolvido pelos professores Fernando Gehm Moraes e Ney Laert Calazans [CAL02] [GAP02], sendo pela primeira vez prototipado neste trabalho. Ele é a oitava versão da família de processadores *Rx* estudado pelos alunos das disciplinas de Organização de Computadores e Arquitetura de Computadores dos cursos de Ciência da Computação e Engenharia da Computação da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

O processador R8 é uma organização Von Neumann (memória de dados/instruções unificada), load/store, com CPI entre 3 e 4, barramento de dados e endereços de 16 bits. Esta arquitetura é praticamente uma máquina RISC, faltando, contudo, algumas características gerais de máquina RISC, tal como *pipeline* e módulos de entrada/saída, como tratamento de interrupções.

Processadores embarcados comerciais simples, tais como o NIOS da Altera [ALT03] e Microblaze da Xilinx [XIL02], possuem organização semelhante ao do processador R8, diferenciando-se principalmente por já disporem de estruturas de entrada/saída, tais como tratamento de interrupções e *timers*. Uma vantagem daqueles em relação à R8 é a possibilidade de dimensionar o soft core para dados e endereços de 16 ou 32 bits.

O projeto da arquitetura, desenvolvido em VHDL, compreende a descrição de dois blocos principais: o bloco de controle e o bloco de dados. O bloco de controle tem por função gerar os comandos para a busca das instruções e envio de comandos ao bloco de dados para que a instrução seja executada. O bloco de dados contém 16 registradores de uso geral; registradores para armazenamento da instrução corrente (*IR*), endereço da próxima instrução a executar (*PC*) e ponteiro de pilha (*SP*); uma ULA (Unidade Lógica e Aritmética) com 13 operações e 4 qualificadores de estado. A Figura 53 ilustra a organização dos blocos do processador, assim como a interface com a memória externa.

Figura 53 – Diagrama de blocos do processador R8.

- *ck*: sincroniza os sinais internos ao processador.
- *rst*: determina a execução de instruções a partir do endereço zero da memória.
- *ce*: sinal que habilita operações de acesso a memória.
- *rw*: sinal que habilita a operação de escrita na memória.
- *address*: sinal que endereça a memória.
- *dataIn*: sinal de entrada de dados/instruções (16 bits) oriundo da memória ou do barramento.
- *dataOut*: sinal de saída de dados/instruções (16 bits) destinado à memória ou ao barramento.
- *haltR8*: sinal que quando ativo indica o fim do processamento.
- *waitR8*: sinal que quando ativo provoca uma pausa na execução do processador.

O sinal *waitR8* foi inserido no soft core original do processador para prover flexibilidade às aplicações de multiprocessamento. A facilidade na alteração do processador R8, tal como sua simplicidade e baixo consumo de área do FPGA justificam a escolha do mesmo.

6.2 Núcleo Processador

O núcleo Processador encapsula o módulo Send/Receive, apresentado na Seção 3.3, e o processador R8, apresentado na Seção 6.1. O núcleo Processador com seus módulos e seus respectivos sinais é ilustrado na Figura 54, sendo os sinais:

- *addressCore*: endereço do núcleo no sistema.
- *addressSerial*: endereço do núcleo Serial no sistema.
- **Sinais do barramento:** *request*, *grant*, *release*, *busy*, *abort*, *dataIn* e *dataOut*.
- *ce*: sinal que habilita operações de acesso a memória local.
- *rw*: sinal que habilita a operação de escrita na memória local.
- *ack*: sinal oriundo da memória local que informa quando uma operação de leitura ou escrita foi finalizada.
- *addressMem*: sinal que endereça a memória local.
- *dataInMem*: sinal de entrada de dados/instruções do barramento processador-memória de 16 bits.
- *dataOutMem*: sinal de saída de dados/instruções do barramento processador-memória de 16 bits.

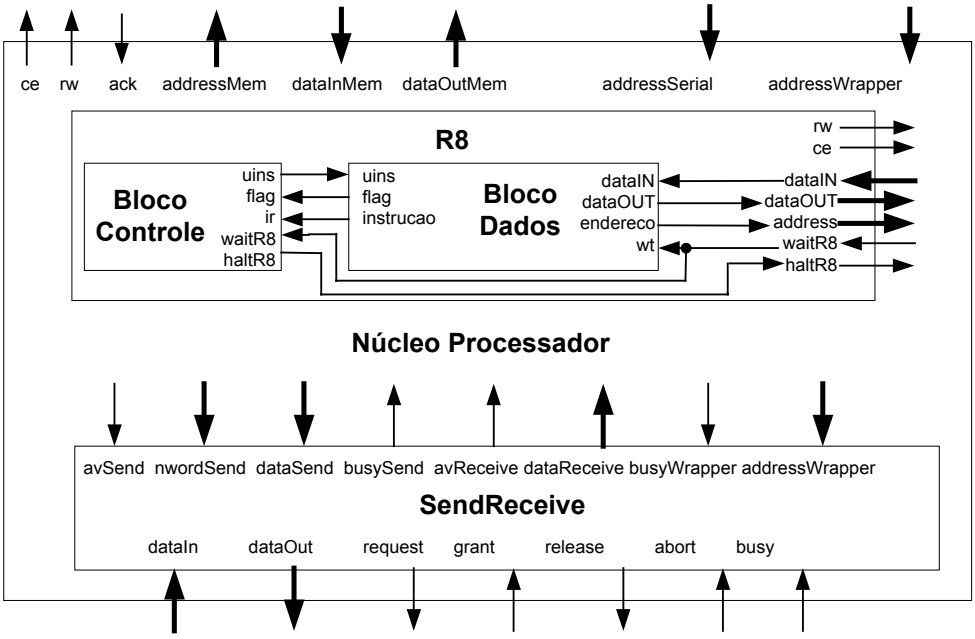


Figura 54 – Diagrama de blocos do núcleo Processador.

Uma das funções do núcleo Processador é controlar a execução do processador R8, provocando pausas toda a vez que o mesmo executar instruções de leitura, escrita, de entrada ou de saída. Quando o processador executa as instruções de leitura ou escrita em posições de memória pertencentes à memória local, o núcleo provoca uma pausa até que o sinal *ack* seja ativado pela memória, indicando que a operação foi executada com sucesso. Quando o processador executa a instrução de escrita destinada a uma posição da memória remota ou a instrução de saída, o núcleo provoca uma pausa somente até montar e transmitir o pacote com o comando para máquina Send (o pacote do comando de escrita pode ser visualizado na Figura 40 e o do comando de saída na Figura 55).

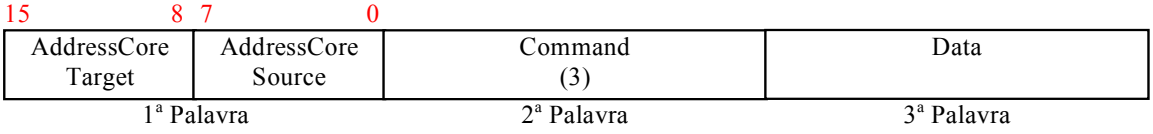


Figura 55 – Pacote do comando de saída.

Nas instruções de entrada e de leitura de uma posição da memória remota a pausa provocada pelo núcleo Processador se estende até que seja recebido o pacote com o dado lido. Na instrução de entrada esta pausa é maior porque é necessário que o pacote do comando de entrada, apresentado na Figura 56, seja transmitido ao núcleo Serial, o núcleo Serial transmita uma mensagem de entrada (Figura 44) ao software Serial, o usuário informe o dado e o caminho de volta se concretize com a recepção do pacote com o dado pelo núcleo Processador (Figura 48).

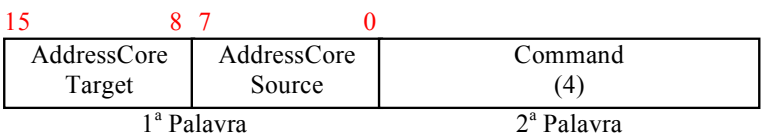


Figura 56 – Pacote do pedido de entrada.

Os comandos de leitura e de entrada são identificados pela instrução LD (*load*) do processador R8 e os comandos de escrita e de saída são identificados pela instrução ST (*store*). É função do núcleo Processador determinar qual destes comandos deve ser executado. Para isto foi estabelecido que os comandos de entrada e saída seriam mapeados na posição FFFFh da memória, ou seja, quando a instrução LD ou ST do processador R8 endereçarem a posição de memória FFFFh, a operação a ser executa é a de entrada ou saída, respectivamente. Outra função atribuída ao núcleo Processador refere-se ao acesso das instruções LD e ST à memória local ou à memória remota. Para determinar qual das memórias do sistema o processador R8 deve acessar foram estabelecidos intervalos de endereçamento referente a cada memória do sistema. Um possível conjunto de intervalos de endereçamento para um sistema com quatro memórias de 2048 posições é apresentado na Figura 57.

```
01 if(address>=0 && address<2047 ){
02         globalAddress=address;
03         addressCore=2;
04 }
05 else if(address>=2048 && address<4095){
06         globalAddress = 2048 - address;
07         addressCore=3;
08 }
09 else if(address>=4096 && address<6143){
10         globalAddress = 4096 - address;
11         addressCore=4;
12 }
13 else if(address>=6144 && address<8191){
14         globalAddress = 6144 - address;
15         addressCore=1;
16 }
```

Figura 57 – Exemplo de código fonte em linguagem C para determinar o endereço global.

Em um sistema multiprocesssado é necessário realizar a sincronização entre os processadores que executam em paralelo. Ao ser escolhida a técnica de troca de mensagens, como relatado na Seção 2.3, foi atribuído uma nova função ao núcleo Processador: o controle destas mensagens de sincronização. Para executar este controle foram incluídos dois novos comandos ao núcleo Processador: o comando *wait* e o comando *notify*. O comando *wait* é responsável por bloquear o processamento até a recepção do comando *notify*. Os comandos de *wait* e *notify* são identificados pela instrução ST do processador R8 e por este motivo, tiveram de ser mapeados em memória como os comandos de entrada e saída. O comando de *wait* foi mapeado na posição FFFEh da memória e o comando *notify* na posição FFFDh da memória. Assim quando o processador R8 executar a instrução ST R3, R1, R2, onde os valores dos registradores R3, R1 e R2 são 2h, 0h e FFFEh, respectivamente, o comando a ser executado será o de *wait*. Com estes valores o núcleo Processador provoca uma pausa no processador R8 até que o mesmo receba um pacote contendo o comando *notify* do núcleo do sistema com endereço 2. Da mesma forma, quando o processador R8 executar a instrução ST R3, R1, R2, e os registradores R3, R1 e R2 contiverem 2h, 0h e FFFDh, respectivamente, será executado o comando *notify* fazendo com que o núcleo Processador monte o pacote do comando *notify* (Figura 58) tendo como destino o núcleo do sistema com endereço 2.

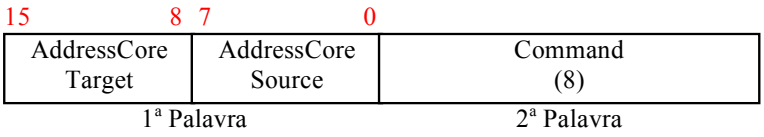


Figura 58 – Pacote do comando *notify*.

A forma como o núcleo Processador trata os comandos de leitura e escrita a memória local e remota, os comandos de entrada e saída e os comandos *wait* e *notify* é parcialmente apresentada na máquina de estados parcialmente mostrada na Figura 59 e o funcionamento detalhado da mesma, é descrito a seguir.

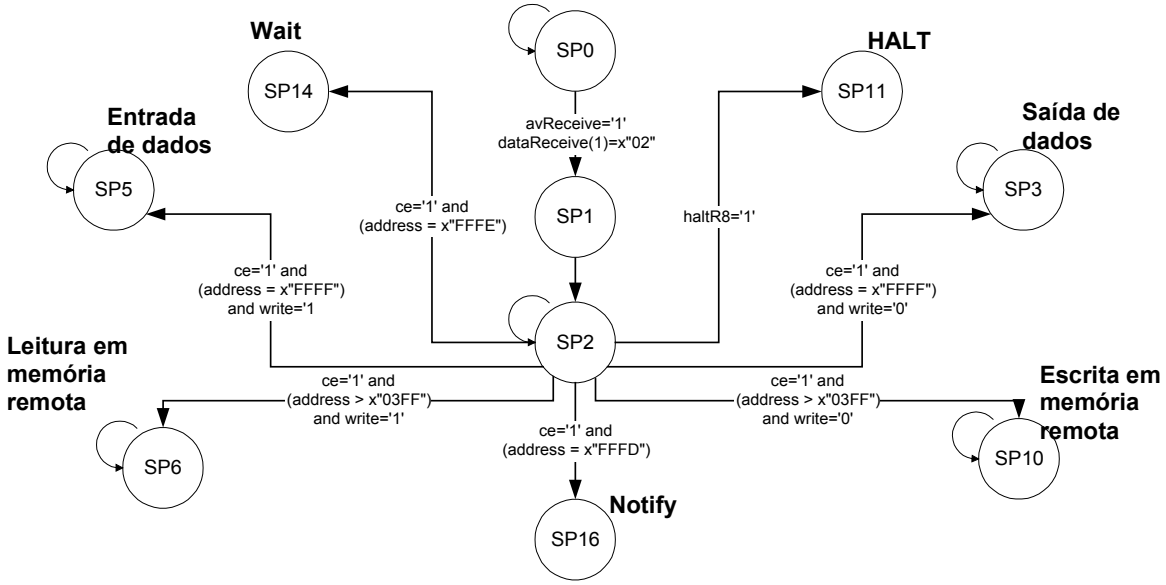


Figura 59 – Máquina de estados parcial do núcleo Processador.

- O estado SP0 é o estado de repouso. Neste estado o processador R8 encontra-se bloqueado. A máquina de estados somente avança do estado SP0 para o estado SP1 quando o núcleo Processador receber o pacote de habilitação (Figura 42).
- No estado SP1 é retirado o bloqueio do processador R8 e a máquina avança automaticamente para o estado SP2.
- No estado SP2 é verificado se o processador R8 ativou o sinal *haltR8*, que indica que o mesmo finalizou sua execução. Quando este sinal estiver ativo a máquina avança para o estado SP11. Caso contrário, é verificado se o sinal *ce* (acesso à memória) está ativo. Se o sinal *address* desta instrução for igual a:
  - FFFEh indica o comando *wait*, e a máquina avança para SP14;
  - FFFDh indica o comando *notify*, e a máquina avança para SP16;
  - FFFFh e *rw*=1 indica comando de saída, e a máquina avança para SP3;
  - FFFFh e *rw*=0 indica comando de entrada, e a máquina avança para SP5.

  Caso o valor do sinal *address* não corresponda a nenhum dos valores acima citados, é necessário selecionar a qual núcleo Memória este corresponde. Se o valor do sinal *address* endereçar alguma das memórias remotas e o sinal *rw* estiver ativo, a máquina avança para o estado SP6, caso contrário a máquina avança para o estado SP10. É importante salientar que qualquer uma das possibilidades citadas acima provoca o

bloqueio do processador, mas se nenhuma destas possibilidades for selecionada, o processador continua seu processamento e a máquina de estados permanece em SP2 .

- O estado SP3 (Figura 60) é alcançado quando o comando de saída é executado, o estado SP10 é alcançado quando o comando de escrita em uma posição da memória remota é executado e o estado SP16 é alcançado quando o comando *notify* é executado. Nestes estados é montado o pacote referente ao comando e é esperado que o módulo Send possa receber a requisição de envio do mesmo. Quando está condição é satisfeita a máquina avança para o estado SP4 onde a requisição de envio é ativada. Do estado SP4 a máquina avança automaticamente para o estado SP1, finalizando o comando de escrita à memória remota, o comando de saída ou o comando *notify* e prosseguindo o processamento.

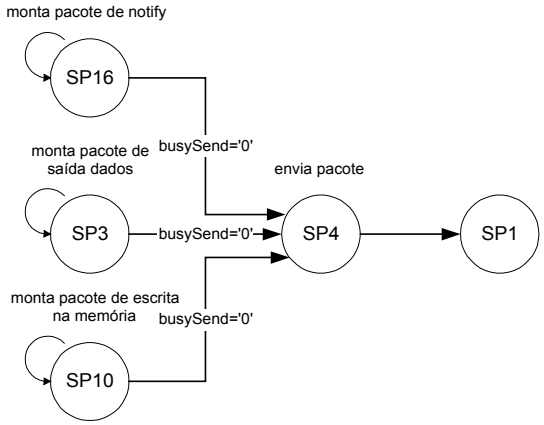


Figura 60 – Máquina de estados parcial do núcleo Processador.

- O estado SP5 (Figura 61) é alcançado quando o comando de entrada é executado e o estado SP6 é alcançado quando o comando de leitura a uma posição da memória remota é executado. Nestes estados é montado o pacote referente ao comando e é esperado que o módulo Send possa receber a requisição de envio do mesmo. Quando está condição é satisfeita a máquina avança para o estado SP7 onde é ativada esta requisição. Do estado SP7 a máquina avança para o estado SP8. No estado SP8 aguarda-se a chegada de um pacote contendo o dado lido do comando de leitura ou o do comando de entrada. Ao receber um pacote com esta característica, a máquina avança para o estado SP9. No estado SP9 é enviado ao processador R8 a terceira palavra do pacote recebido, que corresponde ao dado lido e avança-se para o estado de SP1, finalizando o comando de leitura à memória remota ou o comando de entrada e dando continuidade ao processamento.

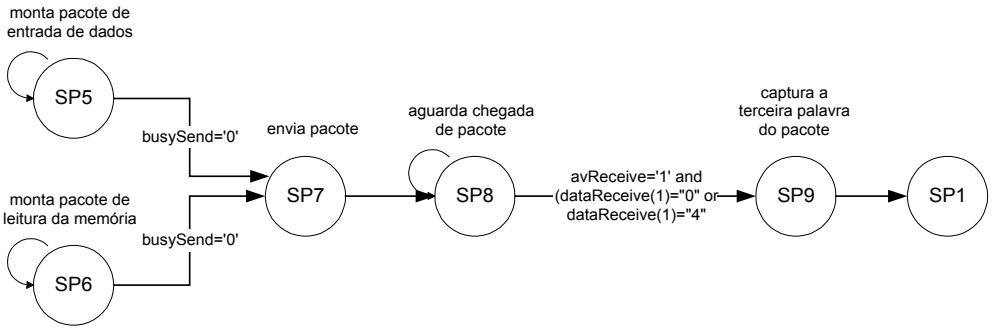


Figura 61 – Máquina de estados parcial do núcleo Processador.

- O estado SP11 (Figura 62) é alcançado quando o processador R8 ativa o sinal *haltR8*, indicando que finalizou sua execução. Neste estado é armazenado o número de ciclos de relógio gastos do momento em que o processador foi habilitado até a ativação do sinal *haltR8*. Do estado SP11 avança-se para o estado SP12 onde é montado o pacote do comando *execution time* (Figura 63) e é esperado que o módulo Send possa receber a requisição de envio. Quando esta condição é satisfeita, a máquina avança para o estado SP13. No estado SP13 é ativada a requisição de envio de um pacote do núcleo ao módulo Send e avança-se para o estado SP0, finalizando o comando *execution time* e o processamento.

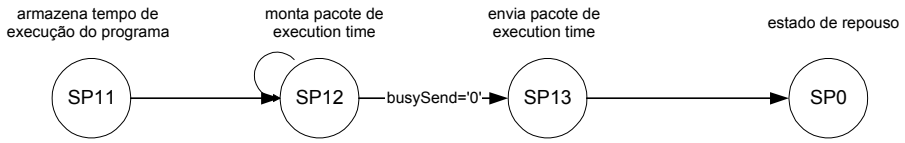


Figura 62 – Máquina de estados parcial do núcleo Processador.

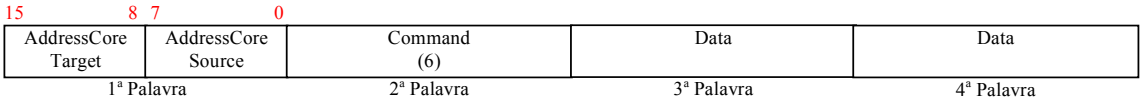


Figura 63 – Pacote do comando *execution time*.

- O estado SP14 (Figura 64) é alcançado quando o comando *wait* é executado. Neste estado é armazenado o endereço do núcleo do qual se está esperando o comando *notify* e a máquina avança automaticamente para o estado SP15. No estado SP15 é aguardada a chegada de um pacote contendo o comando *notify* (Figura 58) enviado pelo núcleo armazenado no estado SP14. Ao receber um pacote com estas características, a máquina avança para o estado SP1, dando seqüência ao processamento.

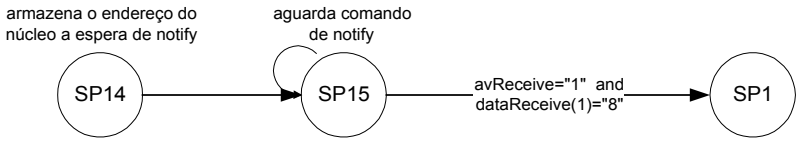


Figura 64 – Máquina de estados parcial do núcleo Processador.

7 Integração e Validação da Arquitetura com Interconexão por Barramento

Este Capítulo aborda a integração dos núcleos Serial (Seção 4.3), Memória (Seção 5.2) e Processador (Seção 6.2), como apresentado na Figura 65 (sistema monoprocessado) e na Figura 79 (sistema biprocessado).

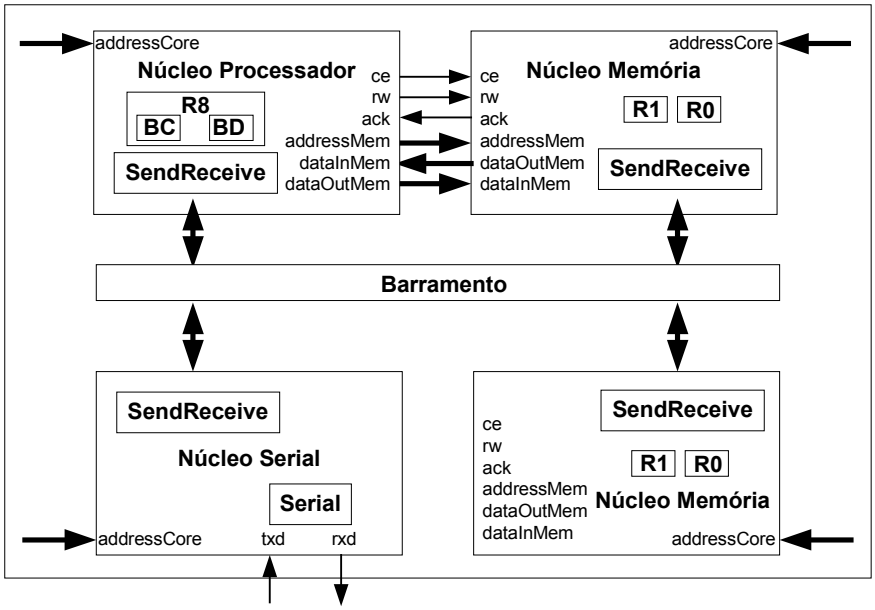


Figura 65 – Diagrama de blocos do sistema monoprocessado com dois núcleos Memória.

A Seção 7.1 descreve o algoritmo utilizado como estudo de caso para o sistema monoprocessado, bem como os critérios que levaram à escolha do mesmo. A Seção 7.2 apresenta os comentários das simulações utilizadas para validar o sistema monoprocessado e a Seção 7.3 mostra os resultados da prototipação deste sistema. A Seção 7.4 descreve o algoritmo utilizado como estudo de caso para o sistema biprocessado. A Seção 7.5 apresenta a validação funcional do sistema biprocessado e os resultados da prototipação são encontrados na Seção 7.6

7.1 Estudo de Caso do Sistema Monoprocessado

Para a escolha do algoritmo utilizado como estudo de caso adotou-se os seguintes critérios:

1. O número de instruções em código objeto do algoritmo não deve exceder a 2048 palavras. Este número representa a capacidade máxima de armazenamento da memória local do processador.
2. O número de dados em código objeto do algoritmo também não deve exceder a 2048 palavras. Este número é o máximo de dados que podem ser armazenados na memória extra (remota) inserida no sistema.
3. O algoritmo deve utilizar não somente o barramento processador-memória, mas também o barramento do sistema. Esse critério foi adotado para que o desempenho global do sistema pudesse ser avaliado e para que fossem gerados dados para futuras comparações.

Com base nos critérios acima mencionados, foi escolhido como estudo de caso o algoritmo de ordenação *Bubble Sort*. Após a escolha do algoritmo, o código assembly (Anexo 1) foi escrito e testado no ambiente de montagem e simulação do processador R8 [MOR01], apresentado na Figura 66. Este ambiente gera o código objeto que é armazenado nas memórias do sistema, sendo o código objeto das instruções do algoritmo armazenadas na memória local do processador e os dados em uma memória remota do sistema.

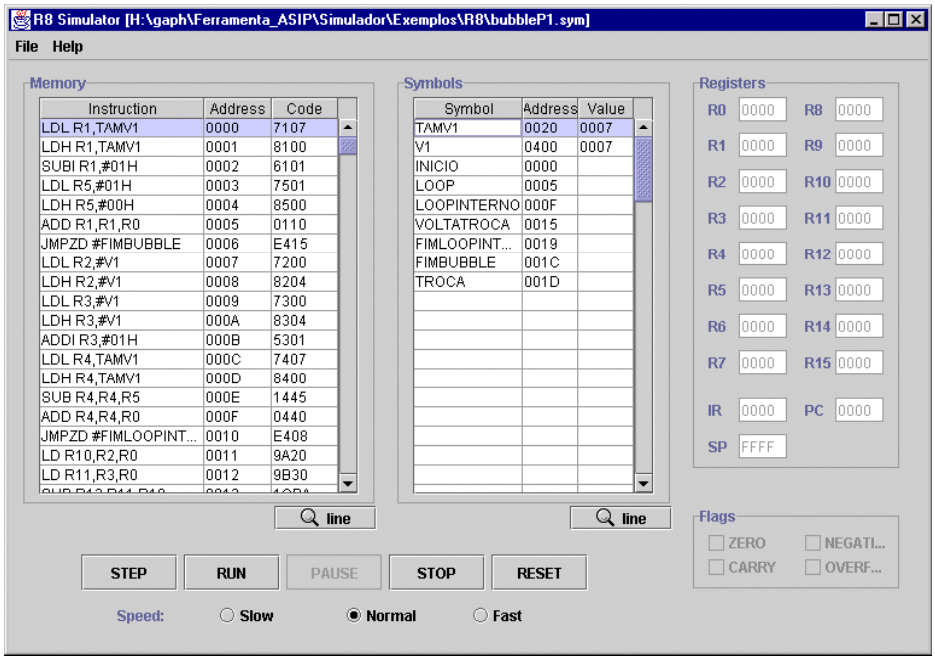


Figura 66 – Ambiente de montagem e simulação do processador R8.

7.2 Validação Funcional do Sistema Monoprocessado

Para realizar a validação dos núcleos independentes e do sistema completo foi utilizada a ferramenta Active-HDL versão 5.1, disponível no laboratório de desenvolvimento do projeto (GAPH). A seguir serão apresentadas e comentadas partes da simulação do sistema monoprocessado.

A Figura 67 apresenta a simulação do software Serial enviando uma mensagem ao núcleo Serial.

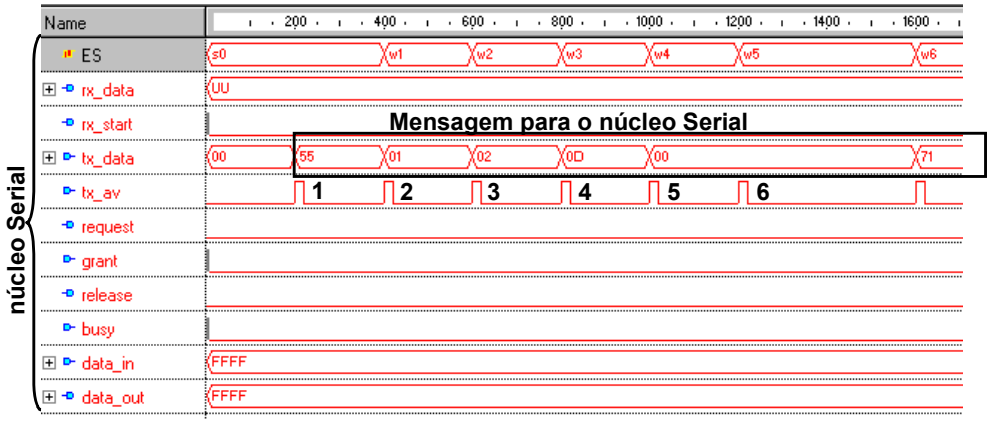


Figura 67 – Simulação do software Serial enviando a mensagem de escrita na memória ao núcleo Serial.

Nesta simulação observa-se que o primeiro dado enviado (1 da Figura 67) é o byte de sincronização do módulo Serial e a partir do segundo dado é enviada a mensagem. Com base nos dados da mensagem, pode-se identificar que se trata de um comando de escrita na memória (2), que é endereçado ao núcleo 2 do sistema (3), que contém 13 dados a serem escritos (4) e que a posição de memória inicial da escrita é a zero (5 e 6). A partir do início do recebimento da mensagem o núcleo Serial executa operações conforme o estado atual da máquina Serial. Esta mensagem é semelhante à executada para armazenar as instruções do algoritmo *Bubble Sort* na memória local do processador e os dados na memória remota do sistema.

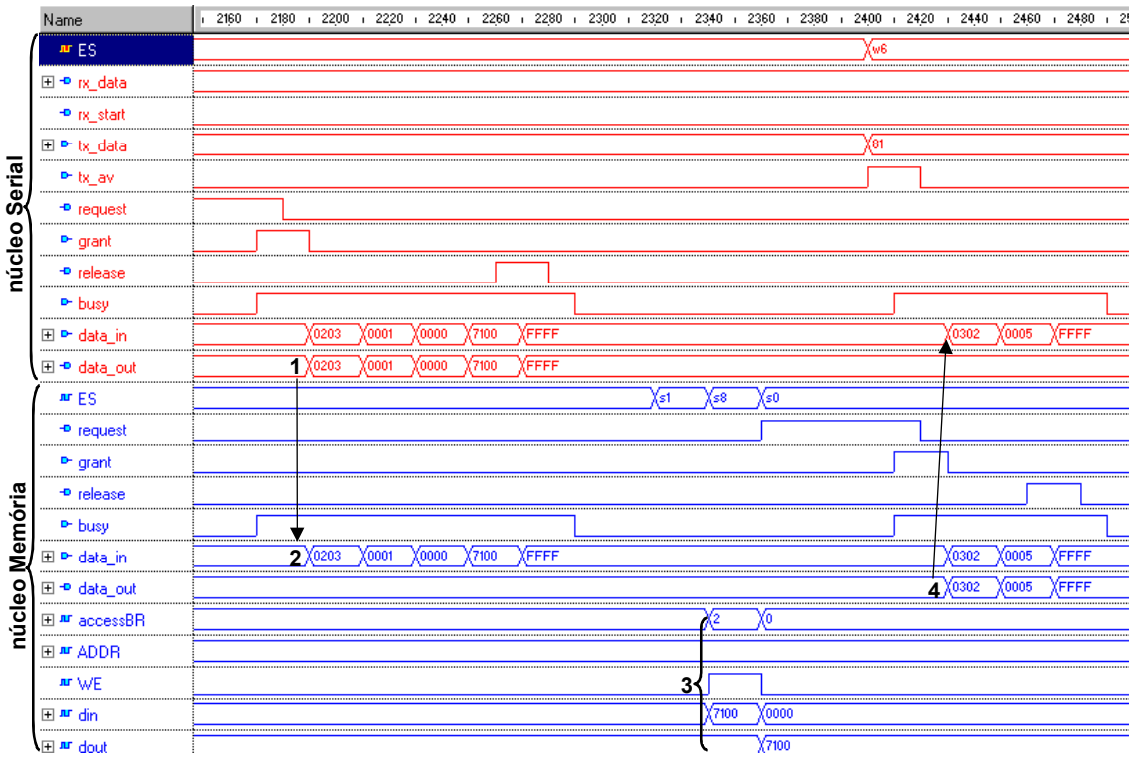


Figura 68 – Simulação do núcleo Serial enviando o pacote de escrita na memória ao núcleo Memória.

A Figura 68 apresenta a continuação da simulação, onde o núcleo Serial, ao receber o primeiro dado a ser escrito na posição inicial informada, monta o pacote de escrita na memória e o envia ao núcleo 2 (1 da Figura 68).

Nesta parte da simulação podemos observar o núcleo Serial acessando o barramento e enviando o pacote de escrita na memória. O núcleo Memória ao receber o pacote, identifica que este é endereçado a ele (2) e realiza a operação de escrita (3).

Todo recebimento de pacote provoca o envio de uma confirmação (4). Portanto, ao receber o pacote de escrita na memória o núcleo Memória monta o pacote de confirmação e requisita o barramento para o envio deste ao núcleo Serial.

Como cada pacote de escrita na memória possui um dado, o envio de pacotes do núcleo Serial ao núcleo Memória repete-se até que os 13 dados informados na mensagem enviada pelo software Serial sejam escritos na memória.

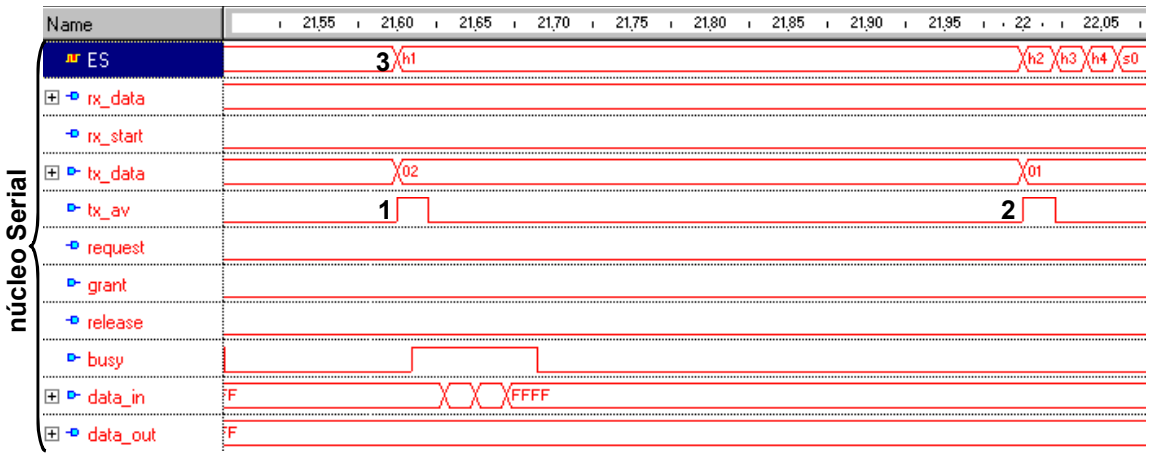


Figura 69 – Simulação do software Serial enviando a mensagem de habilitação do processador ao núcleo Serial.

A simulação apresentada na Figura 69 mostra o software Serial enviando uma mensagem de habilitação do processador ao núcleo Serial. A mensagem é identificada pelo comando 2 (1 da Figura 69) e é destinada ao núcleo 1 do sistema (2). A partir da recepção da mensagem a máquina Serial avança para o estado que trata esta mensagem (3), como apresentado na Seção 4.3.3. A continuação das ações provocadas pelo envio da mensagem de habilitação do processador pode ser visualizadas na Figura 70.

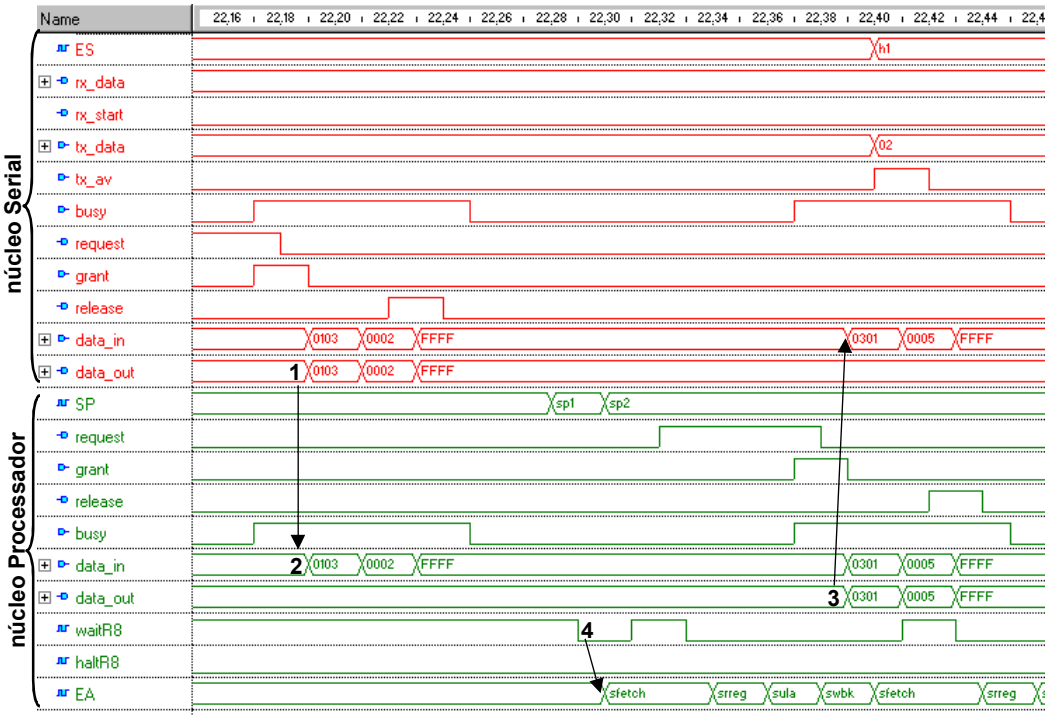


Figura 70 – Simulação do núcleo Serial enviando o pacote de habilitação ao núcleo Processador.

A Figura 70 mostra o envio do pacote de habilitação do núcleo Serial para o núcleo Processador (1). O núcleo Processador ao receber o pacote, verifica se o mesmo tem como núcleo destino o seu endereço (2). Se esta condição for satisfeita, o núcleo Processador envia um pacote de confirmação ao núcleo Serial (3) e desativa o sinal *waitR8* (4), fazendo com que o processador R8 deixe o estado de repouso e comece a executar as instruções armazenadas na memória local.

A Figura 71 mostra o núcleo Processador acessando a memória local para obter a primeira instrução do programa.

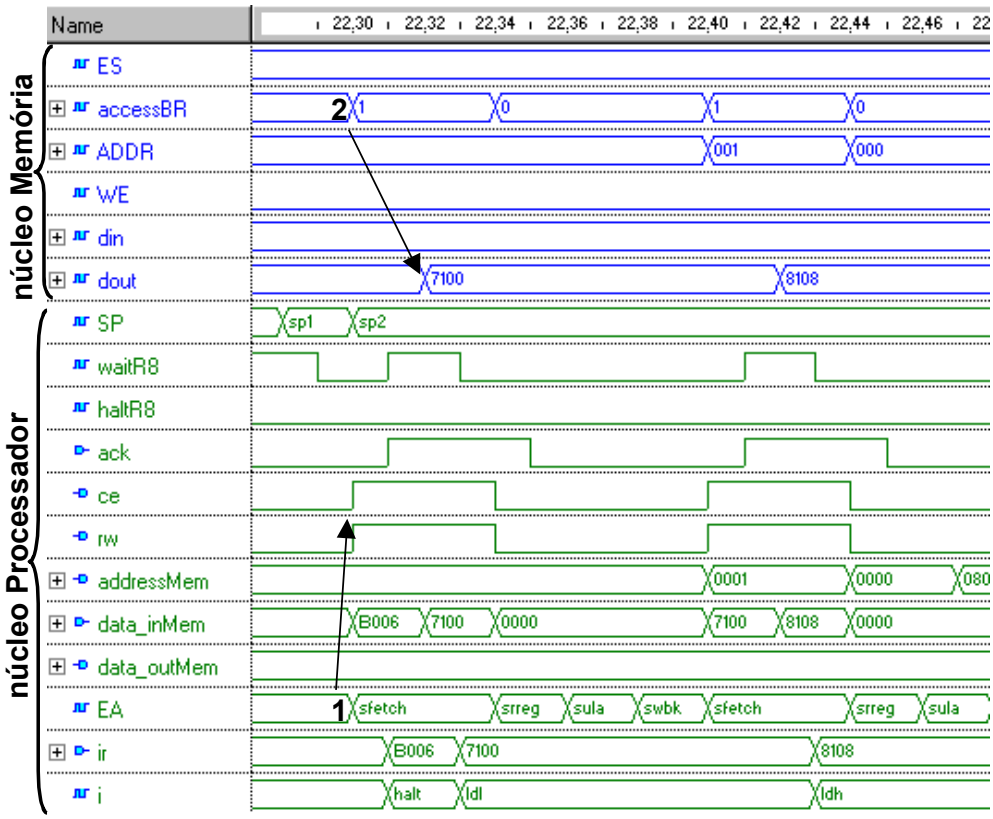


Figura 71 – Simulação do núcleo Processador acessando o núcleo Memória correspondente a sua memória local.

Toda vez que o processador R8 encontra-se no estado Sfetch (1 da Figura 71), ele realiza um acesso a memória local para buscar a próxima instrução a ser executada. Neste momento o núcleo Memória concede o direito de acesso aos módulos de memória ao processador, que realiza a leitura da instrução (2). Este acesso à memória local é realizado através do barramento processador-memória.

A Figura 72 apresenta o núcleo Processador acessando a memória remota. Este acesso necessita de montagem e envio de pacotes através do barramento do sistema. Por este motivo, acessar à memória remota apresenta uma sobrecarga maior no tempo de execução.

Toda vez que o processador R8 executa uma instrução de leitura ou escrita, o núcleo Processador verifica qual posição de memória o processador deseja ler ou escrever e a qual Núcleo Memória esta posição pertence. Quando a posição não pertence à memória local do processador é necessário verificar a qual endereço global a posição corresponde (rever Figura 57). Após esta verificação é enviado o pacote solicitando a leitura ou escrita na memória (1 da Figura 72) e no caso de leitura é aguardado o retorno com o dado lido. O núcleo Memória ao receber o pacote realiza a operação na memória (3) e envia o pacote de confirmação (2). No caso de leitura é enviado o pacote contendo o dado lido (4). Este acesso à memória remota, realizado através do barramento do sistema, descreve exatamente como funcionam as operações de leitura e escrita no vetor a ser ordenado pelo algoritmo *Bubble Sort*.

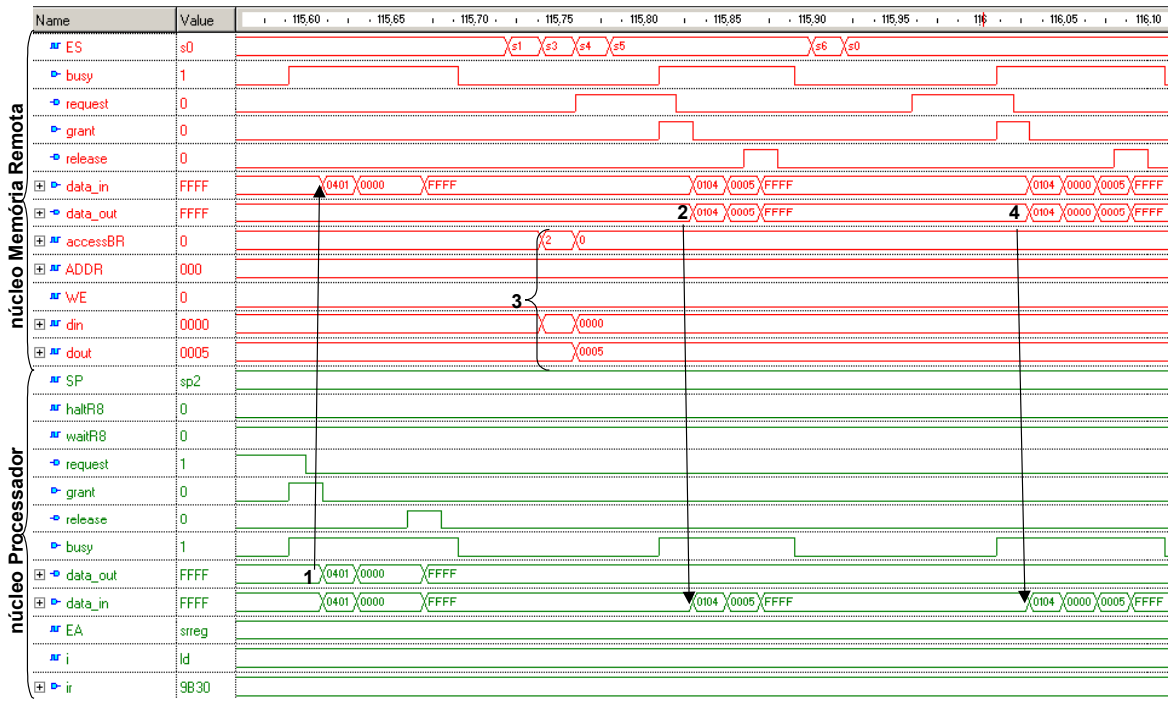


Figura 72 – Simulação do núcleo Processador acessando o núcleo de Memória remota.

Ao finalizar a execução das instruções o processador R8 ativa o sinal *haltR8* (1 da Figura 73), o núcleo Processador então captura o número de ciclos gastos desde a habilitação do processador até o momento e envia um pacote com esta informação ao núcleo Serial (2). Esta operação é mostrada na Figura 73.

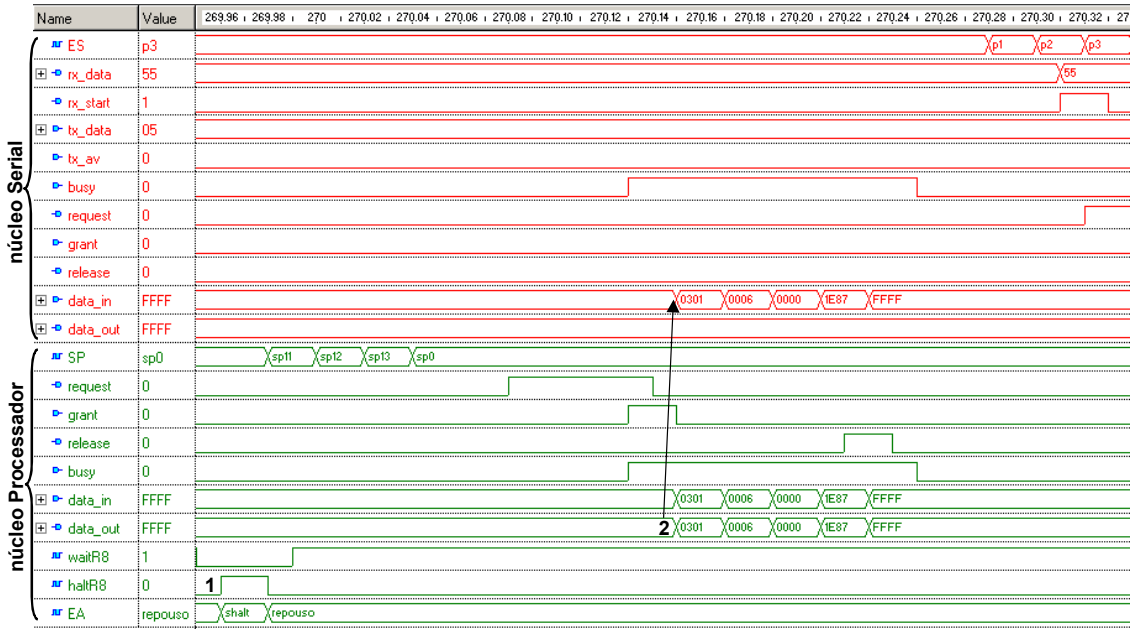


Figura 73 – Simulação do núcleo Processador enviando o comando *execution time* à Serial.

O núcleo Serial, ao receber o pacote contendo o número de ciclos de relógio gastos durante a execução das instruções do núcleo Processador, envia ao software Serial esta informação que é disponibilizada ao usuário. O número de ciclos de relógio gastos pelo processador para executar o algoritmo *Bubble Sort* sobre um vetor de 512 posições com valores aleatórios foi 16.032.562 ciclos.

7.3 Resultados da Prototipação do Sistema Monoprocessado

O sistema monoprocessado, apresentado na Figura 65, foi completamente descrito em VHDL. Ele é composto por 11 pares entidade-arquitetura, compreendendo a hierarquia apresentada na Figura 74.

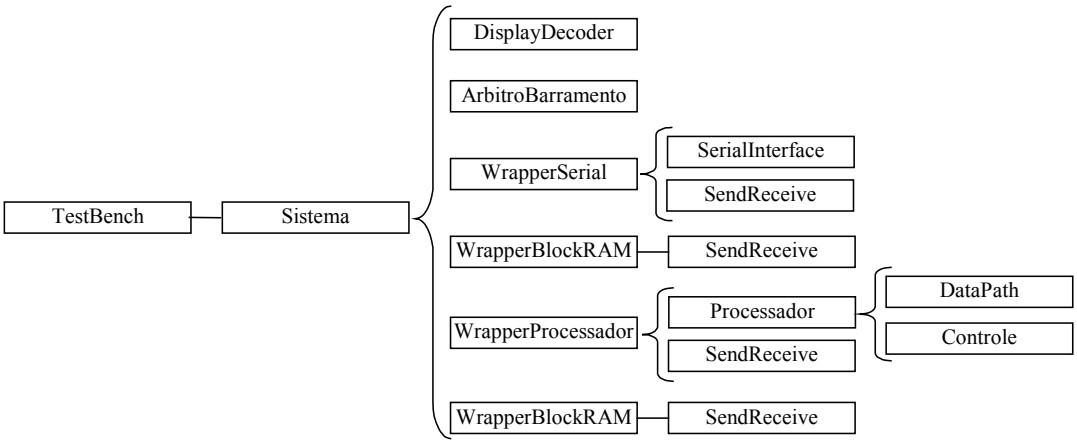


Figura 74 – Hierarquia do projeto monoprocessado.

O código VHDL é descrito no nível RTL de abstração. Excetuando-se o módulo de memória, que utiliza recursos específicos da arquitetura VIRTEX (BlockRAM), o código VHDL é portável para outras famílias de FPGA ou para implementação ASIC.

A interface externa do sistema monoprocessado compreende os seguintes sinais:

- *clock*: relógio do sistema.
- *sw5*: botão da plataforma de prototipação ativo em zero que inicializa o sistema.
- *txd*: sinal que envia dados pela serial.
- *rxd*: sinal que recebe dados da serial.
- *display*: sinal que envia dados para visualização no display da plataforma de prototipação. Utilizado para depuração do sistema.

A Ferramenta ISE 5.1 foi utilizada para realizar a síntese lógica e física do par entidade-arquitetura *Sistema*, com a pinagem externa descrita no arquivo UCF (user constraints file), conforme o Anexo 4.

Os resultados obtidos com a síntese, para o dispositivo XC2V1000, foram:

Número de LUTS.....	2.062 (20%)
Número de portas lógicas equivalentes..	292.646
Número de blockRAMs.....	4
Número de clocks globais.....	1
Freqüência estimada.....	11.526 MHz

Apesar da freqüência de operação estimada ser de 11,5 MHz, o sistema opera corretamente com uma freqüência de 20 MHz. Uma vez que o arquivo de configuração *sistema.bit* é carregado na placa, procede-se ao teste de execução com o auxílio do software Serial.

A Figura 75 mostra o envio das instruções do algoritmo *Bubble Sort* para o FPGA. A

Figura 76 mostra a recepção do vetor ordenado, operação esta executada pelo processador implementado no FPGA.

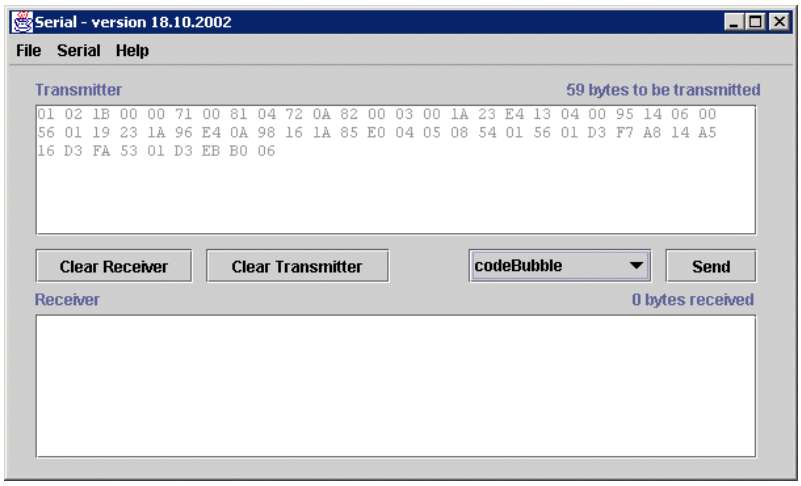


Figura 75 – Envio das instruções do algoritmo *Bubble Sort* para a memória local do processador.

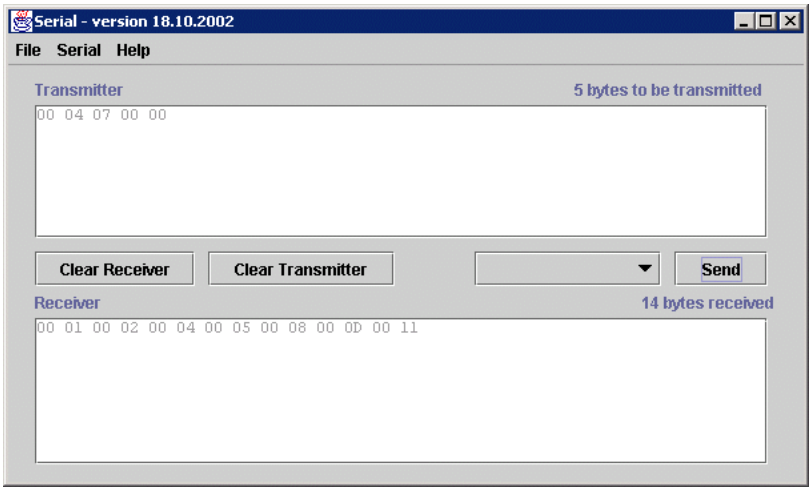


Figura 76 – Leitura da memória remota do vetor ordenado pelo processador.

A Figura 77 apresenta um dado impresso em hexadecimal no terminal do processador através do comando de saída e a Figura 78 mostra o terminal do processador com uma entrada do usuário.

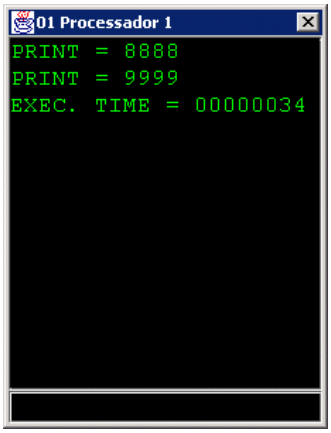


Figura 77 – Exemplo de comando de saída.

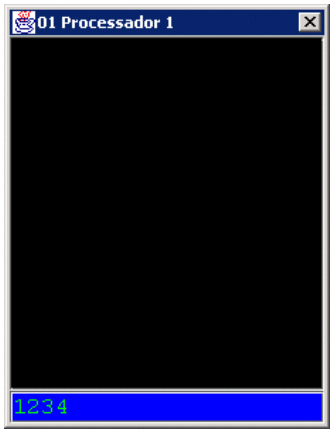


Figura 78 – Exemplo de comando de entrada.

Estes dados mostram que o sistema monoprocessado está operando corretamente na plataforma de prototipação. O passo seguinte deste trabalho é estender o sistema monoprocessado para biprocessado, atividade apresentada nas próximas seções.

7.4 Estudo de Caso do Sistema Biprocessado

O algoritmo utilizado como estudo de caso para o sistema biprocessado (Figura 79) foi a combinação do algoritmo *Bubble Sort* (estudo de caso do sistema monoprocessado) e do algoritmo de *Merge*. O algoritmo *Bubble Sort* é executado pelos dois processadores do sistema, P1 e P2, e o algoritmo de *Merge* é executado apenas pelo processador P1.

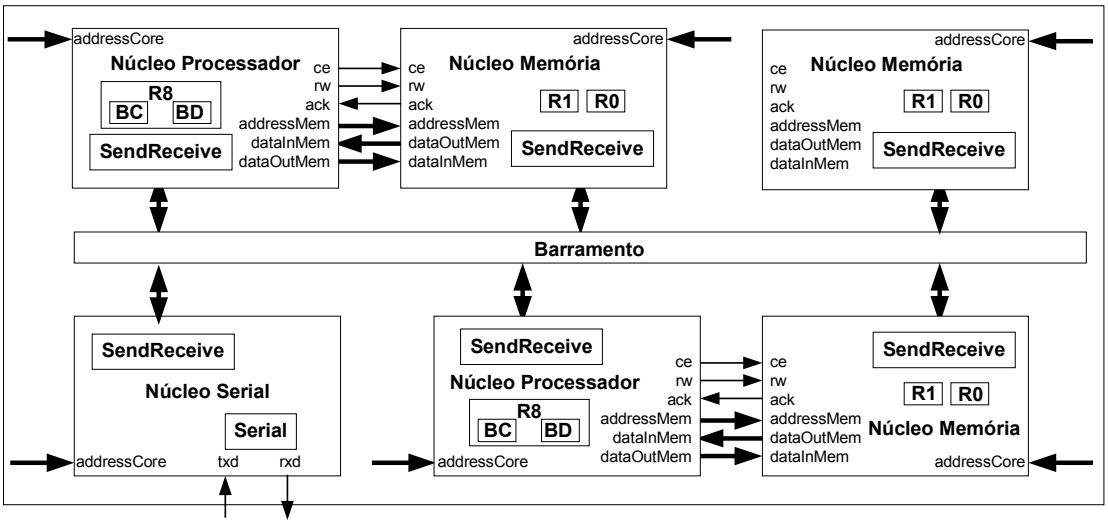


Figura 79 – Diagrama de blocos do sistema biprocessado.

O processador P1 executa o algoritmo de *Bubble Sort* sobre a metade do vetor e ao terminar executa a instrução *wait*, fazendo com que o processador P1 fique bloqueado até receber o *notify* do processador P2. Este bloqueio é necessário porque o processador P1 só pode prosseguir após o processador P2 terminar a execução do algoritmo *Bubble Sort*. Ao ser desbloqueado pelo *notify*, o processador P1 executa o algoritmo de *Merge* sobre todo o vetor e finaliza sua execução.

O processador P2 executa o algoritmo de *Bubble Sort* na outra metade do vetor, ao terminar executa a instrução *notify*, fazendo com que seja enviado o *notify* ao processador P1,

e finaliza sua execução.

Com o algoritmo definido, o código assembly do processador P1 (Anexo 2) e do processador P2 (Anexo 3) foi escrito e testado no ambiente de montagem e simulação do processador R8, apresentado na Figura 66. O ambiente gerou os códigos objetos que foram armazenados nas memórias do sistema, sendo os códigos objetos das instruções armazenadas nas memórias locais dos processadores P1 e P2 e os dados do vetor em uma memória remota do sistema.

7.5 Validação Funcional do Sistema Biprocessado

A validação do sistema biprocessado, como a do sistema monoprocessado, foi realizada na ferramenta Active-HDL versão 5.1.

Antes de iniciar a validação do sistema biprocessado é necessário definir os endereços dos núcleos do sistema. Para simplificar, o sistema biprocessado manteve os endereços dos núcleos já existentes no sistema monoprocessado, incluindo novos endereços para os núcleos inseridos no sistema biprocessado, conforme a Tabela 3.

Tabela 3 – Endereços dos núcleos.

Endereço	Núcleo
1	Processador P1
2	Memória local do processador P1
3	Serial
4	Memória remota
5	Processador P2
6	Memória local do processador P2

O primeiro passo da validação do sistema biprocessado é o envio das instruções do algoritmo do processador P1 para a sua memória local (núcleo 2), o envio das instruções do algoritmo do processador P2 para a sua memória local (núcleo 6) e o envio dos dados do vetor a ser ordenado para a memória remota (núcleo 4). Estes envios são idênticos ao apresentado na Figura 67 que mostra a simulação do software Serial enviando uma mensagem de escrita na memória ao núcleo Serial. A simulação do núcleo Serial enviando o pacote de escrita na memória ao núcleo Memória também é idêntica à simulação apresentada na Figura 68.

O segundo passo da validação é o envio das mensagens de habilitação dos processadores P1 e P2 ao núcleo Serial (Figura 69). A Figura 80 mostra a continuação das ações provocadas pelo envio da mensagem de habilitação do processador pelo software Serial. Nesta simulação, o núcleo Serial envia o pacote de habilitação para o núcleo Processador P2 (1 da Figura 80), todos os núcleos do sistema recebem o pacote e verificam se o mesmo tem como núcleo destino o seu endereço (2). O núcleo Processador P2 ao identificar que recebeu o comando de habilitação envia um pacote de confirmação ao núcleo Serial (3) e desativa o sinal *waitR8* (4), fazendo com que o processador R8 deixe o estado de repouso e comece a executar as instruções armazenadas na memória local. A habilitação do processador P2 ocorre em paralelo com a execução do processador P1 (5), habilitado anteriormente.

Durante a execução das instruções dos algoritmos dos processadores P1 e P2, estes

realizam acessos a memória remota (núcleo 4) para lerem os valores do vetor a ser ordenado e escreverem os valores devidamente ordenados, como mostrado na Figura 72.

O núcleo Processador P1 ao terminar de executar o algoritmo de *Bubble Sort* na sua parte do vetor bloqueia seu processador R8 (1 da Figura 81) e aguarda o comando *notify* do processador P2 (2). O núcleo Processador P2 ao terminar a execução do algoritmo em sua parte do vetor envia o comando de *notify* ao processador P1 (3). O núcleo Processador P1 ao receber o comando *notify* libera seu processador R8 (4) que passa a executar as instruções do algoritmo de *Merge* (5) e envia a confirmação de recebimento ao processador P2 (6). O processador P1 e o processador P2 ao finalizarem suas execuções enviam o comando *execution time* ao núcleo Serial, como mostrado na Figura 73.

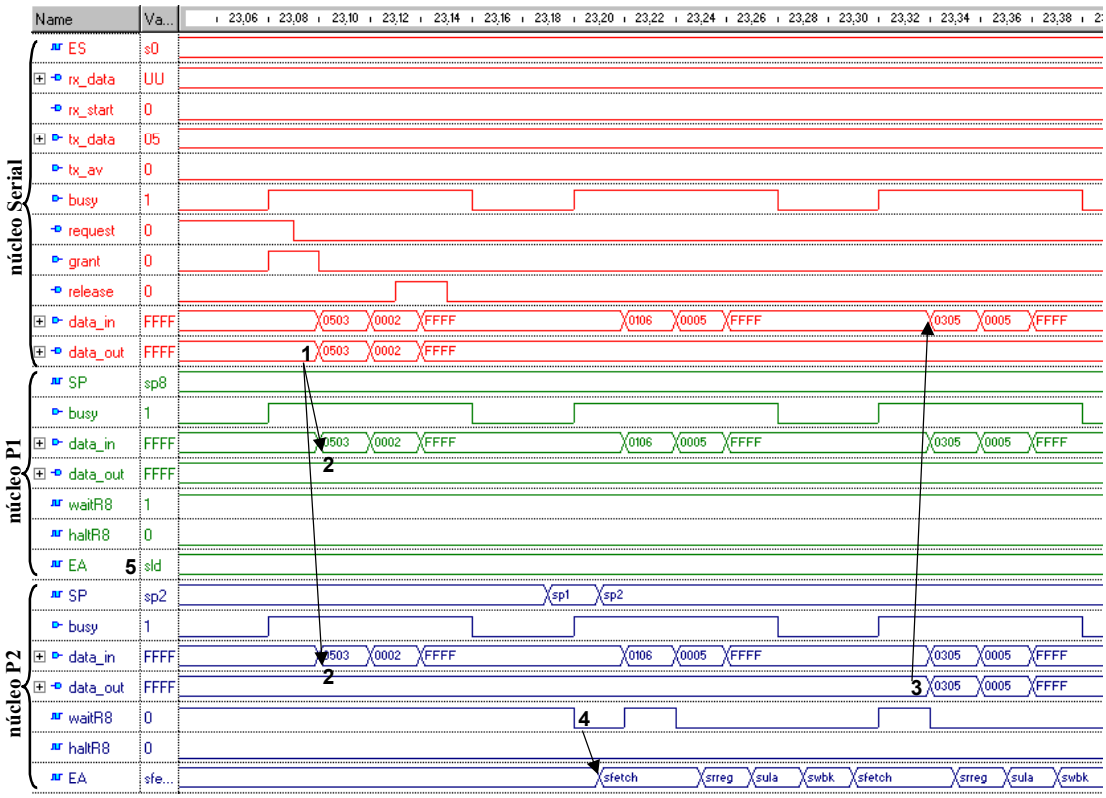


Figura 80 – Simulação do núcleo Serial enviando o pacote de habilitação ao processador P2.

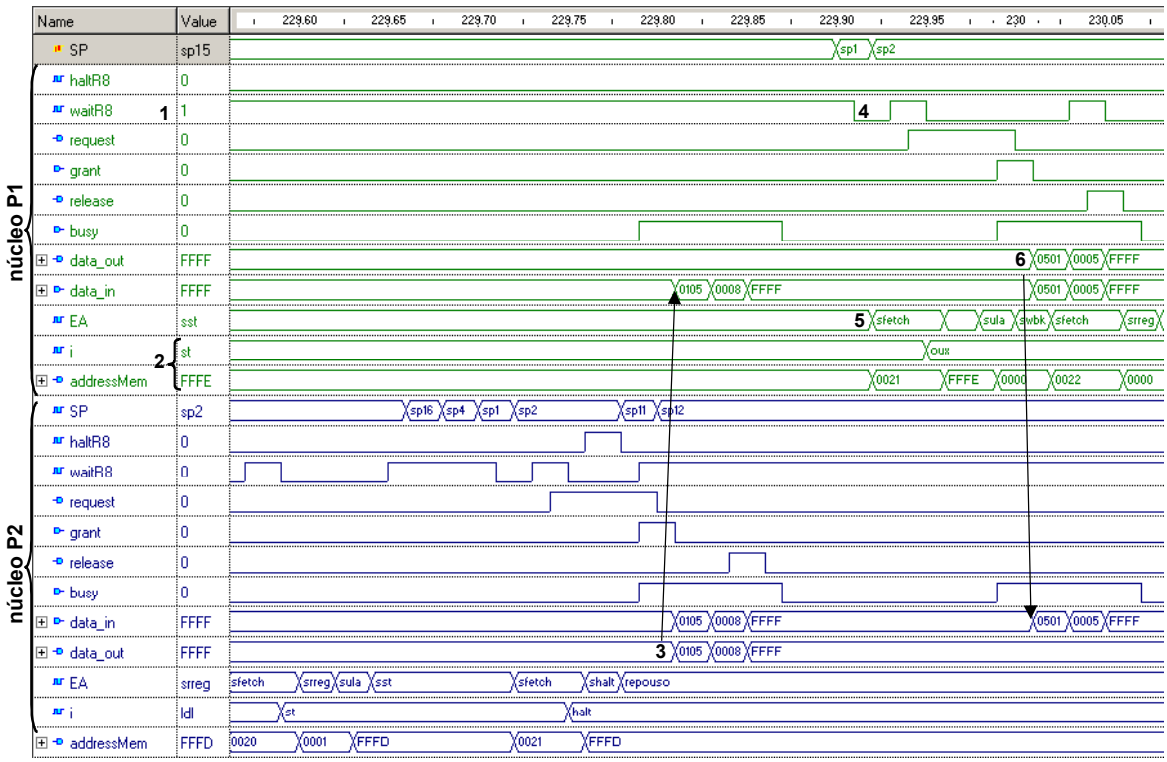


Figura 81 – Simulação do núcleo Processador P1 aguardando o comando *notify* e o núcleo Processador P2 enviando o mesmo.

O número de ciclos de relógio gastos pelo processador P2 para executar o algoritmo *Bubble Sort* sobre um vetor de 256 posições foi 11.089.602 ciclos e o número de ciclos de relógio gastos pelo processador P1 para executar o algoritmo *Bubble Sort* sobre um vetor também de 256 posições, esperar o comando *notify* e executar o algoritmo de *Merge* sobre as 512 posições foi de **12.203.008** ciclos, este número é menor que o apresentado pelo sistema monoprocessado na ordenação das mesmas 512 posições (**16.032.562** ciclos). Portanto, o sistema biprocessado, embora tenha um período de espera pelo comando *notify,* obteve um desempenho superior ao sistema monoprocessado, de **24%**. É importante ressaltar que os dados a serem ordenados encontram-se em uma memória unificada (remota). Este fato conduz à disputa de acesso ao barramento e à memória remota pelos processadores, reduzindo assim o ganho de desempenho.

7.6 Resultados da Prototipação do Sistema Biprocessado

O sistema biprocessado, apresentado na Figura 79, assim como o sistema monoprocessado, também foi descrito em VHDL. Ele é composto pelos mesmos 11 pares entidade-arquitetura do sistema monoprocessado e compreende a hierarquia apresentada na Figura 82.

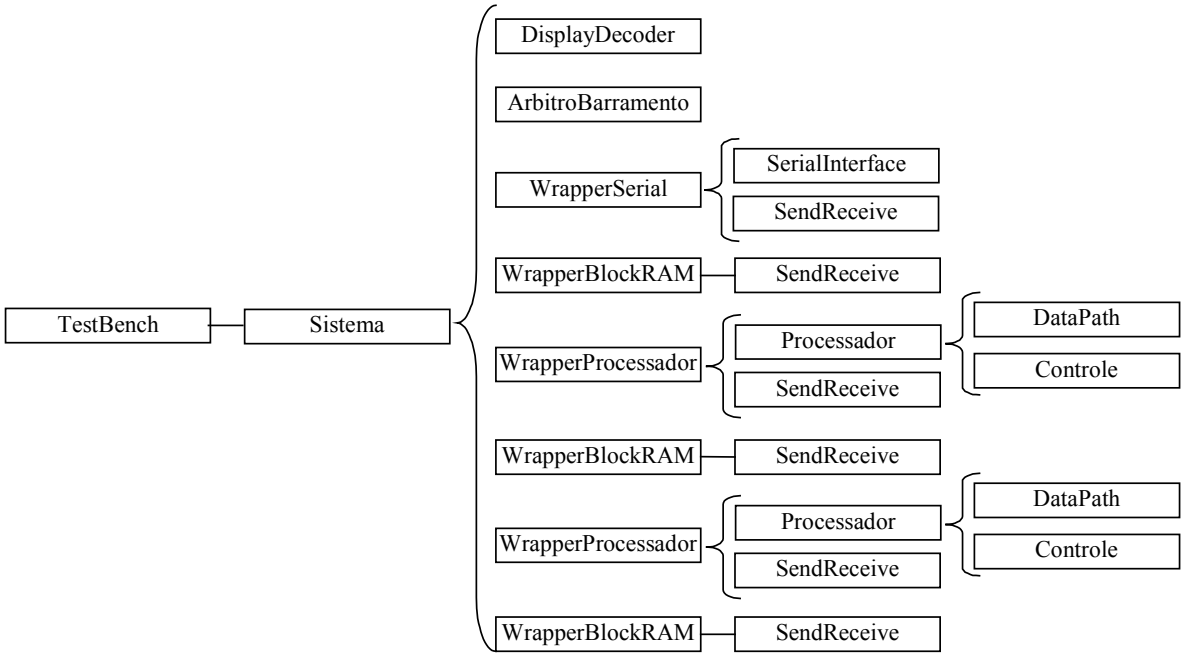


Figura 82 – Hierarquia do projeto biprocessado.

O código VHDL do sistema biprocessado é portável para outras famílias de FPGA se respeitada a exceção do módulo de memória, apresentada na Seção 7.3.

A interface externa do sistema biprocessado é idêntica a do sistema monoprocessado e sua pinagem externa é descrita no arquivo UCF (*user constraints file*), conforme o Anexo 4.

Os resultados obtidos com a síntese, para o dispositivo XC2V1000, foram:

Número de LUTs.....	3.296 (32%)
Número de portas lógicas equivalentes...	441.573
Número de blockRAMs.....	6
Número de clocks globais.....	1
Freqüência estimada.....	11.290 MHz

A partir destes dados, pode-se deduzir que a ocupação de área pelo processador R8 e seu invólucro é de 1.234 LUTs (3.296 - 2.062 LUTs). Uma vez que o arquivo de configuração *sistema.bit* é carregado na placa, procede-se ao teste de execução com o auxílio do software Serial. Primeiramente são enviados os códigos objeto respectivos dos processadores P1 e P2 para as suas memórias locais e os dados do vetor a ser ordenado para a memória remota. A seguir são habilitados os processadores P1 e P2 que passam a executar as instruções armazenadas em suas memórias locais. Ao final da execução os processadores enviam o número de ciclos gastos com a execução de seus algoritmos, como mostra a Figura 83 e a Figura 84. Esta operação finaliza o teste de execução do algoritmo paralelo no sistema biprocessado.

Figura 83 – Tempo de execução do processador P1 em hexadecimal.

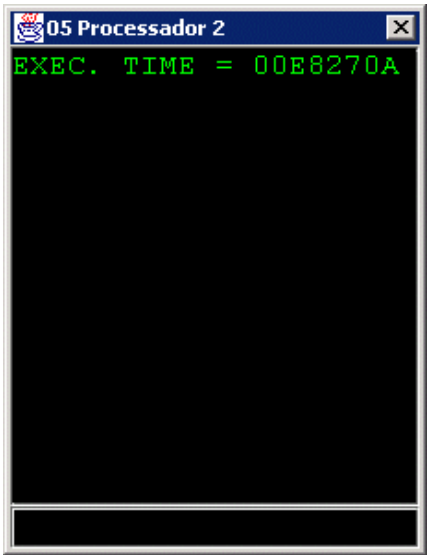


Figura 84 – Tempo de execução do processador P2 em hexadecimal.

Embora as tarefas de validação funcional, prototipação e execução do sistema multiprocessado com 4 processadores não tenham sido apresentadas, estas foram realizadas e o resultado foi o consumo de **6.860.227** ciclos de relógio para a execução do algoritmo de ordenação *Bubble Sort* e *Merge* sobre o mesmo vetor de 512 posições. Isto representa um ganho de desempenho de **44**% em relação ao sistema biprocessado (**12.203.008** ciclos) e **57**% em relação ao sistema monoprocessado (**16.032.562** ciclos).

8 Redes Intra-chip: Conceitos e Decisões de Projeto

Os Capítulos anteriores abordaram a estrutura de interconexão de núcleos de hardware por barramento. O barramento, embora possibilite interconectar diversos núcleos, tem escalabilidade limitada e permite que apenas uma comunicação possa ser efetuada por instante de tempo (seção 1.3.2). Outras restrições dos barramentos são a necessidade de mecanismos de arbitragem do meio físico para evitar desperdício de banda e o uso de fios longos, os quais limitam o desempenho do sistema devido às altas capacitâncias e resistências parasitas inerentes a estes. Estas desvantagens podem ser parcialmente contornadas através do uso de, por exemplo, hierarquia de barramentos, onde o problema continua existindo, sendo apenas minimizado.

O aumento da complexidade dos sistemas computacionais aponta para a necessidade de examinar uma forma mais avançada de interconexão de núcleos, buscando principalmente paralelismo de comunicação e escalabilidade. Uma possível solução pode ser buscada nas redes de computadores e aplicada a sistemas embarcados, criando o conceito de NoC (Network-on-Chip) [GUE99] [KUM02]. NoCs herdam das redes de computadores e de sistemas distribuídos as características de camadas de protocolos e o conceito de ligação de nodos à rede (apresentados na Seção 8.1). Nas NoCs, os núcleos do sistema são interligados por meio de uma rede composta por chaves e canais ponto-a-ponto. A comunicação entre os núcleos ocorre pela troca de mensagens transferidas por meio de chaves e canais intermediários até atingir o seu destino [BEN02].

Topologias NoC são definidas pela estrutura de interconexão entre as chaves. Conforme visto na Seção 1.3.3 existem diversas topologias de rede. Neste trabalho optou-se pela implementação de uma rede direta estritamente ortogonal, pois estas simplificam os algoritmos de chaveamento, são facilmente escaláveis e permitem melhor compreender o comportamento de uma NoC. Dentre as topologias de rede direta foi escolhida a topologia Mesh (Grelha). Esta escolha justifica-se pela regularidade física e a conseqüente implementação no FPGA. A chave proposta para a topologia Mesh também pode ser usada para construir topologias do tipo Torus, Hipercubo e similares. Entretanto, construir estas topologias implica em modificações nas conexões entre as chaves e no algoritmo de chaveamento.

As duas principais características que compõe uma NoC são os *serviços* e o *sistema de comunicação*. Rijpkema et al. [RIJ01] descrevem como alguns *serviços* essenciais à manutenção da integridade de dados, *throughput* e a latência em NoCs. O *sistema de comunicação* suporta a transferência de comunicação entre uma origem e um destino. Como definido por Duato [DUA97], a comunicação entre elementos da rede é baseada na transmissão de pacotes, que são normalmente compostos de um cabeçalho, um corpo e um terminador. O sistema de comunicação permite que cada núcleo envie pacotes para qualquer outro núcleo conectado a NoC.

Dois modos para o roteamento de uma comunicação entre uma origem e um destino são o *chaveamento por circuito* e o *chaveamento por pacote* (Seção 8.2). A utilização de transferência de mensagens por *chaveamento por pacote* implica na utilização de um modo de recebimento e envio de pacotes. Os modos mais populares são *store-and-forward, virtual-cut-*

through e *wormhole* (Seção 8.3). Além da decisão de qual modo de roteamento e qual modo de envio e recebimento de pacotes será adotado, existem outras decisões quanto ao local onde é realizado o roteamento, quanto ao número de caminhos possíveis e quanto ao caminho percorrido pelo pacote (Seção 8.4).

A NoC também precisa assegurar a correta funcionalidade da transferência de mensagens, evitando *deadlock*, *livelock* e *starvation* (Seção 8.5). Dependendo das decisões de projeto é necessário agregar mecanismos que evitem ou tratem essas ocorrências.

8.1 Níveis de Comunicação

O modelo de referência OSI é um padrão internacional de redes de computadores que define níveis hierárquicos com o objetivo de delimitar e isolar funções de comunicações em camadas de protocolos. Cada nível oferece um conjunto de serviços ao nível superior, usando funções realizadas no próprio nível e serviços disponíveis nos níveis inferiores. NoCs implementam algumas das funções dos níveis inferiores (Físico, Enlace, Rede e Transporte) do modelo OSI que são descritas a seguir.

8.1.1 Nível Físico

O nível físico é responsável pela transferência em nível de bits através de um enlace. No presente trabalho este nível corresponde à implementação da interface de comunicação entre as chaves, como apresentado na Figura 85.

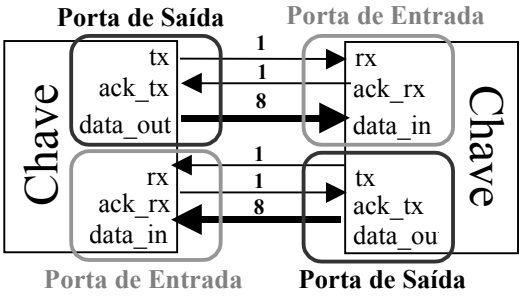


Figura 85 – Interface entre chaves.

A porta de saída é composta pelos seguintes sinais:

- *Tx*: indica que existe um dado a ser enviado.
- *Data_out*: contém o dado (8 bits) a ser enviado.
- *Ack_tx*: indica que o dado foi enviado.

A porta de entrada é composta por:

- *Rx*: indica que existe um dado a ser recebido.
- *Data_in*: contém o dado (8 bits) a ser recebido.
- *Ack_rx*: indica que o dado foi recebido.

A largura do barramento de dados (8 bits) foi escolhida de forma a reduzir os requisitos de chaveamento e o tamanho das filas. Este parâmetro pode ser facilmente parametrizável em função de requisitos de desempenho.

8.1.2 Nível de Enlace

O nível de enlace encarrega-se da comunicação em nível de quadros (grupo de bits), utilizando para isso o nível físico. O nível de enlace tem a preocupação com o enquadramento dos dados e com a transferência desses quadros de forma confiável. O trabalho proposto adota no nível de enlace um protocolo de *handshake* para o envio e recebimento de dados de forma confiável. Neste protocolo, quando uma chave deseja enviar um dado para uma chave vizinha, esta disponibiliza no sinal *data_out* o dado e ativa o sinal *tx* na direção da mesma. Quando a chave vizinha recebe o sinal *rx* ativo, armazena o dado que esta em *data_in* e ativa o sinal *ack_rx*. A chave origem ao receber o sinal *ack_tx* tem a confirmação do envio e pode prosseguir.

8.1.3 Nível de Rede

O nível de rede faz a comunicação em nível de pacotes (grupo de quadros). Responsável pelo empacotamento de mensagens, roteamento de pacotes entre origem e o destino (ponto-a-ponto) e controle de congestionamento. Para compor o nível de rede este trabalho adota o modo de *chaveamento por pacote*.

8.1.4 Nível de Transporte

A camada de transporte é responsável por estabelecer uma conexão fim-a-fim entre a origem e o destino da mensagem. Isto inclui o controle de fluxo, a segmentação e a remontagem de pacotes, provendo um serviço confiável. No presente trabalho essa camada possui apenas a característica de conexão fim-a-fim, não implementando, ainda, as demais funcionalidades atribuídas a ela.

8.2 Modos de Roteamento

Para que uma comunicação seja efetuada na rede deve-se adotar uma para a transferência de mensagens. Os dois métodos mais utilizados são *chaveamento por circuito* e *chaveamento por pacotes* [HWA93] [MOH98].

- Chaveamento por circuito

No método de *chaveamento por circuito* um caminho é estabelecido antes do envio da mensagem. Quando um circuito entre a origem e o destino foi estabelecido, a mensagem pode ser enviada e qualquer outra requisição de comunicação no canal alocado será negada. A grande vantagem deste método é que não são necessárias filas nas chaves intermediárias, pois uma vez que a comunicação foi estabelecida a mensagem não é bloqueada. A desvantagem é que esse método causa a degradação do desempenho do sistema como um todo, porque o caminho fica reservado da chave origem à chave destino durante toda a fase de transmissão de dados.

- Chaveamento por pacotes

No método de *chaveamento por pacotes* a mensagem é quebrada em vários pacotes que são transmitidos pela rede de interconexão. Cada pacote possui um cabeçalho que é verificado na chegada de cada chave intermediária. A chave intermediária, com base no cabeçalho do

pacote, decide para qual porta de saída ela deve enviar o pacote. A grande vantagem do *chaveamento por pacotes* é que o canal permanece ocupado apenas enquanto o pacote está sendo transferido. A desvantagem deste método é que existe a necessidade de filas para o armazenamento temporário de pacotes.

8.3 Modos de Recebimento e Envio de Pacotes

Para estabelecer uma comunicação entre uma origem e um destino de uma mensagem as chaves intermediárias devem assumir uma política de repasse de dados para a chave seguinte. As três políticas mais utilizadas são *store-and-forward*, *virtual-cut-through* e *wormhole* [RIJ01] [MOH98].

- Store-and-Forward

No método *store-and-forward* o pacote inteiro é recebido e armazenado, para só então ser enviado adiante na rede. A conseqüência disto é a necessidade de uma fila capaz de armazenar o pacote inteiro, acarretando em uma alta latência em cada chave intermediária.

- Virtual-Cut-Through

O método de *virtual-cut-through* é um aperfeiçoamento do método *store-and-forward*. A diferença deste método em relação ao anterior é que este só armazena o pacote inteiro caso a chave destino esteja ocupada. Este método, assim como o *store-and-forward*, precisa de uma fila com capacidade para armazenar um pacote inteiro. A vantagem desse método em relação ao *store-and-forward* é que é possível reduzir a latência da comunicação quando a chave seguinte não está ocupada.

- Wormhole

No chaveamento por *wormhole*, o pacote é dividido em *flits*, sendo transmitidos entre as chaves intermediárias até o destino. O método *wormhole* funciona como um pipeline, onde os *flits* de cabeçalho (contendo informações do destino) se movem pela rede e todos os *flits* de dados (*payload*) os seguem. Quando os *flits* de cabeçalho forem bloqueados, os *flits* do corpo do pacote ocuparão as filas das chaves intermediárias. A vantagem do *wormhole* é que a latência não depende diretamente da distância, como os métodos anteriores, mas basicamente do tráfego entre as chaves origem e destino. Outra vantagem deste método é a redução das filas nas chaves intermediárias, que não precisam armazenar o pacote inteiro. A desvantagem do método é a contenção de recursos causada pelo bloqueio do pacote.

8.4 Roteamento

As técnicas de roteamento procuram definir como é realizada a transferência de dados entre uma chave origem da rede e uma chave destino. A seguir são apresentadas várias classificações de técnicas de roteamento segundo vários critérios.

Critério: Quanto ao local de decisão de roteamento

- Roteamento na origem

O caminho é decidido na chave de origem, antes da mensagem ser enviada. Como desvantagens desta abordagem citamos: (i) o cabeçalho do pacote deve carregar todas as informações de roteamento; (*ii*) o método não é tolerante a falhas, ou seja, caso um caminho

apresente defeito, todas as mensagens que deveriam passar por este caminho serão bloqueadas.

- Roteamento distribuído

No método de roteamento distribuído cada chave que recebe o pacote decide para onde irá enviá-lo. A decisão é feita a partir do endereço de destino presente no cabeçalho do pacote.

Critério: Quanto ao número de caminhos possíveis

- Roteamento determinístico

Este método provê sempre o mesmo caminho entre um par origem-destino. Isto ocorre porque ele é baseado em um algoritmo estático que decide qual o caminho a ser tomado em função da chave atual.

- Roteamento adaptativo

O roteamento é definido em função do tráfego da rede. A vantagem do método é que o pacote tem mais de uma alternativa para chegar ao destino. Como desvantagem pode-se citar a possibilidade de *deadlock* e entrega de pacotes fora de ordem. O roteamento adaptativo é subdividido em parcialmente adaptativo e totalmente adaptativo. No método parcialmente adaptativo apenas um subconjunto dos caminhos físicos disponíveis é alocado para a comunicação. No método totalmente adaptativo todos os caminhos físicos da rede podem ser utilizados para enviar uma dada mensagem.

Critério: Quanto ao caminho percorrido

- Roteamento não mínimo

No algoritmo de roteamento por caminho não mínimo a mensagem pode escolher qualquer caminho entre a origem e o destino. A vantagem de uma abordagem como esta é que a mensagem pode desviar dos caminhos bloqueados e assim encontrar um caminho alternativo. A desvantagem é que isso pode causar *livelock* (ver Seção 8.5).

- Roteamento mínimo

No algoritmo de roteamento por caminho mínimo a mensagem é roteada por um dos menores caminhos entre o par origem-destino. A vantagem é a garantia que a cada chaveamento a mensagem se aproxima mais do destino. Outra vantagem é que esta técnica não entra em *livelock* como o roteamento por caminho não mínimo (ver Seção 8.5). A desvantagem é que este algoritmo fica aguardando o caminho escolhido até este ser liberado.

8.5 Tráfego de Pacotes

Para assegurar a correta funcionalidade da rede em termos de entrega de pacotes, a NoC deve evitar *deadlock*, *livelock* e *starvation* [DUA97] [PAT96].

Deadlock pode ser definido como uma dependência cíclica entre nodos requisitando acesso a um conjunto de recursos, de forma que nenhum possa obter progresso algum, independente da seqüência de eventos que ocorra.

Livelock refere-se a pacotes circulando na rede sem fazer nenhum progresso em direção ao seu destino. Este é um problema que normalmente atinge os algoritmos de roteamento adaptativos não mínimos. Isto pode ser evitado com estratégias de roteamento adaptativo,

restringindo o número de desvios que o pacote pode efetuar.

Starvation ocorre quando um pacote de uma fila requisita um canal de saída e é bloqueado porque o canal de saída está sempre alocado a outro pacote.

8.6 Quadro resumo das características da NoC

Após o estudo das características das NoCs foram decididas quais as de maior relevância para a obtenção dos objetivos deste trabalho. O fator de maior importância é a validação da NoC em dispositivos FPGAs. Desta forma, especificações físicas do dispositivo como roteamento, fios, memória on-chip e pinos de entrada e saída devem ser considerados.

A

Tabela 4 apresenta um resumo das características da NoC proposta e implementada neste trabalho.

Tabela 4 – Quadro resumo das características da NoC implementada neste trabalho

Nível Físico	Caminhos bidirecionais de 8 bits com suporte à protocolo *handshake*
Nível Enlace	Protocolo assíncrono, técnica de handshake
Nível Rede	Comunicação ponto-a-ponto, utilizando chaveamento por pacotes
Nível Transporte e superiores	Não implementados
Formato do Pacote	1 *flit* com destino 1 *flit* com tamanho do payload 1-255 *flits* de payload
Chaveamento	Chaveamento por circuito
Repasse de Pacotes	Wormhole
Roteamento	Distribuído, Adaptativo, Caminho Mínimo
Topologia	Mesh (2x2 na prototipação, 5x5 na simulação)
Filas	Na entrada de cada porta, com tamanho parametrizável.

9 Chave

As chaves realizam a transferência de mensagens entre núcleos. Em geral, elas possuem uma lógica para chaveamento e arbitragem, portas de comunicação para outras chaves e, dependendo da topologia, para um núcleo. As portas de comunicação incluem canais de entrada e de saída, os quais podem possuir, ou não, filas para o armazenamento temporário de informações. As portas possuem ainda um controlador de enlace para a implementação do protocolo físico de comunicação.

A chave proposta neste trabalho possui uma lógica de controle de chaveamento e 5 portas bidirecionais: East, West, North, South e Local. Cada porta possui uma fila para o armazenamento temporário de *flits*. A porta Local estabelece a comunicação entre a chave e seu núcleo. As demais portas ligam a chave às chaves vizinhas, conforme apresentado na Figura 86. A lógica de controle de chaveamento engloba arbitragem e lógica de chaveamento.

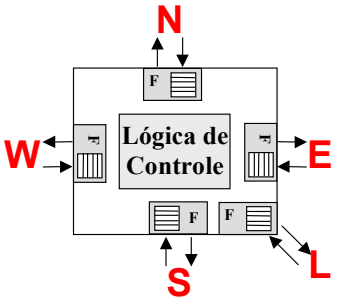


Figura 86 – Estrutura da chave.

O modo de chaveamento wormhole, adotado neste trabalho, possibilita a multiplexação de canais lógicos. Embora essa multiplexação aumente o desempenho do chaveamento [RIJ01], optou-se por utilizar um único canal lógico para cada canal físico, objetivando reduzir a complexidade e o custo da chave.

Os endereços das chaves são compostos pelas coordenadas XY da rede de interconexão, onde X é a posição horizontal e Y a posição vertical. A atribuição de endereços às chaves é necessária para a execução do algoritmo de chaveamento.

A Figura 87 ilustra os módulos principais que compõem a chave, e o protocolo adotado para iniciar e concluir uma conexão será apresentado a seguir.

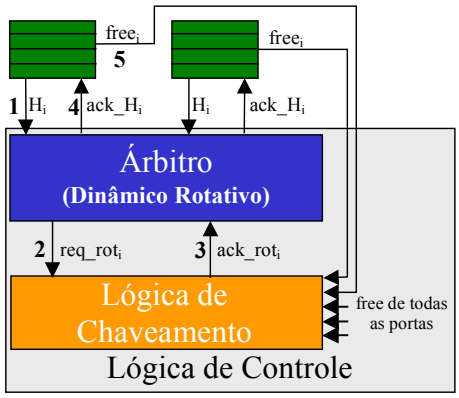


Figura 87 – Protocolo de estabelecimento e conclusão de conexão.

Cada uma das filas da chave (E, W, N, S e L), ao receber um novo pacote requisita chaveamento ao árbitro (Figura 87, 1). O árbitro seleciona a requisição de maior prioridade, quando existem requisições simultâneas, e encaminha o pedido de chaveamento à lógica de chaveamento (Figura 87, 2). A lógica de chaveamento verifica se é possível atender à solicitação. Sendo possível, a conexão é estabelecida e o árbitro é informado (Figura 87, 3). Por sua vez, o árbitro informa a fila (Figura 87, 4) que começa a enviar os *flits* armazenados. Quando todos os *flits* do pacote foram enviados, a conexão é concluída pela sinalização, por parte da fila, através do sinal *free* (Figura 87, 5).

9.1 Lógica de Controle

A lógica de controle é composta por dois módulos: árbitro e lógica de chaveamento, conforme apresentado na Figura 87. Ao receber um pacote, a chave utiliza-se de um algoritmo de chaveamento para determinar por qual porta de saída o pacote deve ser enviado. O algoritmo utilizado é denominado chaveamento XY adaptativo, apresentado na Figura 88.

```
01 if (xL > xT) dirx = WEST; else dirx = EAST;
02 if (yL > yT) diry = NORTH; else diry = SOUTH;
03 if (xL == xT && yL == yT) && free[LOCAL]==1 {
04     in[dir_incoming] = LOCAL;
05     out[LOCAL] = dir_incoming;
06     free[LOCAL] = 0;
07     ack_rot = TRUE;}
08 else if (xL != xT && free[dirx]==1) {
09     in[dir_incoming] = dirx;
10     out[dirx] = dir_incoming;
11     free[dirx] = 0;
12     ack_rot = TRUE;}
13 else if (yL != yT && free[diry]==1) {
14     in[dir_incoming] = diry;
15     out[diry] = dir_incoming;
16     free[diry] = 0;
17     ack_rot = TRUE;}
18 else
19     ack_rot = FALSE;
```

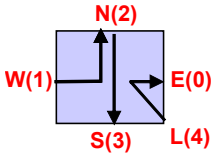
Figura 88 – Exemplo de código fonte em linguagem C do algoritmo de chaveamento XY adaptativo.

O algoritmo de chaveamento XY faz a comparação do endereço da chave atual com o endereço da chave destino do pacote (armazenado no primeiro *flit* do pacote). O pacote deve ser chaveado para a porta Local da chave quando o endereço xLyL[2] da chave atual for igual ao endereço xTyT[3] da chave destino do pacote. Caso contrário, é realizada, primeiramente, a comparação horizontal de endereços. A comparação horizontal determina se o pacote deve ser chaveado para o Leste (xL<xT), para o Oeste (xL>xT), ou se o mesmo já está horizontalmente alinhado à chave destino (xL=xT). Caso esta última condição seja verdadeira ou a porta horizontal escolhida esteja ocupada é realizada a comparação vertical que determina se o pacote deve ser chaveado para o Sul (yL<yT), para o Norte (yL>yT), ou se o mesmo já está verticalmente alinhado à chave destino (yL=yT). Caso esta última condição seja verdadeira ou a porta vertical escolhida esteja ocupada, é realizado o bloqueio dos *flits* do pacote em todas as chaves intermediárias até que a porta de saída escolhida seja liberada. Observa-se que o caminho pode variar, em função das chaves ocupadas na rede.

Quando o algoritmo de chaveamento interno retorna uma porta de saída livre, a conexão

[2] xLyL é o endereço da chave atual, onde xL é o endereço horizontal e yL o endereço vertical.
[3] xTyT é o endereço da chave destino dos dados, onde xT é o endereço horizontal e yT o endereço vertical.

entre a porta de entrada e a porta de saída é estabelecida e é preenchida a tabela de chaveamento. A tabela de chaveamento é composta pelos vetores *in*, *out* e *free*. O vetor *in* é indexado pela porta de entrada e preenchido com a porta de saída da conexão. O vetor *out* funciona de forma análoga, sendo indexado pela porta de saída e preenchido com a porta de entrada. O vetor *free* serve para alterar o estado da porta de saída que no momento encontra-se livre (1), passando para o estado de ocupado (0). Por exemplo, se a chave possuir os chaveamentos apresentados na Figura 89a os vetores serão preenchidos como ilustrado na Figura 89b.



	0 – E	1 – W	2 – N	3 – S	4 – L
Free	0	1	0	0	1
In	-	2	3	-	0
Out	4	-	1	2	-

Figura 89 – a) Chaveamento. b) Tabela de chaveamento.

Considere a porta Norte da Figura 89b. A porta de Saída Norte está ocupada (free = 0) e conectada na porta Oeste (out = 1). A porta de entrada Norte está conectada a porta de saída Sul (in = 3).

9.2 Contadores

A conexão entre as portas deve ser encerrada quando todos os *flits* do pacote forem enviados. Por este motivo, a chave possui cinco contadores, um para cada porta de entrada. Esses contadores estão implementados dentro do módulo Fila (Figura 90) e são inicializados quando o segundo *flit* do pacote, que contém o número de *flits* restantes do mesmo, chega na porta de entrada da conexão. Esses contadores são decrementados a cada *flit* enviado com sucesso. Quando o valor do contador chega a zero, a conexão é encerrada e o índice da porta de saída no vetor *free* é liberado. O funcionamento do contador é apresentado na Figura 91 e é descrito a seguir.

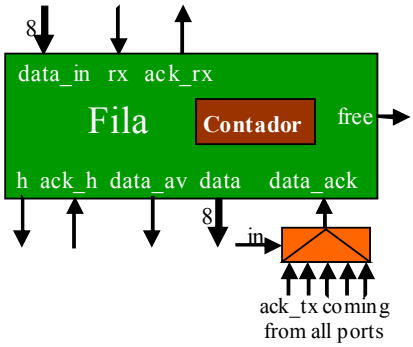


Figura 90 – Módulo Fila.

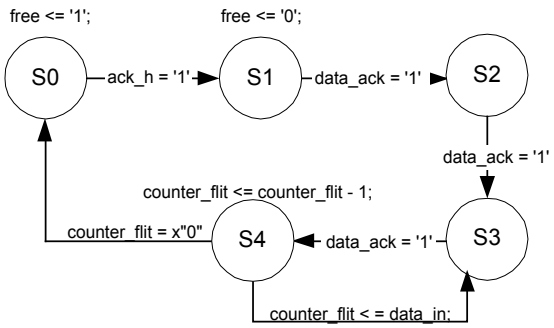


Figura 91 – Máquina de Estados do Contador.

- No estado S0 não há conexão estabelecida (*free*=’1’) e o contador possui o valor zero (*counter_flit* = x”00”). Quando chega o primeiro *flit* a máquina avança para o estado S1.
- A máquina de estados permanece em S1 até que o sinal *data_ack* (que indica que o *flit* foi transmitido) seja ativado, neste momento a máquina avança para o estado S2.
- No estado S2 o contador armazena o valor do segundo *flit* do pacote que se encontra no

topo da fila e avança para o estado S3 quando o sinal *data_ack* é ativado.

- No estado S3 a máquina espera que o sinal *data_ack* seja ativado, neste momento a máquina avança para o estado S4.
- O estado S4 é responsável por decrementar o valor do contador (*counter_flit*). Se o valor do contador for zero a conexão é encerrada e a máquina retorna ao estado S0, onde a conexão é encerrada. Caso contrário, a máquina retorna ao estado S3 aguardando o envio do próximo *flit* do corpo do pacote.

9.3 Arbitragem

A chave suporta até cinco conexões simultaneamente, mas apenas uma pode ser estabelecida a cada instante. Logo, existe a necessidade de um árbitro para determinar qual o pacote deve ser chaveado primeiro, quando mais de um chega à chave em um mesmo instante de tempo. O módulo que realiza esta arbitragem é apresentado na Figura 92.

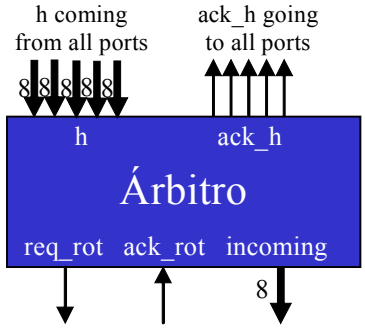


Figura 92 – Módulo do Árbitro.

A arbitragem dinâmica rotativa já utilizada no barramento foi adotada também na chave. Essa arbitragem permite o chaveamento do pacote da porta de entrada com maior prioridade. A prioridade de cada porta depende da última que obteve permissão de chaveamento. Por exemplo, se a porta de entrada Local (índice 4) foi a última a ter permissão de chaveamento, a porta de entrada Leste (índice 0) terá a maior prioridade seguida das portas de entrada Oeste, Norte, Sul e Local, que recebem prioridades decrescentes. Neste caso, forma se a porta de entrada Sul (índice 3) e Oeste (índice 1) solicitarem chaveamento simultaneamente, a porta de entrada Oeste terá sua solicitação atendida, porque tem maior prioridade. É importante observar que a prioridade das portas de entrada é variável e dependente da última porta de entrada a ter a permissão de chaveamento. Isto significa que todas as portas de entrada serão atendidas, mesmo as de menor prioridade.

O árbitro, após atender uma solicitação, aguarda 4 ciclos de relógio para que o algoritmo de chaveamento interno seja executado e somente após este período volta a atender solicitações. Se o algoritmo de chaveamento interno não consegue estabelecer uma conexão, a porta de entrada volta a solicitar chaveamento ao árbitro, porém com a menor prioridade.

O exemplo de código fonte em linguagem C para a seleção da porta a ter permissão de chaveamento é apresentado na Figura 93 e a máquina de estados é apresentada na Figura 94 com seu funcionamento detalhado a seguir.

```
01 if(h[EAST]==´1´ || h[WEST]==´1´ || h[NORTH]==´1´ || h[SOUTH]==´1´ || h[LOCAL]){
02        ask =´1´;
03        sel = prox;
04 }
05
06 switch(sel){
07        case EAST:
08               if(h[WEST]==´1´) prox=WEST;
09               else if(h[NORTH]==´1´) prox=NORTH;
10               else if(h[SOUTH]==´1´) prox=SOUTH;
11               else if(h[LOCAL]==´1´) prox=LOCAL;
12               else prox=EAST;
13               break;
14        case WEST:
15               if(h[NORTH]==´1´) prox=NORTH;
16               else if(h[SOUTH]==´1´) prox=SOUTH;
17               else if(h[LOCAL]==´1´) prox=LOCAL;
18               else if(h[EAST]==´1´) prox=EAST;
19               else prox= WEST;
20               break;
21        case NORTH:
22               if(h[SOUTH]==´1´) prox=SOUTH;
23               else if(h[LOCAL]==´1´) prox=LOCAL;
24               else if(h[EAST]==´1´) prox=EAST;
25               else if(H[WEST]==´1´) prox= WEST;
26               else prox=NORTH;
27               break;
28        case SOUTH:
29               if(h[LOCAL]==´1´) prox=LOCAL;
30               else if(h[EAST]==´1´) prox=EAST;
31               else if(H[WEST]==´1´) prox= WEST;
32               else if(h[NORTH]==´1´) prox=NORTH;
33               else prox=SOUTH;
34               break;
35        case LOCAL:
36               if(h[EAST]==´1´) prox=EAST;
37               else if(H[WEST]==´1´) prox= WEST;
38               else if(h[NORTH]==´1´) prox=NORTH;
39               else if(h[SOUTH]==´1´) prox=SOUTH;
40               else prox=LOCAL;
41               break;
42 }
```

Figura 93 – Exemplo de Código Fonte em Linguagem C para a seleção da porta de entrada que terá permissão de chaveamento.

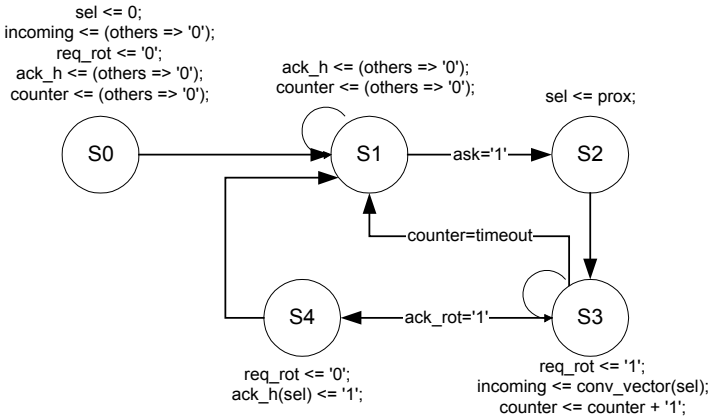


Figura 94 – Máquina de Estados do Árbitro.

- O estado S0 é o estado de inicialização da máquina. Este estado somente é atingido quando o sinal *reset* é ativado.
- O estado S1 é o estado de espera por requisição de chaveamento. Quando o árbitro recebe uma ou mais requisições o contador (responsável por encerrar o processo de

chaveamento quando o *timeout* é alcançado) é zerado e o sinal *ask* é ativado fazendo a máquina avançar para o estado S2.

- No estado S2 a porta de entrada que solicitou chaveamento é selecionada. Se houver mais de uma, aquela com maior prioridade é a selecionada.
- No estado S3 é requisitado o chaveamento (*req_rot*='1') informando a porta de entrada selecionada (*incoming=sel*). A máquina permanece no estado S3 incrementando o contador a cada ciclo de relógio até que a lógica de chaveamento informe que o chaveamento foi realizado (*ack_rot*='1') ou que o contador atinja o *timeout*. Se o contador atingir o *timeout* a máquina retorna ao estado S1, caso o chaveamento tenha sido realizado a máquina avança para o estado S4.
- No estado S4 o sinal *ack_h* é ativado indicando que o chaveamento foi realizado com sucesso. Do estado S4 a máquina retorna para o estado S1 onde aguarda novas requisições.

9.4 Bufferização da Mensagem

Quando o algoritmo de chaveamento resulta no bloqueio dos *flits* de um pacote, ocorre uma perda de desempenho em toda rede de interconexão, porque os *flits* são bloqueados não somente na chave atual, mas em todas as intermediárias. Por exemplo, se as chaves 00 e 01 transmitem ao mesmo tempo um pacote de 8 *flits* destinado à chave 21, o pacote que atingir primeiro o destino tem seu chaveamento autorizado e a conexão estabelecida, enquanto o outro deverá ser bloqueado e aguardar até que a conexão seja finalizada. Como a chave 01 está fisicamente mais próxima à chave 21 seu pacote será entregue primeiro e o pacote da chave 00 terá seus *flits* bloqueados nas chaves intermediárias, como é apresentado na Figura 95.

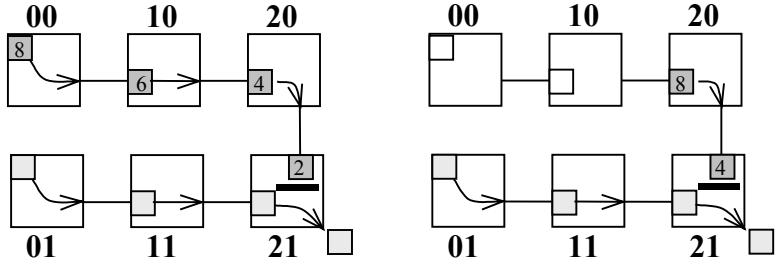


Figura 95 – a) Fila com 2 posições. b) Fila com 4 posições.
O valor presente em cada fila corresponde ao índice do último flit armazenado.

Para diminuir a perda de desempenho foi adicionada uma fila em cada porta de entrada da chave, reduzindo as chaves afetadas com o bloqueio dos *flits* de um pacote. É importante observar que quanto maior for o tamanho da fila menor será o número de chaves intermediárias afetadas. Como pode ser observado na Figura 95, o pacote com 8 *flits* bloqueia 4 chaves intermediárias quando a fila possui 2 posições. Enquanto, o mesmo pacote bloqueia apenas 2 chaves intermediárias quando a fila possui 4 posições.

As filas usadas no presente trabalho contêm 8 posições de 8 bits (*flit*) e funcionam como FIFOs circulares. Cada fila possui dois ponteiros: *first* e *last*. *First* aponta para a posição da fila onde se encontra o *flit* a ser consumido. *Last* aponta para a posição onde deve ser inserido o próximo *flit*. O controle de inserção de *flits* na fila é apresentado na Figura 96 e detalhado a

seguir.

```
tem_espaco <= '1' when not((first=x"0" and last=(TAMBUF-1)) or (first=last+1)) else '0';
ack_rx <= '1' when  (tem_espaco='1' and rx='1') else '0';
```

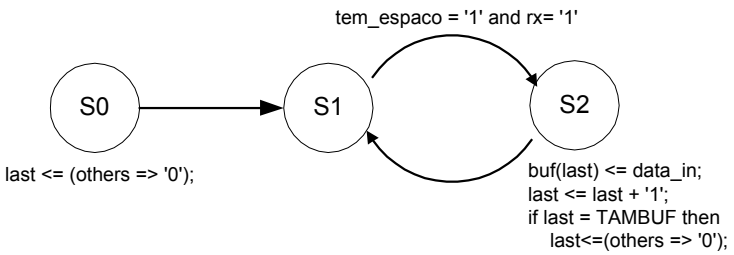


Figura 96 – Máquina de estados que controla a inserção de flits na fila.

- O estado S0 é o estado onde o ponteiro *last* é inicializado com o valor zero. Esse estado somente é atingido quando o sinal *reset* é ativado. A máquina de estados avança automaticamente do estado S0 para o estado S1.
- O estado S1 é o estado que espera pela recepção de *flits*. Quando o sinal *rx* é ativado indicando que existe um *flit* na porta de entrada é verificado se existe espaço na fila para armazená-lo. Se existir espaço na fila o sinal *ack_rx* é ativado, indicando que o *flit* foi recebido e a máquina avança para o estado S2. Caso contrário, a máquina permanece no estado S1 aguardando espaço para o armazenamento.
- No estado S2 o *flit* recebido é armazenado na posição apontada pelo ponteiro *last* e o mesmo é incrementado. Quando *last* atingir o tamanho da fila, ele recebe zero. Do estado S2 a máquina retorna ao estado S1 para aguardar novos *flits*.

A máquina de estados responsável por consumir *flits* na fila é visualizada na Figura 97 e seu funcionamento é apresentado a seguir.

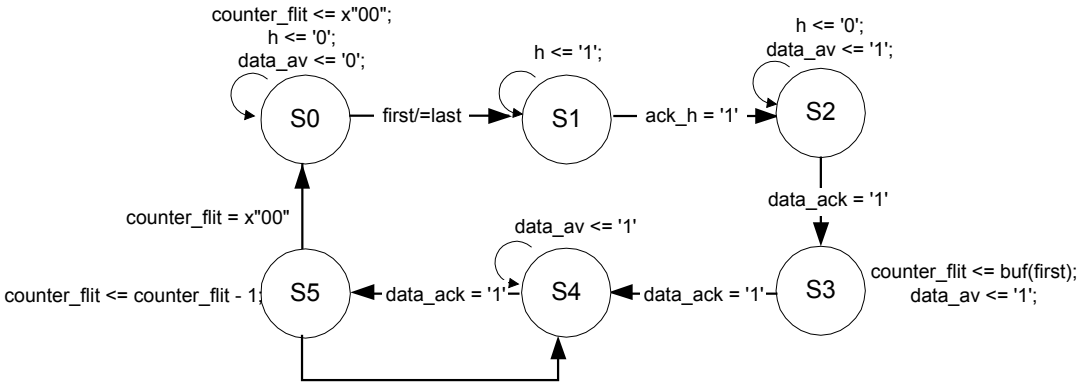


Figura 97 – Máquina de estados que controla a retirada de flits da Fila.

- No estado S0 os sinais *counter_flit* (contador de *flits* do corpo do pacote), *h* (que indica requisição de chaveamento) e *data_av* (que indica a existência de *flit* a ser transmitido) são inicializados com zero. Se existir algum *flit* na fila, ou seja, os ponteiros *first* e *last* apontarem para posições diferentes, a máquina de estados avança para o estado S1.
- No estado S1 é requisitado o chaveamento (*h*=’1’), porque o *flit* na posição apontada pelo ponteiro *first,* quando a máquina encontra-se nesse estado, é sempre o *header* do pacote. A máquina permanece no estado S1 até que receba a confirmação do chaveamento (*ack_h*=’1’) então avança para S2.

- Em S2 o sinal *h* recebe o valor zero e é indicado que existe um *flit* a ser transmitido (*data_av*='1'). A máquina de estados permanece em S2 até receber a confirmação da transmissão (*data_ack*='1') então avança para o estado S3.
- No estado S3 o ponteiro *first* aponta para o segundo *flit* do pacote. O segundo *flit* é armazenado no sinal *counter_flit* porque corresponde ao tamanho do corpo do pacote. O sinal *data_av* é ativado e é aguardada a ativação do sinal *data_ack* para a máquina avançar para o estado S4.
- O estado S4 é responsável por ativar o sinal *data_av* indicando que existe um *flit* do corpo do pacote a ser transmitido e aguardar a ativação do sinal *data_ack* ativo. Após a ativação de *data_ack* a máquina avança para S5.
- O estado S5 é responsável por decrementar o sinal *counter_flit*. Do estado S5 a máquina retorna para o estado S0 quando a transmissão do pacote foi finalizada (*counter_flit*=x"00"), ou retorna para o estado S4 para receber os outros *flits* do corpo do pacote.

9.5 Validação Funcional da Chave

A chave foi descrita em VHDL e validada por simulação funcional. A Figura 98 apresenta os blocos internos à chave e os sinais de duas portas (Local e Leste). A Figura 99 apresenta a simulação funcional dos sinais mais importantes da Figura 98. Os passos da simulação são os seguintes (a numeração abaixo tem correspondências na Figura 98 e na Figura 99):

1. A chave (xLyL=00) recebe um *flit* pela porta Local (índice 4, ver Figura 89), o sinal *rx* é ativado e o sinal *data_in* contém o *flit*.
2. O *flit* é armazenado na fila e o sinal *ack_rx* é ativado indicando que o *flit* foi recebido.
3. A porta Local solicita chaveamento ao árbitro ativando o sinal *h*.
4. O árbitro, após selecionar a porta que será chaveada, solicita o chaveamento à lógica de chaveamento enviando o *flit*, que é o endereço da chave destino (valor 10), e a porta de origem da requisição (sinal *incoming*, valor 4, representando a porta Local) .
5. O algoritmo de chaveamento XY é executado, a tabela de chaveamento é preenchidas e o sinal *ack_rot* é ativado, indicando que a conexão foi estabelecida.
6. O árbitro informa à fila que a conexão foi estabelecida e que o *flit* pode ser transmitido.
7. A chave ativa o sinal *tx* da porta de saída selecionada e coloca o *flit* em *data_out* desta mesma porta.
8. Uma vez o sinal *ack_tx* ativo o *flit* é removido da fila e o próximo *flit* armazenado pode ser tratado.
9. O segundo *flit* inicializa o contador indicando após quantos ciclos de relógio a conexão deve ser encerrada.

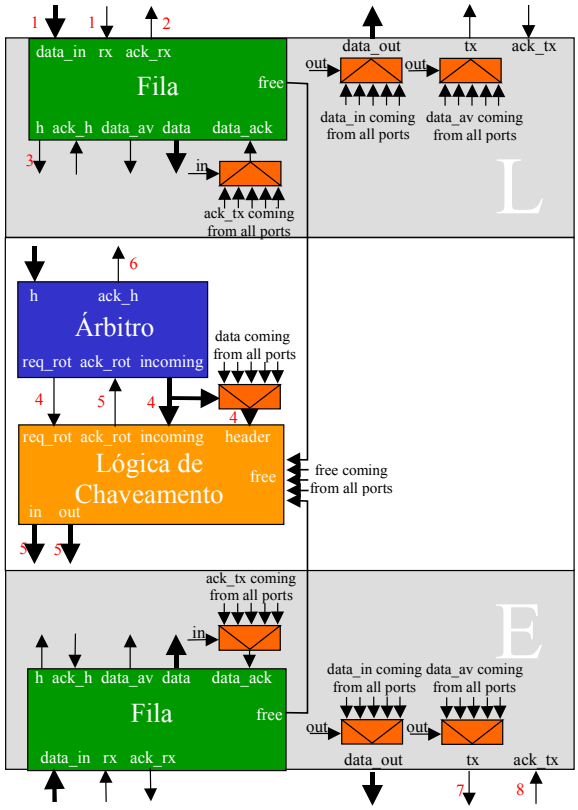


Figura 98 – Diagrama de blocos da chave, ilustrando apenas duas portas (os números indicam a seqüência de eventos da Figura 99).

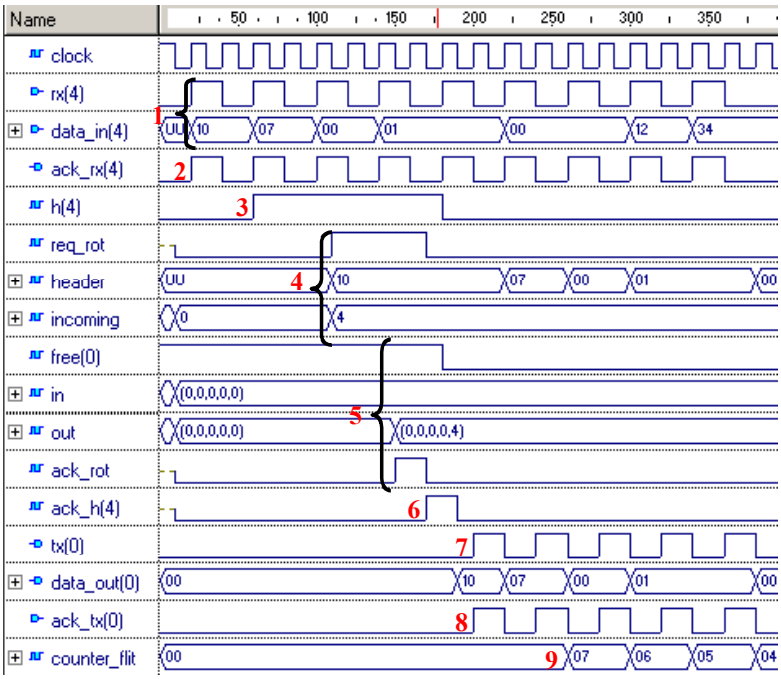


Figura 99 – Simulação de uma conexão entre a porta Local e a porta Leste.

Observa-se na Figura 100 o que ocorre na chave quando dois pacotes chegam simultaneamente.

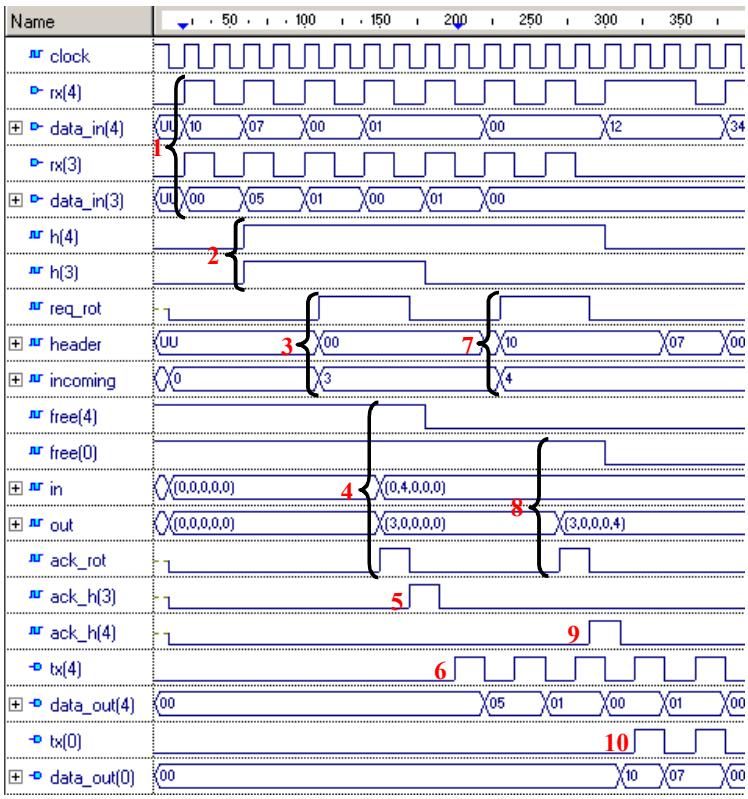


Figura 100 – Simulação de requisição de chaveamento simultânea.

1. A chave (xLyL=00) recebe dois pacotes simultaneamente, um pela porta Local e o outro pela porta Sul. O sinal *rx* de ambas as portas são ativados e o sinal *data_in* de cada uma delas contém o primeiro *flit* do pacote.
2. Cada porta solicita chaveamento ao árbitro, ativando o sinal *h*.
3. O árbitro seleciona a porta de maior prioridade. A porta Sul é selecionada porque, neste momento, assume-se que possua maior prioridade que a porta Local. O árbitro solicita o chaveamento à lógica de chaveamento enviando o *flit* da porta Sul, que é o endereço da chave destino (valor 00), e a porta de origem da requisição (sinal *incoming*, valor 3, representando a porta Sul) .
4. O algoritmo de chaveamento XY é executado, as tabelas de chaveamento são preenchidas e o sinal *ack_rot* é ativado indicando que a conexão foi estabelecida.
5. O árbitro informa à fila que a conexão foi estabelecida e que o *flit* pode ser transmitido.
6. A chave ativa o sinal *tx* da porta de saída selecionada e coloca o *flit* em *data_out* desta mesma porta.
7. Assim que o árbitro informa à fila que a conexão foi estabelecida, ele pode atender a solicitação de chaveamento da porta Local. O árbitro envia a lógica de chaveamento o *flit* da porta Local (valor 10) e informa a porta de origem da requisição (sinal *incoming*, valor 4, representando a porta Local) .
8. O algoritmo de chaveamento XY é executado, as tabelas de chaveamento são preenchidas com a nova conexão e o sinal *ack_rot* é ativado indicando que a conexão foi estabelecida.
9. O árbitro informa à fila que a conexão foi estabelecida e que o *flit* pode ser transmitido.

10. A chave ativa o sinal *tx* da porta de saída selecionada e coloca o *flit* em *data_out* desta mesma porta.

A validação funcional da chave também foi realizada a partir de um testbench, que lê pacotes de arquivos conectados às portas de entrada e escreve em arquivos conectados às saídas. Nessa validação foi verificado que todos os pacotes enviados foram corretamente recebidos, validando a arquitetura da chave.

10 Rede Intra-chip

Na topologia Mesh, utilizada neste trabalho, cada chave possui um número diferente de portas dependendo de sua posição na rede, conforme pode ser observado na Figura 101. Por exemplo, a chave central tem as cinco portas definidas no Capítulo 9. Entretanto, a chave superior esquerda possui apenas três portas: Leste, Sul e Local.

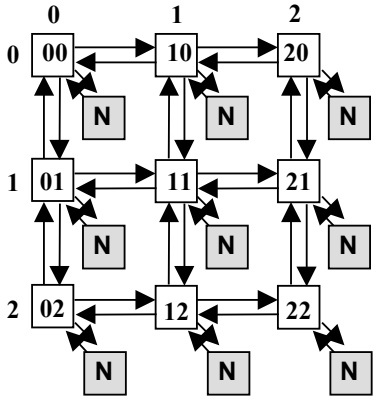


Figura 101 – Topologia Mesh 3x3. N representa o núcleo de hardware. Endereços das chaves indicam a posição XY na rede.

A parametrização do número de portas da chave permite a redução no consumo de área do dispositivo FPGA, parâmetro diretamente relacionado ao custo da implementação.

10.1 Validação Funcional da NoC com Topologia Mesh 2x2

A transmissão de pacotes na NoC proposta foi validada primeiramente por simulação. A Figura 102 ilustra a transmissão de um pacote da chave 00 para a chave 11 na rede de interconexão apresentada na Figura 101. Somente as interfaces de entrada e saída da chave 10 são apresentadas na simulação.

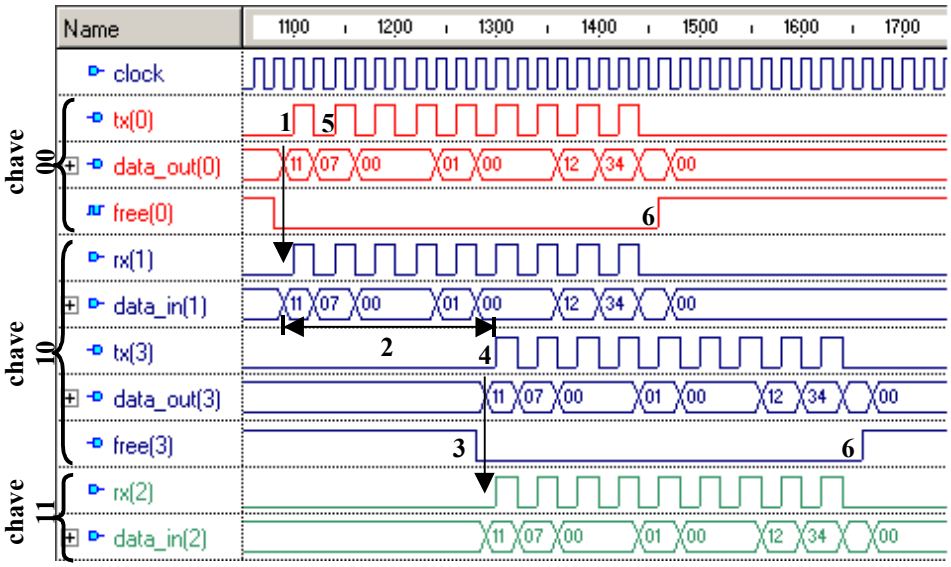


Figura 102 – Simulação da chave 00 enviando um pacote à chave 11 na topologia da Figura 101.

Os passos da simulação são os seguintes:

1. A chave com endereço 00 disponibiliza o primeiro *flit* do pacote (endereço da chave destino) no sinal *data_out* da porta Leste e ativa o sinal *tx* desta porta.
2. A chave 10 percebe o sinal *rx* de sua porta Oeste ativo e captura o *flit* no sinal *data_in*. A chave demora 10 ciclos de relógio entre a recepção do primeiro *flit* (*rx*='1') até transmissão para a próxima chave (*tx*='1'), sendo 2 ciclos para o armazenamento do *flit* na fila e 8 ciclos para o chaveamento da conexão, como mencionado na Seção 9.3. Os próximos *flits* são chaveados com latência de 2 ciclos de relógio.
3. A chave 10 coloca o sinal *free*(3) no estado *busy* indicando que o pacote foi chaveado para a porta Sul. Os sinais *free*(i) são elementos do vetor definido na Seção 9.1.
4. A chave 10 coloca o primeiro *flit* da mensagem no sinal *data_out* e ativa o sinal *tx* da porta Sul. A chave 11 detecta o sinal *rx* de sua porta Norte ativo. O *flit* é capturado no sinal *data_in* e a conexão entre a origem e o destino é estabelecida.
5. O segundo *flit* do pacote contém o número de *flits* do corpo do pacote.
6. Depois que todos os *flits* forem enviados, a conexão é finalizada e as entradas das tabelas *free* de cada chave intermediária retornam para o estado *free*.

A latência mínima em ciclos de relógio para transferir um pacote da origem para a chave destino é dada por:

$$latência = \left(\sum_{i=1}^{n} R_i\right) + P \times 2$$

onde: n é o número de chaves no caminho da comunicação (origem e destino inclusive), R_i é o tempo requerido pelo algoritmo de chaveamento para cada chave (no mínimo 8 ciclos de relógio), P é o tamanho do pacote. Este número é multiplicado por 2, porque cada *flit* requer 2 ciclos de relógio para ser enviado.

10.2 Integração do Sistema com Interconexão por NoC

Esta Seção apresenta a integração dos núcleos de hardware Serial (apresentado na Seção 4.3), Memória (apresentado na Seção 5.2) e Processador (apresentado na Seção 6.2) com a rede de interconexão NoC, como apresentado na Figura 103.

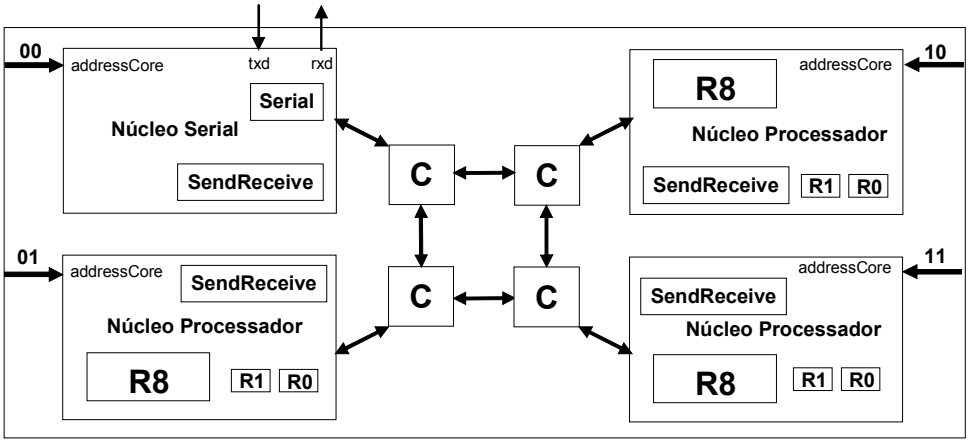


Figura 103 – Arquitetura Mesh 2x2 biprocessada.

Para realizar a integração entre os núcleos de hardware e a rede de interconexão NoC, foi necessário: (i) modificar o endereçamento dos núcleos de hardware; (ii) a forma como os pacotes são transmitidos na rede de interconexão; (iii) o módulo Send/Receive responsável pelo envio e recebimento destes pacotes; (iv) o núcleo Processador. Essas modificações são detalhadas a seguir.

10.2.1 Adaptação dos Endereços

Os endereços atribuídos aos núcleos de hardware devem ser alterados de forma a corresponderem ao endereço da chave a qual este está associado. Por exemplo, se o núcleo Serial estiver conectado a chave com endereço 00, este terá que possuir também o endereço 00.

10.2.2 Adaptação dos Pacotes

Os pacotes oriundos dos núcleos genéricos (Serial, Memória e Processador) permanecem semelhantes, sendo necessário apenas o acréscimo de um *flit* para que os mesmos possam ser transmitidos pela NoC. A Figura 104 apresenta o pacote de escrita na memória e a sua fragmentação em *flits*.

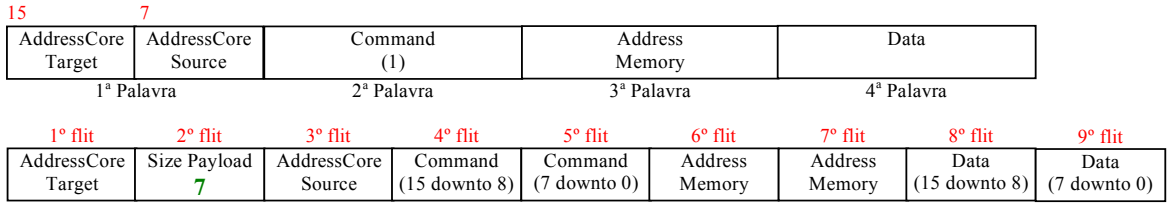


Figura 104 – Fragmentação do pacote de escrita na memória em *flits*.

O primeiro *flit* corresponde ao endereço do núcleo de hardware destino. O segundo *flit* é o tamanho do corpo do pacote, este valor é obtido por: $(N \times 2) - 1$, onde N é o número de palavras do pacote. A palavra do pacote possui 16 bits dando origem a dois *flits* (8 bits). O total de *flits* deve ser reduzido de 1, porque o endereço do núcleo destino faz parte do cabeçalho e não do corpo do pacote. Por exemplo, o pacote da Figura 104 possui 4 palavras resultando em um corpo de pacote de tamanho 7.

Da mesma forma que os pacotes foram fragmentados em *flits* para serem transmitidos pela rede, os *flits* devem ser montados dando origem ao pacote a ser entregue ao núcleo destino.

10.2.3 Adaptação do Módulo Send/Receive

O módulo Send/Receive necessita ser adaptado para comunicar-se com o protocolo utilizado pelas chaves. A interface entre o módulo e um núcleo genérico não sofre alteração, sendo apenas modificada a interface entre o módulo e o meio de comunicação. Antes, o módulo Send/Receive possuía os sinais: *busy*, *abort*, *request*, *grant, release*, *data_in*(16 bits) e *data_out*(16 bits) que faziam parte do protocolo de comunicação adotado pelo barramento. Agora esses sinais foram substituídos por: *tx*, *ack_tx*, *data_out* (8 bits), *rx*, *ack_rx* e *data_in* (8

bits), conforme o protocolo de comunicação da NoC (Seção 8.1.2).

Uma modificação significativa no módulo Send/Receive refere-se à confirmação de recebimento de pacote (ack). O protocolo implementado pelo barramento necessitava que o módulo Send/Receive enviasse uma confirmação de recebimento à origem do pacote, para garantir que o mesmo foi entregue com sucesso ao destino. No protocolo implementado na NoC isto não é necessário, porque a confirmação é realizada ponto-a-ponto através dos sinais *ack_tx* e *ack_rx*.

Ao módulo Send/Receive também foi atribuída a função de fragmentar e remontar os pacotes. Essa função consiste em dividir os pacotes recebidos dos núcleos em *flits*, que devem ser transmitidos pela rede de interconexão, e remontar os *flits* recebidos em pacotes, que devem ser entregues aos núcleos. Em redes de computadores segmentação e remontagem de mensagens são uma das principais tarefas de interfaces adaptadoras de rede. Neste sentido, o módulo Send/Receive assume o papel de adaptador de rede nos núcleos processadores da NoC.

10.2.4 Adaptação do Núcleo Processador

No sistema interconectado por barramento, o núcleo Processador possuía uma comunicação dedicada com sua memória local através de um barramento processador-memória. Neste esquema cada núcleo, processador e memória, possuía um endereço no sistema, como apresentado na Figura 105.

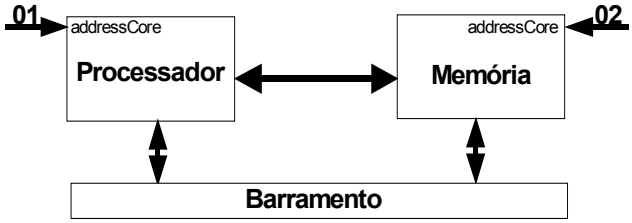


Figura 105 – Núcleo Processador e memória interconectado por barramento.

Na integração do sistema através da NoC percebeu-se que não seria interessante que um núcleo possuísse outra ligação além da ligação com a chave. Portanto, optou-se por integrar em um único núcleo o processador e sua memória local, como apresentado na Figura 106.

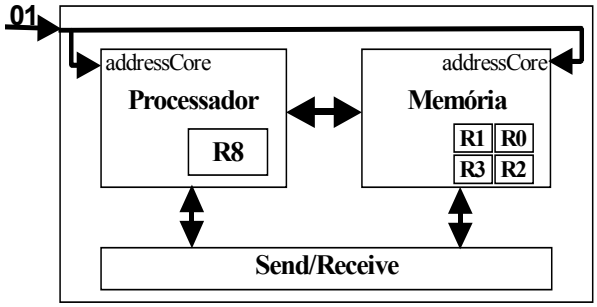


Figura 106 – Núcleo Processador integrando processador e memória local.

Embora com o mesmo endereço global, tanto o processador quanto à memória podem receber ou enviar mensagens para os demais núcleos da rede. A diferença é que este núcleo além de verificar o endereço destino, verifica também o comando da mensagem. É através do

comando que é identificado se a mensagem é destinada ao processador ou à memória.

10.3 Validação e Prototipação do Sistema Biprocessado com Interconexão por NoC

O sistema biprocessado interconectado por NoC, apresentado na Figura 103, foi descrito em VHDL e é composto por 15 pares entidade-arquitetura, implementando a hierarquia apresentada na Figura 107.

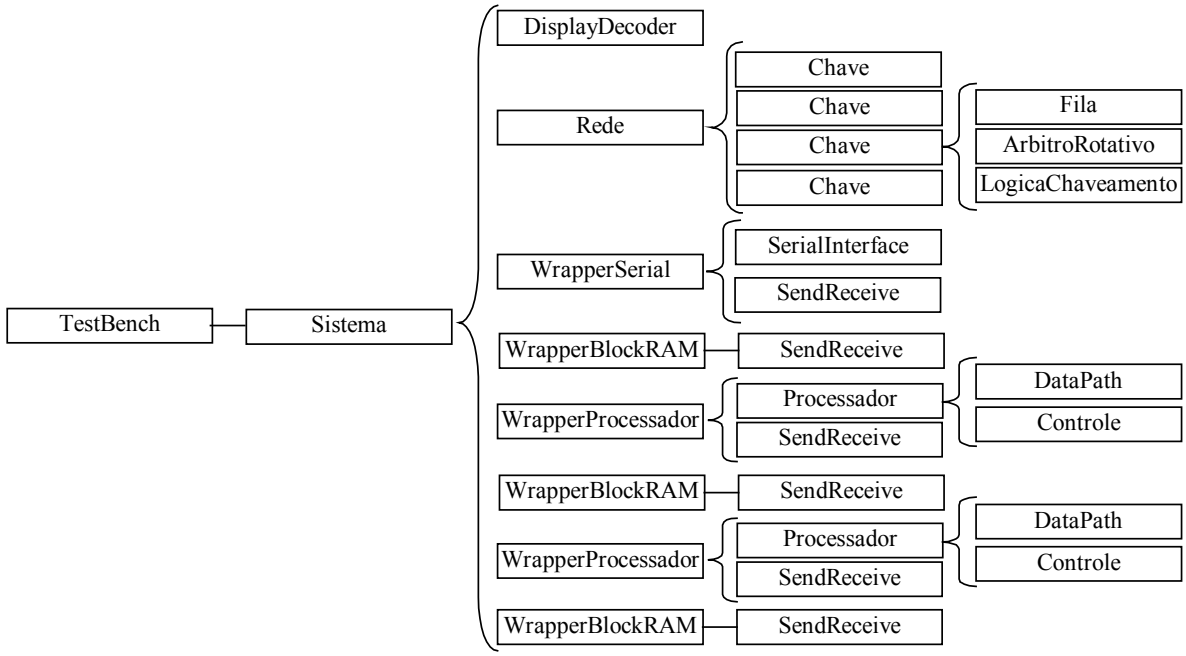


Figura 107 – Hierarquia do sistema biprocessado com conexão por NoC.

A rede de interconexão foi prototipada na plataforma de prototipação V2MB1000 da Memec Insight que contém um dispositivo FPGA XC2V1000-fg456 [XIL02]. Para validar a rede prototipada foram utilizados dois softwares. O primeiro software foi o ChipScope [XIL02], da Xilinx. O ChipScope permite a visualização dos sinais internos do FPGA selecionados pelo usuário em tempo de execução em hardware. A Figura 108 apresenta, com fins novamente ilustrativos, uma tela do ChipScope com alguns sinais presentes na simulação funcional (Figura 102). Com este software foi possível mostrar que a NoC proposta funciona em FPGAs exatamente como previsto por simulação.

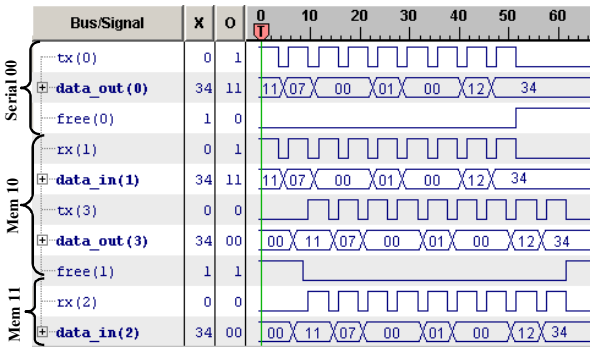


Figura 108 – Tela do software ChipScope, com dados obtidos diretamente da plataforma de prototipação.

O segundo software utilizado foi o Sistema Hospedeiro (Seção 4.1). Através dele são enviadas as instruções do algoritmo dos processadores P1 e P2 para as suas memórias locais e os dados do vetor a serem ordenados para a memória remota. A seguir são habilitados os processadores P1 e P2 que passam a executar as instruções armazenadas em suas memórias locais. Ao final da execução, os processadores enviam o número de ciclos gastos com a execução de seus algoritmos, como mostra a Figura 109 e a Figura 110. Esta operação finaliza o teste de execução do algoritmo paralelo no sistema biprocessado com interconexão por NoC.

Figura 109 – Tempo de execução do processador P1.

Figura 110 – Tempo de execução do processador P2.

O número de ciclos de relógio gastos pelo processador P2 para executar o algoritmo *Bubble Sort* sobre um vetor de 256 posições foi 8.146.085 ciclos e o número de ciclos de relógio gastos pelo processador P1 para executar o algoritmo *Bubble Sort* sobre um vetor também de 256 posições, esperar o comando *notify* e executar o algoritmo de *Merge* sobre as 512 posições foi de **8.809.259** ciclos, este número é menor que o apresentado pelo sistema biprocessado com interconexão por barramento (**12.203.008** ciclos). Portanto, o sistema biprocessado com interconexão por NoC obteve um desempenho de **28%** sobre o sistema biprocessado com interconexão por barramento.

A NoC com topologia Mesh 2x2, apresentada na Figura 103, foi sintetizada usando a ferramenta de síntese Leonardo Spectrum. A Tabela 5 apresenta os dados de área gerados por síntese, mostrando o uso dos recursos do FPGA. Aproximadamente, 50% dos recursos foram explorados, i.e. 500.000 portas equivalentes.

Tabela 5 – Dados de área da NoC 2x2 para XC2V1000 FPGA.

Recursos	Utilizado	Disponível	Utilizado/Total
Portas	513.107	1.000.000	51,31%
Slices	3.757	5.120	73,38%
LUTs	5.654	10.240	55,21%
Flip Flops	2.942	10.240	28,73%
BRAM	6	40	15,00%

A Tabela 6 detalha a área usada para os módulos da NoC para dois mapeamentos, FPGA e ASIC. A chave sozinha utiliza 458 LUTs para ser implementada, o que representa 4,47% da capacidade de LUTs em um dispositivo de 1 milhão de portas. A Tabela também mostra

dados de área para os núcleos de hardware: serial, memória e processador R8 (Seção 6.1). SR representa o módulo Send/Receive.

Tabela 6 – Relatório de área dos módulos da NoC para FPGA e ASIC (0,35μm CMOS).

	Mapeamento Virtex II			Mapeamento
	LUTs	FFs	BRAM	ASIC
Chave	458	172	-	2.930
SR	193	233	-	1.986
Serial	91	93	-	752
Serial + SR	590	563	-	4.587
RAM	-	-	2	-
R8	513	114	-	1.885
RAM + SR + R8	1.043	576	2	5.678

11 Avaliação Funcional da Rede Intra-Chip

Este Capítulo tem o objetivo de avaliar o desempenho da rede e a influência dos parâmetros: número de chaves intermediárias, tamanho das filas, número de pacotes a serem entregues e tamanho dos pacotes.

Com a finalidade de atingir estes objetivos foram realizados três estudos de caso sobre uma rede com topologia Mesh 5x5. Estes estudos foram feitos utilizando o ambiente de simulação ModelSim e a biblioteca FLI. A biblioteca FLI permite a comunicação das linguagens VHDL e C. Desta forma o teste da NoC (implementada em VHDL) foi feito a partir de leituras e escritas em arquivos implementadas em linguagem C. Cada arquivo de entrada do teste possui os pacotes que um nodo deve enviar para a rede. Um arquivo de saída do teste apresenta os pacotes que chegaram ao nodo ao qual o arquivo esta associado.

O estudo de caso I, apresentado na Seção 11.1, tem o objetivo de avaliar a entrega de pacotes de uma única chave origem à uma única chave destino, sem a ocorrência de colisões. A Seção 11.2 apresenta o estudo de caso II, tendo como objetivo avaliar a entrega de pacotes de duas chaves origem distintas à uma única chave destino, com a ocorrência de colisão na chave destino. A Seção 11.3 apresenta no estudo de caso III uma avaliação do comportamento da rede quando ocorrem transmissões simultâneas entre *S* chaves origem e *T* chaves destino.

11.1 Estudo de Caso I

No estudo de caso I busca-se identificar o quanto o número de chaves intermediárias e a capacidade de armazenamento das filas influenciam no tempo total de entrega dos pacotes na rede sem a existência de colisão.

Para realizar esse estudo foi conectado um arquivo de entrada à porta Local de uma chave qualquer da rede. Esse arquivo contém 50 pacotes de tamanho igual a 39 *flits* destinados à mesma chave. Na chave destino foi conectado a sua porta Local um arquivo de saída. Esta configuração de conexão é ilustrada na Figura 111.

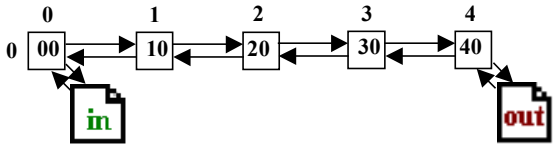


Figura 111 – Estudo de caso I com 5 chaves intermediárias.

O tempo ideal de transmissão para esse estudo de caso, sem considerar o tempo de chaveamento, é $NP \times (TP \times 2)$, onde NP é o número de pacotes a serem entregues, TP é o tamanho dos pacotes e 2 é o número de ciclos consumidos na entrega de cada *flit* do pacote. Ou seja, o tempo ideal é $50 \times (39 \times 2) = 3900$ ciclos.

O experimento considerou os seguintes parâmetros: número de chaves intermediárias e tamanho da fila. O número de chaves intermediárias variou no intervalo de 1 a 5 chaves e o tamanho da fila variou de 2 a 14 posições. Os resultados dos testes podem ser observados no Gráfico 1.

O Gráfico 1 mostra que o tempo de chaveamento permanece constante para filas com

dimensionamento igual ou superior a 6 posições. Este fato indica que filas com tamanho superior a 6 posições, para este caso, não são necessárias.

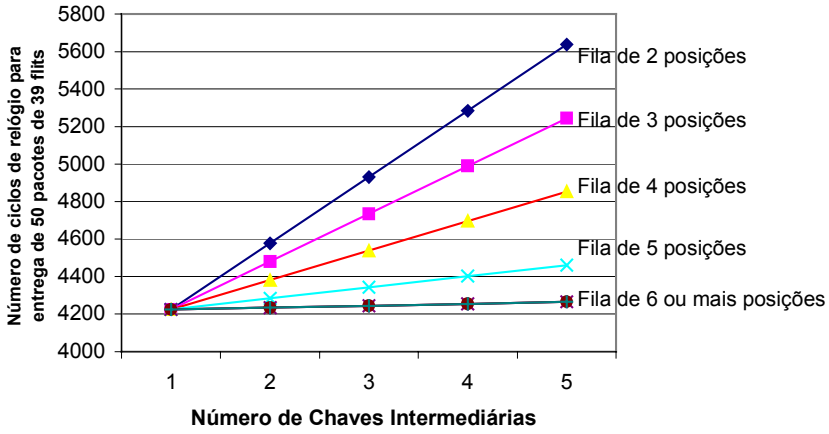


Gráfico 1 – Tempo total de entrega de 50 pacotes de 39 *flits* para o estudo de caso I.

A vantagem de se utilizar filas maiores está na diminuição da contenção dos *flits* na rede. Deve-se ter em mente durante o projeto da rede o compromisso entre latência e contenção. Filas pequenas aumentam a latência, como é mostrado a seguir, enquanto filas maiores permitem diminuir o número de *flits* que ficam parados na rede aguardando chaveamento.

Para mostrar porque o tamanho ideal da fila para o caso I é de 6 posições, apresenta-se na Figura 112 a ocupação das filas no decorrer do tempo, considerando 5 chaves intermediárias.

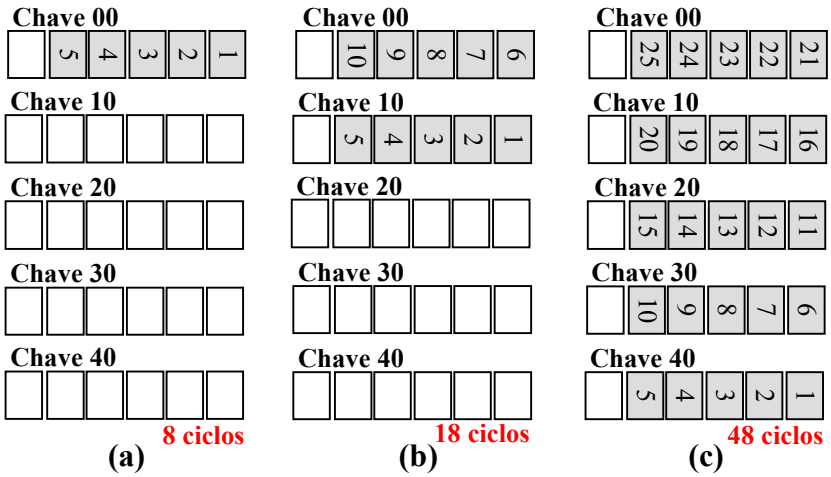


Figura 112 – Ocupação das filas durante a transmissão do 1º pacote entre a chave origem 00 e a chave destino 40, considerando fila com 6 posições. O valor presente em cada fila corresponde ao índice do flit armazenado.

A Figura 112a apresenta os primeiros *flits* do primeiro pacote chegando à chave 00. São gastos 2 ciclos de relógio para armazenar cada *flit* na fila e 8 ciclos para executar o chaveamento, totalizando 10 ciclos de relógio para que o *flit* de cabeçalho (1) chegue na chave seguinte. A chave 10, ao receber o primeiro *flit*, realiza o chaveamento em 8 ciclos, Figura 112b. Quando os primeiros *flits* do pacote chegam a chave destino, passaram-se 48 ciclos de relógio e as filas intermediárias encontram-se com a ocupação apresentada na Figura 112c. Após 50 ciclos de relógio, a conexão entre a chave 00 e o destino é estabelecida,

disponibilizando o primeiro *flit* do primeiro pacote para o destino. A partir deste momento são necessário 2 ciclos de relógio para entregar cada *flit* do primeiro pacote. O tempo total de entrega do pacote (TTP) pode ser calculado pela fórmula:

TTP = TTC + (TP * 2)

onde: TTC é o tempo total de chaveamento, TP é o tamanho do pacote e 2 é o número de ciclos gastos para a entrega de cada *flit* do pacote.

Baseado na fórmula apresentada acima, o tempo total (TT) para entregar os 50 pacotes deveria ser o tempo de entrega de um pacote (TTP) multiplicado pelo número de pacotes (NP) a serem entregues:

$$TT = TTP \times NP$$
$$TT = (50 + (39 \times 2)) \times NP$$
$$TT = 128 \times 50$$
$$TT = 6400$$

Porém, o tempo total representado no Gráfico 1 para uma fila de 6 posições é 4264. Esta diferença ocorre devido ao paralelismo entre a entrega dos *flits* de um pacote e o chaveamento do pacote seguinte. Esse paralelismo pode ser observado na Figura 113, sendo detalhado a seguir.

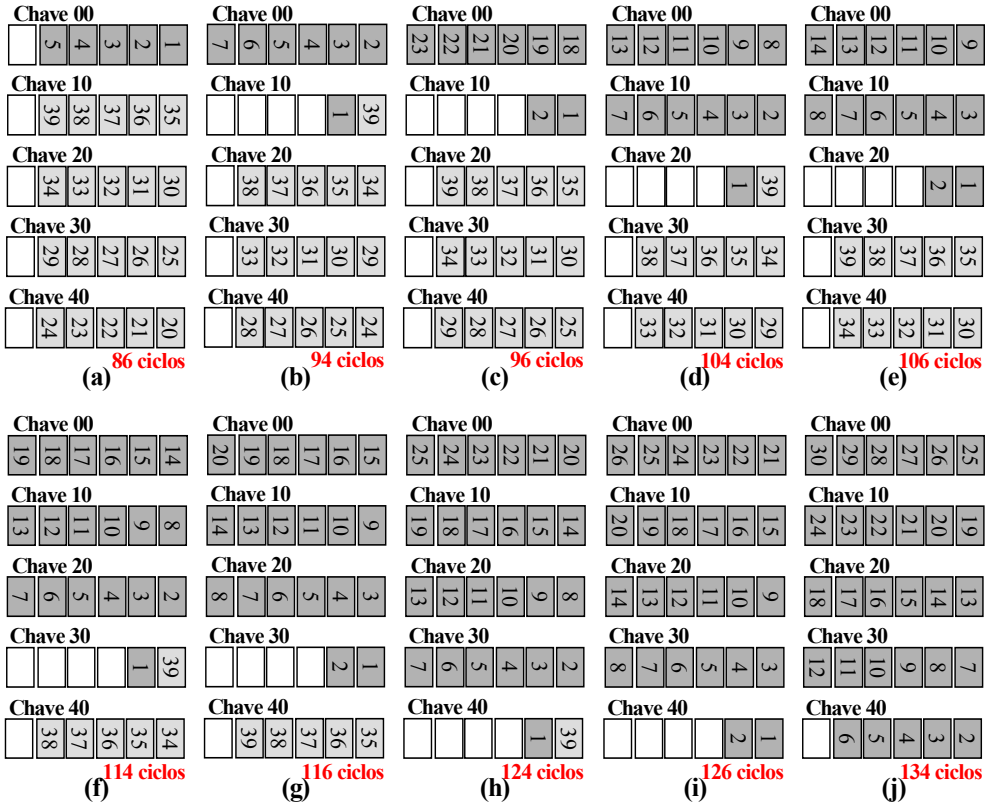


Figura 113 – Paralelismo entre a transmissão de um pacote e o seguinte.

No momento em que o primeiro *flit* do pacote 2 encontra-se no topo da chave 00, Figura 113a, inicia o paralelismo entre a entrega de *flits* do pacote 1 e o chaveamento do pacote 2. Os primeiros 8 ciclos de relógio em que ocorre o paralelismo são destinados ao chaveamento na chave 00. Os 2 ciclos de armazenamento do *flit* são desprezados na chave 00, porque o primeiro *flit* já estava armazenado. Nas figuras seguintes o leitor pode acompanhar o

comportamento em função do tempo das filas das chaves intermediárias. Quando o primeiro *flit* do pacote 2 chega a chave destino, Figura 113i, os *flits* do pacote 1 acabaram de ser entregues, totalizando 40 ciclos de relógio (126-86) com o paralelismo.

A partir do comportamento das filas apresentado, pode-se extrair a fórmula para o tempo de chegada de qualquer cabeçalho dos pacotes enviados de uma origem para um destino:

$$TC_N = TC_1 + ((TRP + (TP \times 2)) \times (N - 1))$$

onde: $\mathbf{TC_N}$ é o tempo de chegada do cabeçalho do pacote **N**, $\mathbf{TC_1}$ é o tempo de chegada do cabeçalho 1, **TRP** é o tempo de roteamento parcial, **TP** é o tamanho do pacote e **N** é o pacote que se deseja descobrir o tempo de chegada. Abaixo é demonstrada a aplicação da fórmula para o 50º pacote: $TC_{50} = 50 + ((8 + (39 \times 2)) \times 49) = 4264$.

Para se obter o tempo em que um dado pacote chega no destino aplica-se à fórmula:

$$TTP_N = TC_N + (NTF - 1) \times 2$$

onde: $\mathbf{TTP_N}$ é o tempo total do pacote **N**, $\mathbf{TC_N}$ é o tempo de chegada do cabeçalho do pacote **N** e **NTF** é o número total de *flits* do pacote. Abaixo é demonstrada a aplicação da fórmula para o 50º pacote: $TTP_{50} = TC_{50} + (20 - 1) \times 2 = 4302$.

A descrição acima ilustra que a rede opera como um pipeline, desde que as filas estejam corretamente dimensionadas.

A Figura 114 apresenta a ocupação das filas durante a transmissão do primeiro pacote da chave origem 00 à chave destino 40, considerando fila de 3 posições, ou seja, sub-dimensionadas para a presente rede.

Figura 114 – Ocupação das filas durante a transmissão do 1º pacote, considerando filas de 3 posições.

A Figura 114a apresenta os primeiros *flits* do primeiro pacote chegando à chave 00. São gastos 2 ciclos de relógio para armazenar um *flit* na fila e mais 8 ciclos para executar o chaveamento, totalizando 10 ciclos de relógio para a chegada do cabeçalho do primeiro pacote na segunda chave. Em paralelo ao chaveamento são armazenados 2 *flits* do pacote e não 4 *flits*, como ocorre quando a fila possui 6 posições. Esse mesmo comportamento é verificado com as demais chaves, até que o *flit* alcance a chave destino no 50º ciclo de relógio, Figura 114b. Portanto, são gastos exatamente os mesmos 50 ciclos de relógio para realizar o chaveamento do primeiro pacote, tanto para a fila com 6 posições quanto para a fila com 3 posições. Entretanto, existe uma grande diferença entre o tempo total da entrega dos 50

pacotes para as diferentes dimensões de fila. Essa diferença pode ser observada na Figura 115, onde é apresentado o paralelismo que ocorre em filas com 3 posições.

O paralelismo entre a entrega dos *flits* do pacote 1 e o chaveamento do pacote 2 inicia-se quando o primeiro *flit* do pacote 2 requisita roteamento na chave 00, Figura 115a. Os primeiros 8 ciclos de relógio, em que ocorre o paralelismo, são destinados ao chaveamento do pacote 2 na chave 00 e a entrega de 4 *flits* do pacote 1 ao destino, Figura 115b. Nos 8 ciclos seguintes é realizado o chaveamento do mesmo na chave 10. Simultaneamente, são entregues ao destino 4 *flits* do pacote 1, restando apenas 4 *flits* para serem entregues, Figura 115c. Para realizar a entrega desses *flits* são necessários 8 ciclos de relógio, Figura 115d. Esses ciclos são usados para realizar parte do chaveamento do pacote 2 na chave 20. O número de ciclos de relógio em que ocorre o paralelismo é igual a 126-102 = 24 ciclos de relógio, e não mais 40 ciclos como ocorre quando a fila possui 6 posições. Conclui-se então que a diferença entre o tempo total de entrega está vinculada ao tempo de paralelismo entre os pacotes.

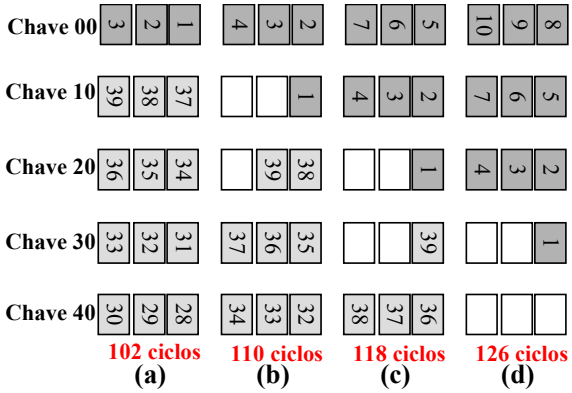


Figura 115 – Paralelismo entre a transmissão de pacotes, considerando fila com 3 posições.

O estudo de caso I permitiu mostrar que a dimensão mínima das filas deve ser proporcional ao tempo gasto pelo algoritmo de chaveamento. Com este dimensionamento, os *flits* são entregues como em um pipeline. Se o objetivo é reduzir contenção de *flits*, um dimensionamento superior ao mínimo é aceitável, lembrando-se sempre que isto terá forte custo em área.

11.2 Estudo de Caso II

No estudo de caso II buscou-se avaliar o tempo de entrega de pacotes quando ocorre uma colisão na rede. Para realizar esse estudo foi conectado um arquivo de entrada à porta Local de duas chaves quaisquer da rede e um arquivo de saída à outra chave distinta. O arquivo de entrada contém *n* pacotes de tamanho igual a 39 *flits* destinados à mesma chave. O arquivo de saída é escrito a medida em que os pacotes são entregues à chave destino. A Figura 116 apresenta a estrutura do estudo de caso II.

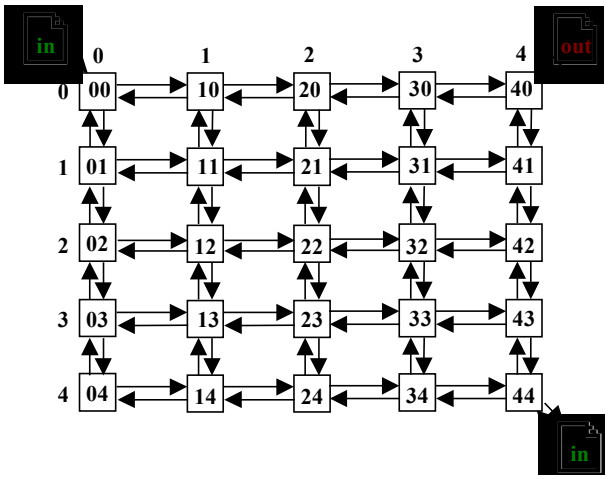


Figura 116 – Estudo de caso II, mostrando o posicionamento dos arquivos de entrada e saída.

O experimento foi realizado considerando os seguintes parâmetros: número de chaves intermediárias, tamanho da fila e número de pacotes a serem transmitidos. O número de chaves intermediárias variou no intervalo de 2 a 5 chaves; o tamanho da fila variou de 2 a 14 posições; e o número de pacotes a serem transmitidos foi 50, 100, 150 e 200 pacotes.

A Figura 117 apresenta a ocupação das filas durante a transmissão do primeiro pacote das chaves origem 00 e 44 à chave destino 40, considerando fila de 6 posições.

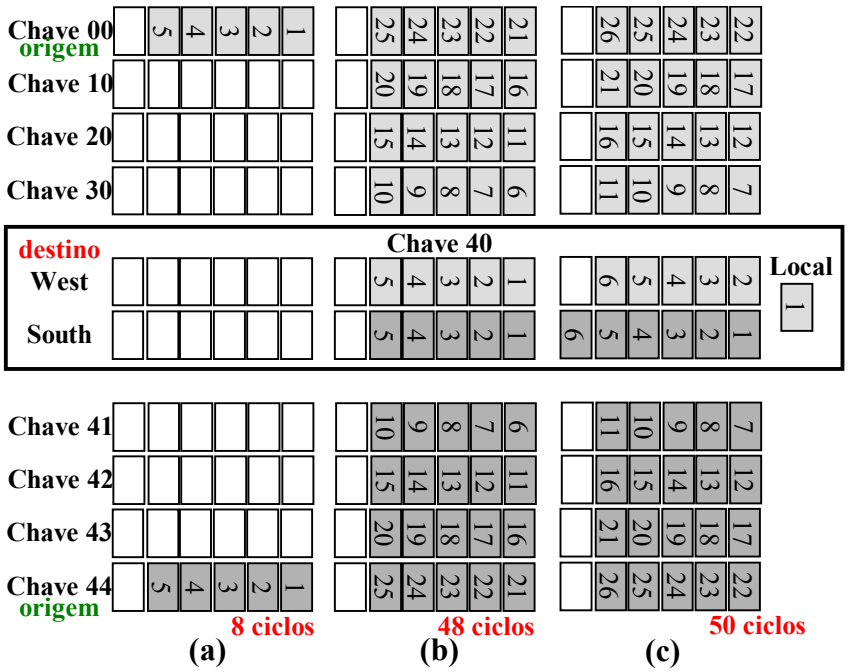


Figura 117 – Ocupação das filas durante a transmissão do 1º pacote das chaves origem 00 e 44 à chave destino 40, considerando fila de 6 posições.

A Figura 117a mostra as chaves origem 00 e 44 recebendo simultaneamente os primeiros *flits* do primeiro pacote armazenados nos arquivos de entrada. Os 8 ciclos de relógio iniciais são gastos no chaveamento na chave origem. São gastos 10 ciclos de relógio em todas as chaves intermediárias para que o cabeçalho do pacote alcance a chave seguinte. A Figura 117b apresenta o pacote da chave origem 00 chegando à chave destino pela porta Oeste e o pacote da chave origem 44 chegando à chave destino pela porta Sul. O primeiro *flit* desses pacotes é armazenado no 50º ciclo de relógio e solicitam simultaneamente o chaveamento.

Apenas um dos pacotes tem a solicitação atendida após 8 ciclos. A decisão de qual pacote será atendido cabe ao árbitro (Seção 9.3). Na Figura 117c observa-se que o pacote oriundo da chave 00 teve a solicitação atendida e começou a ser entregue pela porta Local, enquanto o pacote da chave origem 44 ficou bloqueado. Existirá o bloqueio até que todo o pacote da origem 00 seja entregue.

A Figura 118 apresenta a continuação da Figura 117. Ela ilustra o paralelismo existente entre os pacotes de uma mesma origem e de origens distintas.

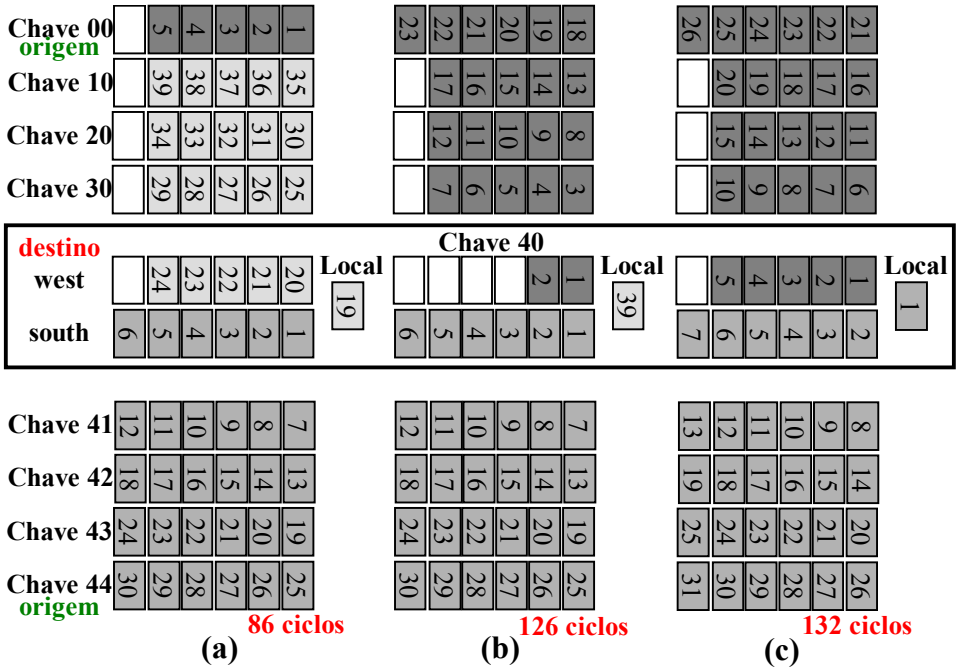


Figura 118 – Paralelismo na entrega de pacotes da mesma origem e de origens distintas.

A Figura 118a mostra o início do paralelismo de pacotes oriundos da mesma chave. Enquanto ocorre a entrega de um pacote é realizado o chaveamento do pacote seguinte. Esse paralelismo não ocorre somente na entrega do primeiro pacote, mas na entrega de todos os pacotes das duas origens, 00 e 44. A Figura 118b mostra a transmissão do último *flit* do pacote da chave origem 00. Neste momento, a conexão que existia entre a porta Oeste e a porta Local é finalizada e o pacote bloqueado pode realizar o chaveamento. Esse chaveamento gasta 4 ciclos de relógio, porque quando um pacote é bloqueado, ele fica aguardando apenas a conexão ser finalizada para ter sua solicitação de chaveamento atendida. Portanto, no 6º ciclo de relógio, após a entrega completa do pacote da chave 00, os *flits* do pacote da chave 44 começam a ser entregues, Figura 118c. Em paralelo, chega à porta Oeste o segundo pacote da chave origem 00, que ficará bloqueado até que o pacote da origem 44 seja totalmente entregue.

Na Figura 118b observa-se o mesmo paralelismo encontrado no estudo de caso I, onde no momento da entrega de um pacote, o próximo realiza o chaveamento. O paralelismo ocorre durante 40 ciclos do chaveamento, restando 10 ciclos de relógio para completar o chaveamento entre a chave origem e a chave destino. Contudo, este paralelismo não é relevante para o estudo de caso II. Como observado na Figura 118c, o segundo pacote da chave origem 00, embora esteja disponível para entrega ao destino, fica bloqueado aguardando a entrega do pacote da chave origem 44. Outra importante consideração sobre o caso II refere-se ao parâmetro tamanho da fila. O tempo total de entrega de pacotes neste caso

não sofre influência desse parâmetro, como é observado na Figura 119.

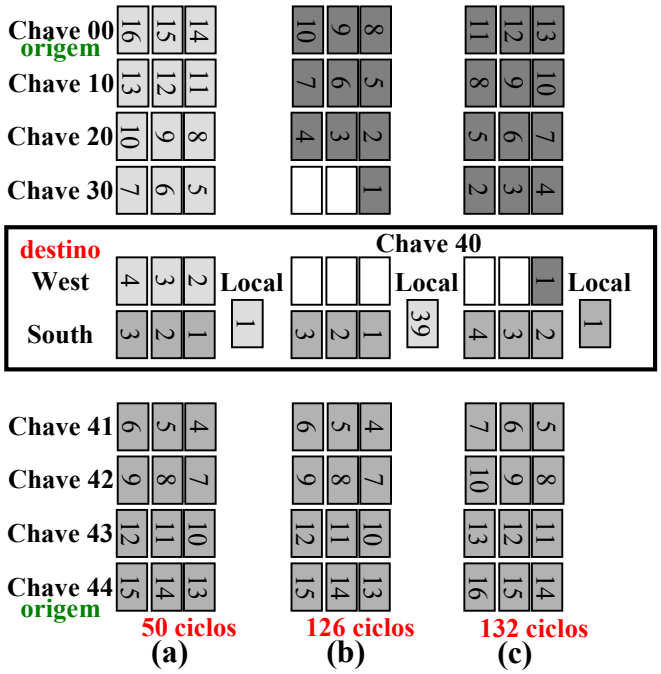


Figura 119 – Ocupação das filas durante a transmissão de pacotes das chaves origem 00 e 44 à chave destino 40, considerando tamanho de fila igual a 3 posições.

Observa-se na Figura 119a que o primeiro pacote da chave origem 00 é entregue ao destino no 50º ciclo de relógio, mesmo ciclo que foi entregue quando a fila possuía 6 posições, Figura 117c. A diferença entre as filas com 3 e 6 posições é que no momento em que o primeiro pacote oriundo da chave 00 termina de ser enviado, o segundo pacote ainda não encontra-se pronto para ser chaveado na chave 40, Figura 119b. No entanto, este fato não causa um atraso nas entregas de pacotes ao destino, pois ele ficará bloqueado até que todo o pacote 1 oriundo da origem 44 seja entregue, processo iniciado no 132º ciclo de relógio Figura 119c.

Os resultados obtidos com a execução do estudo de caso II são apresentados no Gráfico 2.

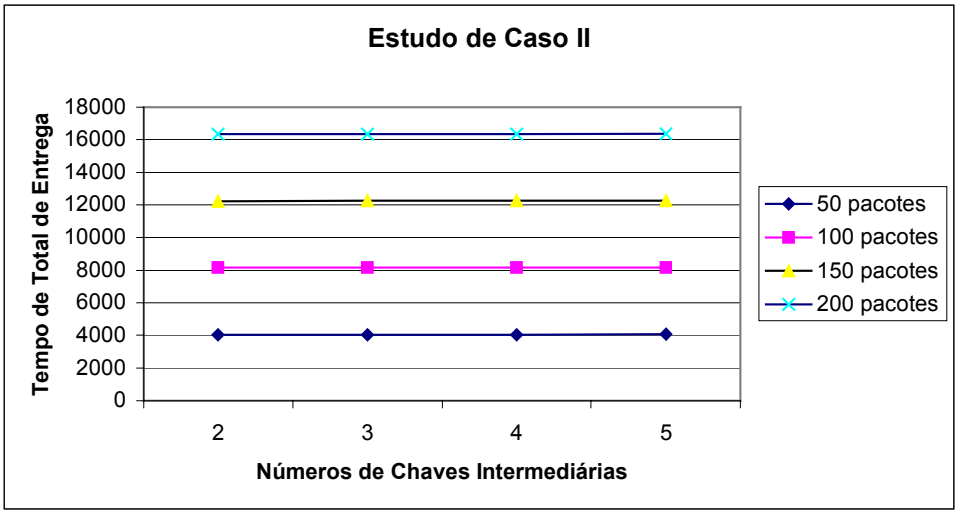


Gráfico 2 – Tempo de total de entrega do estudo de caso II.

Como explicado no parágrafo anterior, o parâmetro tamanho de fila não influencia no tempo total de entrega (TT) no caso II. Portanto, o Gráfico 2 representa os valores para qualquer tamanho de fila.

O tempo total de entrega cresce linearmente em 10 ciclos de relógio para cada chave intermediária inserida no caminho entre a origem e o destino. Esse crescimento é ilustrado no Gráfico 3, onde é apresentado em detalhe o tempo de total de entrega para 50 pacotes.

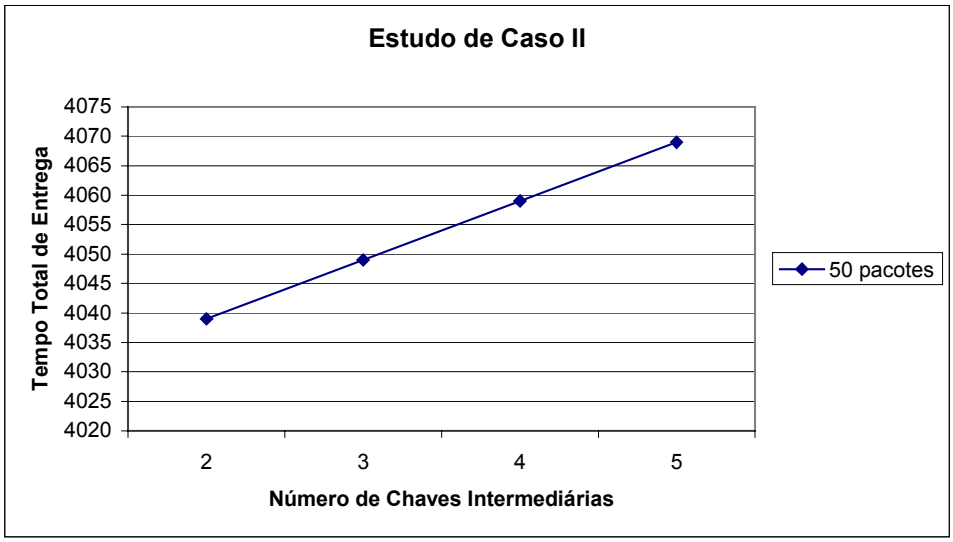


Gráfico 3 – Tempo total de entrega para 50 pacotes no estudo de caso II.

Esse estudo de caso permitiu observar que quando o gargalo encontra-se na chave destino o dimensionamento das filas não irá influenciar no tempo de entrega dos pacotes, devido à existência de bloqueio. Contudo, filas com número de posições maior apresentam um número menor de chaves intermediárias afetadas pelo bloqueio.

11.3 Estudo de Caso III

No estudo de caso III buscou-se avaliar o comportamento da rede quando ocorrem transmissões simultâneas entre *S* chaves origem e *T* chaves destino com a ocorrência de colisões. Para realizar esse estudo foi conectado um arquivo de entrada e um arquivo de saída à porta Local de todas as 25 chaves que formam a Mesh 5x5 apresentada na Figura 120. Cada arquivo de entrada contém 20 pacotes com tamanho total de 20 *flits*. Cada um desses pacotes possui um destino aleatório. Cada arquivo de saída é escrito toda a vez que é recebido um *flit* destinado à porta Local da chave.

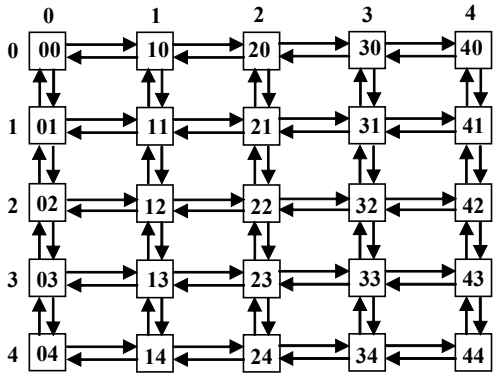


Figura 120 – Estudo de caso III.

O número de pacotes que são transmitidos nesse estudo de caso é igual ao número de chaves multiplicado pelo número de pacotes por chave, ou seja, 500 pacotes (25×20). O estudo de caso III foi aplicado para duas configurações de rede. A primeira com fila de 6 posições e a segunda com fila de 14 posições. Os resultados são apresentados na Tabela 7 e discutidos posteriormente.

Tabela 7 – Avaliação do estudo de caso para tráfego aleatório, com 25 fontes de geração de fluxo contínuo de dados. Três conjuntos de dados aleatórios utilizados.

	Fila com 6 posições				Fila com 14 posições			
	Tráfego 1	Tráfego 2	Tráfego 3	Média	Tráfego 1	Tráfego 2	Tráfego 3	Média
Média	183	183	172	**179**	263	265	248	**259**
Desvio Padrão	114	111	101	**109**	158	161	149	**156**
Mínimo	51	51	51	**51**	51	51	51	**51**
Máximo	815	848	653	**772**	888	1252	1055	**1065**
Tempo Total	3259	3267	3017	**3181**	2638	2947	2917	**2834**

A primeira linha da Tabela 7 mostra que o tempo médio para a entrega de um pacote quando a fila possui 6 posições de armazenamento (179 ciclos de relógio) é menor que o tempo médio quando a fila possui 14 posições (259 ciclos de relógio). A razão deste tempo maior para filas maiores deve-se ao fato que os *flits* devem percorrer um maior caminho, como ilustrado na Figura 121.

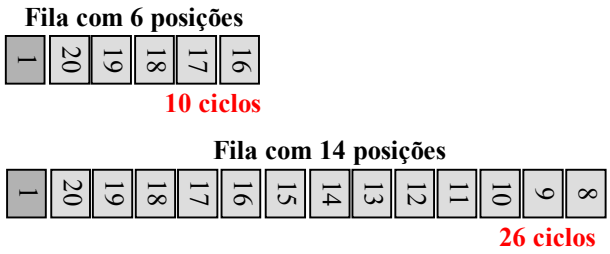


Figura 121 – Tempo de espera na fila até o chaveamento.

As linhas seguintes da Tabela apresentam uma análise da distribuição do tempo de envio dos 500 pacotes. Apesar de haver pacotes com elevado tempo de transmissão (máximo), o desvio padrão mostra que a maioria dos pacotes são transmitidos com tempo próximo à média. Isto deve-se ao paralelismo obtido pela rede. Em um sistema baseado em barramento, haveria uma serialização dos pacotes ocasionando tempos muito superiores aos obtidos com uma NoC.

A última linha da Tabela, tempo total, refere-se ao tempo total para a transmissão dos 500 pacotes. O tempo total para entregar os 500 pacotes a suas chaves destino é menor para a fila com dimensão igual a 14 posições (2834 ciclos de relógio) do que para fila com dimensão igual a 6 posições (3181 ciclos de relógio). Este resultado explica-se pelo fato que quanto maior a profundidade das filas, menor a contenção de dados na rede, diminuindo desta forma as colisões.

O estudo de caso III também permite comparar o tempo que uma rede gasta para transmitir 500 pacotes e o tempo que um barramento de largura 8 bits gastaria para realizar a transmissão do mesmo número de pacotes. A quantidade total de palavras que trafega nestas

arquiteturas é $500 \times 20 = 10000$ palavras. Sabendo que cada palavra no barramento consome 1 ciclo de relógio, e sem considerar o tempo gasto pela arbitragem, o barramento gastaria 10000 ciclos de relógio. Enquanto que a rede, considerando o tempo de chaveamento, consome 3181 ciclos de relógio para uma fila de 6 posições.

Esta análise encontra-se em andamento, sendo os resultados acima apresentados preliminares. Os valores da Tabela 7 pode sofrer alterações na apresentação final do trabalho.

Contudo, alguns pontos devem ser mencionados:

- O algoritmo de roteamento adaptativo apresenta possibilidade de "deadlock". Isto foi constatado durante o estudo de caso III, por este motivo a Tabela 7 foi obtida com um algoritmo de chaveamento XY puro, e encontra-se em fase de implementação um algoritmo parcialmente adaptativo.
- Como há mensagens que ficam muito tempo bloqueadas na rede, deve ser considerada a implementação da técnica *best effort* [JAN03] futuramente.

12 Conclusões e Trabalhos Futuros

Este trabalho apresentou o projeto, a implementação e avaliação de dois sistemas computacionais completos em dispositivos programáveis FPGA. A vantagem do emprego de SoCs para integrar diversos núcleos de processamento está na velocidade da troca de informações entre os núcleos, que é muito superior se comparada com a comunicação entre diferentes sistemas implementados em múltiplos circuitos integrados.

Para a implementação destes sistemas computacionais foram desenvolvidos os seguintes *núcleos comuns*: (*i*) *wrappers* de transmissão e recepção; (*ii*) memória; (*iii*) processador embarcado; (*iv*) interface serial compatível com padrão RS232. O sistema de interconexão por barramento é composto pelos núcleos comuns, pelo barramento propriamente dito e a política de arbitragem. O sistema de interconexão por rede é também composto pelos núcleos comuns e o conjunto de chaves que compõe a rede. Estes sistemas foram inicialmente simulados e posteriormente prototipados em FPGA, apresentando funcionamento correto. A Tabela 8 apresenta um resumo dos dados de área e tempo de execução dos sistemas computacionais implementados.

Tabela 8 – Quadro resumo com avaliação de área e desempenho para as diferentes topologias de interconexão desenvolvidas.

Sistema	Utilização da Plataforma	Portas Lógicas	Processador	Algoritmo	Nº de posições do vetor	Nº de ciclos
Monoprocessado Barramento	20%	292.646	P1	Bubble	512	16.032.562
Biprocessado Barramento	32%	441.573	P1	Bubble + Merge	256	12.203.008
			P2	Bubble	256	11.089.602
Biprocessado NoC	56%	513.107	P1	Bubble + Merge	256	**8.809.259**
			P2	Bubble	256	8.245.330

O sistema biprocessado interconectado por NoC obteve um desempenho 28% superior ao mesmo sistema interconectado por barramento, conforme a Tabela 8. Através deste estudo comprovou-se que uma estrutura de interconexão hierarquizada em camadas e que permite múltiplas comunicações por instante de tempo, como uma NoC, possui um desempenho superior ao barramento, mesmo para pequenas redes.

A análise funcional de tráfego também mostrou que a NoC (topologia Mesh 5x5) é potencialmente 3 vezes mais rápida que o barramento no envio de 500 pacotes entre núcleos aleatórios. Esse ganho de desempenho observado indica que essa topologia de interconexão é candidata às atuais demandas de desenvolvimento de SoCs.

O reuso de núcleos é possibilitado, entre outros aspectos, quando estes possuem o mesmo protocolo de comunicação adotado pelo meio de interconexão, seja barramento ou NoC. Em um sistema conectado por barramento, a inclusão de novos núcleos obriga ao desenvolvedor determinar o impacto deste no sistema. Um novo núcleo é mais uma carga adicionada ao barramento, degradando o desempenho. As NoCs, por proverem comunicação ponto-a-ponto, garantem escalabilidade na conexão de novos núcleos, não degradando o desempenho e privilegiando o reuso.

As principais contribuições do presente trabalho foram: (*i*) desenvolvimento de núcleos genéricos validados para reuso em outros projetos; (*ii*) comparação de área e desempenho entre barramento e NoC; (*iii*) desenvolvimento de uma chave facilmente parametrizável para

diferentes topologias, com baixa ocupação de área; (*iv*) demonstração da factibilidade de implementar redes intra-chip utilizando dispositivos programáveis.

Além disto, este trabalho proveu ao grupo de pesquisa que o sediou com uma forte experiência em arquitetura de computadores, redes de comunicação, projeto e implementação de sistemas computacionais embarcados compostos por hardware e software, além de ferramentas de CAD e arquitetura de FPGAs.

Como dificuldades encontradas durante este trabalho pode-se citar a visualização do tráfego de pacotes na topologia Mesh 5x5. Devido ao grande número de sinais desta topologia é difícil acompanhar os *flits* de um pacote da chave origem até o seu destino.

Outra dificuldade encontrada correspondeu a prototipação em FPGA. Mesmo havendo uma simulação funcional perfeita, podem ocorrer casos em que o sistema simplesmente não funciona no FPGA. Situações como não inicialização de máquinas de estado, não inicialização de registradores, atribuições onde não se consideram todos os casos possíveis (ocasionando o *warning latch inferred*), freqüência do relógio da placa superior à freqüência de operação do circuito, por exemplo, são problemas difíceis de detectar e ocasionam atrasos no desenvolvimento do projeto. A fase de prototipação é imprescindível a qualquer projeto de sistemas computacionais, pois prova que o sistema realmente funciona no hardware e não é apenas um conjunto de formas de onda em um simulador.

Encontra-se em andamento no GAPH (Grupo de Apoio ao Projeto de Hardware) a implementação de interfaces OCP para os núcleos implementados e sua comunicação com a NoC. Este trabalho teve também como objetivo facilitar o reuso da rede de interconexão e de núcleos de hardware.

A partir do trabalho desenvolvido pode-se enumerar diversas direções para pesquisas futuras, dentre as quais:

- Aprofundar os testes de avaliação funcional de tráfego da rede, considerando a geração de tráfego real.
- Desenvolver uma ferramenta para análise do tráfego da rede, permitindo ao projetista especificar parâmetros como a melhor largura dos flits, a profundidade ideal das filas, o melhor algoritmo de chaveamento, e até mesmo a topologia de rede.
- Desenvolver uma ferramenta para parametrização da rede, mais adequada para o problema a tratar.
- Pesquisar e implementar novos algoritmos de roteamento adaptativos que sejam imunes a deadlock e livelock.
- Estudar técnicas para garantir *throughput* em conexões com restrições de taxa de transferência (exemplo, *streaming* de áudio e vídeo). Esta técnica irá empregar as atuais chaves, acrescentadas de canais virtuais.
- Pesquisar técnicas de reconfiguração parcial e dinâmica de núcleos de hardware aplicada à NoC. A reconfiguração parcial e dinâmica permite, por exemplo, que núcleos de hardware tenham sua posição na rede alterada em tempo de execução, privilegiando a comunicação de núcleos com alta taxa de transferência para reduzir a contenção de *flits* da rede. A reconfiguração também pode ser aplicada para reduzir o consumo de área do sistema a partir da inserção e remoção de núcleos conforme a demanda de utilização.

- Uso de técnicas de planta baixa para o projeto físico da NoC. O correto posicionamento dos núcleos permite melhorar o desempenho e é imprescindível para habilitar reconfiguração parcial em dispositivos Virtex.
- Modelar funcionalmente as NoCs em SystemC TLM para validação de SoCs. Esta modelagem está em andamento, já obteve resultados preliminares e está sendo guiada pelas figuras de desempenho obtidas com a implementação VHDL. A motivação para a modelagem SystemC deve-se à quase impossibilidade de simular ciclo-a-ciclo um SoC composto por dezenas de núcleos.

13 Referências Bibliográficas

[ALT03] Altera. "*Excalibur Devices*". Capturado em: http://www.altera.com/products/devices/arm/arm-index.html (Jun. 2003)

[ARM02] ARM *"AMBA 2.0 Specification"*. Capturado em: http://www.arm.com/armtech/AMBA_Spec (Jul. 2002).

[BEN02] Benini, L. De Micheli,G. *"Networks on chips: a new SoC paradigm"*. Computer, Volume: 35(1), Jan. 2002, pp. 70-78.

[BER00] Bergamaschi, R. A.; Lee, W. R. *"Designing Systems-on-Chip Using Cores"*. In: Design Automation Conference, Los Angeles, California, USA, 2000, pp. 420-425.

[BRA02] Braun, F.; Lockwood, J.; Waldvogel, M. *"Protocol wrappers for layered network packet processing in reconfigurable hardware"*. IEEE Micro, Volume: 22(1), Jan.-Feb. 2002, pp. 66-74.

[CAL02] Calazans, N. L. V.; Moraes, F. G.; Marcon, C. A. M. "*Teaching Computer Organization and Architecture with Hands-On Experience*". In: Frontiers In Education Conference (FIE), 2002, pp. T2F15 – T2F20.

[CUL98] Culler, D.; Singh, J. P. *"Parallel Computer Architecture: a Hardware Software Approach"*. Los Altos, California: Morgan Kaufmann, 1998, 1100 p.

[DAL01] Dally, W.J.; Towles, B. *"Route packets, not wires: on-chip interconnection networks"*. In: Design Automation Conference, 2001, pp. 684-689.

[DAY83] Day, J.D.; Zimmermman, H. *"The OSI reference model"*. Proceedings of IEEE, Volume: 71, Dec. 1983, pp. 1334-1340.

[DEM95] De Micheli, G. "*Hardware/Software Codesign: Application Domains and Design Technologies*". Kluwer Academic Publishers, Italy, 1995.

[DUA97] Duato, J. et al. *"Interconnection Networks"*. Los Alamitos, California: IEEE Computer Society Press, 1997, 515 p.

[EXT03] ExtremeTech. "*Embedded Processors, Part One*". Capturado em: http://www.extremetech.com/print_article/0,3998,a=21014,00.asp (Mar. 2003)

[FEN81] Feng, T.-Y. *"A Survey of Interconnection Networks"*. Computer, Volume: 14, Dec. 1981, pp. 12-27.

[GAP02] GAPH. "*Hardware Support Design Group*". Capturado em: http://www.inf.pucrs.br/~gaph (Nov. 2002)

[GUE99] Guerrier. P.; Greiner. A.. "*A Scalable Architecture for System-On-Chip Interconnections***"**. In: SAME99, 1999.

[HAM97] Hammond, L.; Nayfeh, B. A.; Olukotun, K. "*A Single-Chip Multiprocessor*". Computer, Volume: 30(9), Sep. 1997, pp. 79-85.

[HWA93] Hwang, K. "Advanced Computer Architecture : Parallelism, Scalability, Programmability". New York : McGraw-Hill, 1993, pp. 213-256.

[IBM99] IBM, *"The RS/6000 SP High Performance Communication Network.IBM"*. Capturado em: http://www.rs6000.ibm.com/resource/technology/sp_sw1/spswp1.book_1.html (Aug. 1999).

[IYE02] Iyengar, V.; Chakrabarty, K.; Marinissen, E.J. *"Efficient Wrapper/TAM co-optimization for large SoCs"*. In: Design, Automation and Test in Europe, 2002, pp. 491-498.

[JAN03] Jantsch, A.; Tenhunen, H. "*Networks on Chip*". Dordrecht: Kluwer Academic Puublishers, 2003, 303 p.

[KOR02] Koranne, S. *"Design of reconfigurable access wrappers for embedded core based SoC test"*. In: International Symposium on Quality Electronic Design, 2002, pp. 106-111.

[KUM94] Kumar, V. et al. "*Introduction to Parallel Computing: Design and Analysis of Algorithms*". Redwood City, California : Benjamin/Cummings, 1994, 580 p.

[KUM02] Kumar, S.; Jantsch, A.; Soininen, J. P.; Fonsell, M.. "*A Network on Chip Architecture and Design Methodology*". In: Proceedings of the Computer Society Annual Symposium on VLSI. (ISVLSI'02), IEEE 2002.

[LAV95] Lavagno, L.; Sangionvanni-Vicentelli A.; Hsieh H. "*Embedded System Codesign: Synthesis and Verification*". Kluwer Academic Publishers, Italy, 1995.

[LEN90] Lenoski, D. et al. "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor". In: ACM ISCA, 17, 1990, pp. 148-159.

[MAD97] Madisetti, V. K., Shen L. "*Interface Design for Core-Based Systems*". IEEE Design & Test of Computers, Oct.-Dec. 1997, pp. 42-51.

[MEM03] Memec Insight. *"Virtex II V2MB1000 Development Board User´s Guide"*. Capturado em http://www.memec.com (Jun. 2003).

[MOH98] Mohapatra, P. "*Wormhole Routing Techniques for Directly Connected Multicomputer Systems*". ACM Computing Surveys, Volume: 30(3), Sep. 1998.

[MOR01] Moraes, F.; Mello, A.; Calazans, N. "*Ambiente de Desenvolvimento de Processador Embarcado para Aplicações de Codesign"*. In: SCR'2001 - Seminário de Computação Reconfigurável. 2001.

[OPE01] Opencores.org. *"Wishbone SoC Interconnection"*. Capturado em: http://www.opencores.com/press/pr_8jan2001.shtml (Aug. 2001).

[PAL02] Palma, J.C.S. "*Métodos de Distribuição e Conexão de IC Cores para Dispositivos Programáveis FPGA"*. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2002, 108 p.

[PAT96] Patterson, D.; Hennessy, J. L. "*Computer Architecture: A Quantitative Approach*". San Francisco, California : Morgan Kaufmann, 1996, 760 p.

[PAT97] Patt, Y. N., Patel, S. J., Evers, M., Friendly, D. H., Stark, J. "*One Billion Transistors, One Uniprocessors, One Chip"*. Computer, Volume: 30 (9), Sep. 1997, pp. 51-57.

[PAT98] Patterson, D.; Hennessy, J. L. "*Organização e Projeto de Computadores*". San Francisco, California : Morgan Kaufmann, 1998, 551 p.

[RIJ01] Rijpkema, E.; et al. "*Router Architecture for Networks on Silicon*". In: Workshop PROGRESS'2001, Netherlands, 2001.

[SGR01] Sgroi, M.; Sheets, M.; Mihal, A.; Keutzer, K.; Malik, S.; Rabaey, J.; Sangiovanni-Vincentelli, A. *"Addressing the system-on-a-chip interconnect woes through communication-based design"*. In: Design Automation Conference, 2001, pp. 667-672.

[SIL02] Silicore. *"The WISHBONE Service Center"*. Capturado em http://www.silicore.net/wishbone.htm/ (Jul. 2002).

[ZEF03] Zeferino, C. A. "*Redes-em-chip: Arquiteturas e Modelos para Avaliação de Área e Desempenho*". Tese de doutorado, Universidade Federal do Rio Grande do Sul - UFRGS/PPGC, Porto Alegre, RS, 2003.

[WIN01] Wingard, D. *"MicroNetwork-based integration for SoCs"*. In: Design Automation Conference, 2001, pp. 673-677

[XIL02] Xilinx. *"Virtex-II Platform FPGA Overview"*. Capturado em: http://www.xilinx.com (Jul. 2002).

14 ANEXO 1 – Código Assembly do *Bubble Sort*

```
.CODE
    INICIO:
        LDL R1,TAMV1            ;;R1 contem o tamanho do vetor V1
        LDH R1,TAMV1
        SUBI R1,#01H            ;;Subtrai 1 de TAMV1
        LDL R5,#01H             ;;R5 recebe #0001H
        LDH R5,#00H
    LOOP:
        ADD R1,R1,R0            ;;Aciona o flag de Zero quando R1 é zero
        JMPZD #FIMBUBBLE        ;;Salta quando o flag de Zero está ativo
        LDL R2,#V1              ;;R2 contem o endereço inicial do vetor V1
        LDH R2,#V1
        LDL R3,#V1              ;;R3 contem o endereço inicial do vetor V1 adicionado de 1
        LDH R3,#V1
        ADDI R3,#01H
        LDL R4,TAMV1            ;;R4 contem o tamanho do vetor V1
        LDH R4,TAMV1
        SUB R4,R4,R5            ;;Subtrai R4 de R5
    LOOPINTERNO:
        ADD R4,R4,R0            ;;Aciona o flag de Zero quando R4 é zero
        JMPZD #FIMLOOPINTERNO   ;;Salta quando o flag de Zero está ativo
        LD R10,R2,R0            ;;R10 recebe V1[R2]
        LD R11,R3,R0            ;;R11 recebe V2[R3]
        SUB R12,R11,R10         ;;Se R10 > R11 então TROCA
        JMPND #TROCA
    VOLTATROCA:
        ADDI R2,#01H            ;;Adiciona 1 em R2
        ADDI R3,#01H            ;;Adiciona 1 em R3
        SUBI R4,#01H            ;;Subtrai 1 de R4
        JMPD #LOOPINTERNO       ;;Salta incondicionalmente
    FIMLOOPINTERNO:
        SUBI R1,#01H            ;;Subtrai 1 de R1
        ADDI R5,#01H            ;;Adiciona 1 em R5
        JMPD #LOOP              ;;Salta incondicionalmente
    FIMBUBBLE:                  ;;Termina a execução do algoritmo de bubble
        HALT

    TROCA:
        ST R11,R2,R0            ;;Grava na posição de memória contida em R2 o conteúdo de R11
        ST R10,R3,R0            ;;Grava na posição de memória contida em R3 o conteúdo de R10
        JMPD #VOLTATROCA        ;;Salta incondicionalmente
.ENDCODE
.DATA
    TAMV1:   DB #0200H
    .ORG #0400H
    V1:      DB #00H
.ENDDATA
```

15 ANEXO 2 – Código Assembly do Algoritmo do Processador P1

```
.CODE
        INICIO:
                LDL R1,TAMV1            ;;R1 contem o tamanho do vetor V1
                LDH R1,TAMV1
                SUBI R1,#01H            ;;Subtrai 1 de TAMV1
                LDL R5,#01H             ;;R5 recebe #0001H
                LDH R5,#00H
        LOOP:
                ADD R1,R1,R0            ;;Aciona o flag de Zero quando R1 é zero
                JMPZD #FIMBUBBLE        ;;Salta quando o flag de Zero está ativo
                LDL R2,#V1              ;;R2 contem o endereço inicial do vetor V1
                LDH R2,#V1
                LDL R3,#V1              ;;R3 contem o endereço inicial do vetor V1 adicionado de 1
                LDH R3,#V1
                ADDI R3,#01H
                LDL R4,TAMV1            ;;R4 contem o tamanho do vetor V1
                LDH R4,TAMV1
                SUB R4,R4,R5            ;;Subtrai R4 de R5
        LOOPINTERNO:
                ADD R4,R4,R0            ;;Aciona o flag de Zero quando R4 é zero
                JMPZD #FIMLOOPINTERNO   ;;Salta quando o flag de Zero está ativo
                LD R10,R2,R0            ;;R10 recebe V1[R2]
                LD R11,R3,R0            ;;R11 recebe V2[R3]
                SUB R12,R11,R10         ;;Se R10 > R11 então TROCA
                JMPND #TROCA
        VOLTATROCA:
                ADDI R2,#01H            ;;Adiciona 1 em R2
                ADDI R3,#01H            ;;Adiciona 1 em R3
                SUBI R4,#01H            ;;Subtrai 1 de R4
                JMPD #LOOPINTERNO       ;;Salta incondicionalmente
        FIMLOOPINTERNO:
                SUBI R1,#01H            ;;Subtrai 1 de R1
                ADDI R5,#01H            ;;Adiciona 1 em R5
                JMPD #LOOP              ;;Salta incondicionalmente
        FIMBUBBLE:                      ;;Termina a execução do algoritmo de bubble
                LDL R15,#FEH            ;;Carrega o FFFE para R15 - FFFE indica WAIT
                LDH R15,#0FFH
                LDL R14,#05H            ;;Carrega 05 para R14 - 05 é o addressC de quem se esta esperando
        o NOTIFY
                LDH R14,#00H
                ST R14,R0,R15           ;;WAIT por core 05
        MERGE:                          ;;Começa a execução do algoritmo de Merge
                XOR R1,R0,R0            ;;R1 contem o índice i que percorre o vetor V1
                XOR R2,R0,R0            ;;R2 contem o índice j que percorre o vetor V2
                XOR R3,R0,R0            ;;R3 contem o índice k que percorre o vetor V3
                LDL R4,TAMV1            ;;R4 contem o tamanho do vetor V1 subtraído de 1
                LDH R4,TAMV1
                LDL R5,TAMV2            ;;R5 contem o tamanho do vetor V2 subtraído de 1
                LDH R5,TAMV2
                LDL R6,#V1              ;;R6 contem o endereço inicial do vetor V1
                LDH R6,#V1
                LDL R7,#V2              ;;R7 contem o endereço inicial do vetor V2
                LDH R7,#V2
                LDL R8,#V3              ;;R8 contem o endereço inicial do vetor V3
                LDH R8,#V3
        FOR:
                SUB R15,R4,R1           ;;Se i(R1) < TAMV1(R4) continua
                JMPZD #WHILEV1          ;;Senão vai para WHILEV1
                SUB R15,R5,R2           ;;Se j(R2) < TAMV2(R5) continua
```

```
                JMPZD #WHILEV1      ;;Senão vai para WHILEV1
                LD R9,R1,R6         ;;R9 recebe V1[endereço inicial(R6)+i(R1)]
                LD R10,R2,R7        ;;R10 recebe V2[endereço inicial(R7)+j(R2)]
                SUB R15,R9,R10      ;;Se R9<R10 então vai para MENORV1
                JMPND #MENORV1
                JMPD #MENORV2       ;;Senão vai para MENORV2
        RETORNOFOR:
                ADDI R3,#01H        ;;k++
                JMPD #FOR           ;;Retorna a FOR
        WHILEV1:
                SUB R15,R4,R1       ;;Se i(R1) < TAMV1(R4) continua
                JMPZD #WHILEV2      ;;Senao vai para WHILEV2
                LD R9,R1,R6         ;;R9 recebe V1[endereço inicial(R6)+i(R1)]
                ST R9,R3,R8         ;;V3[endereço inicial(R8) + k(R3)] recebe R9
                ADDI R1,#01H        ;;i++
                ADDI R3,#01H        ;;k++
                JMPD #WHILEV1       ;;Retorna a WHILEV1
        WHILEV2:
                SUB R15,R5,R2       ;;Se j(R2) < TAMV2(R5) continua
                JMPZD #FIM          ;;Senão vai para FIM
                LD R10,R2,R7        ;;R10 recebe V2[endereço inicial(R7)+j(R2)]
                ST R10,R3,R8        ;;V3[endereço inicial(R8) + k(R3)] recebe R10
                ADDI R2,#01H        ;;j++
                ADDI R3,#01H        ;;k++
                JMPD #WHILEV2       ;;Retorna a WHILEV2
        FIM:
                HALT

        TROCA:
                ST R11,R2,R0        ;;Grava na posição de memória contida em R2 o conteúdo de R11
                ST R10,R3,R0        ;;Grava na posição de memória contida em R3 o conteúdo de R10
                JMPD #VOLTATROCA    ;;Salta incondicionalmente

        MENORV1:
                ST R9,R3,R8         ;;V3[endereço inicial(R8) + k(R3)] recebe R9
                ADDI R1,#01H        ;;i++
                JMPD #RETORNOFOR    ;;Vai para RETORNOFOR
        MENORV2:
                ST R10,R3,R8        ;;V3[endereço inicial(R8) + k(R3)] recebe R10
                ADDI R2,#01H        ;;j++
                JMPD #RETORNOFOR    ;;Vai para RETORNOFOR
.ENDCODE
.DATA
        TAMV1:      DB #0FFH
        TAMV2:      DB #0FFH
        .ORG #0400H
        V1:         DB #00H
        .ORG #0500H
        V2:         DB #00H
        .ORG #0600H
        V3:         DB #00H
.ENDDATA
```

16 ANEXO 3 – Código Assembly do Algoritmo do Processador P2

```
.CODE
        INICIO:
                LDL R1,TAMV1            ;;Carrega R1 com o conteúdo de TAMV1
                LDH R1,TAMV1
                SUBI R1,#01H            ;;Subtrai 1 de R1
                LDL R5,#01H             ;;Carrega R5 com 1
                LDH R5,#00H
        LOOP:
                ADD R1,R1,R0            ;;Aciona o flag de Zero quando R1 é zero
                JMPZD #FIMBUBBLE        ;;Salta quando o flag de Zero está ativo
                LDL R2,#V1              ;;carrega R2 com o endereço do vetor V1
                LDH R2,#V1
                LDL R3,#V1              ;;Carrega R3 com o endereço de vetor V1
                LDH R3,#V1
                ADDI R3,#01H            ;;Adiciona 1 a R3
                LDL R4,TAMV1            ;;Carrega R4 com o conteúdo de TAMV1
                LDH R4,TAMV1
                SUB R4,R4,R5            ;;Subtrai R4 de R5
        LOOPINTERNO:
                ADD R4,R4,R0            ;;Aciona o flag de Zero quando R4 é zero
                JMPZD #FIMLOOPINTERNO   ;;Salta quando o flag de Zero está ativo
                LD R10,R2,R0            ;;R10 recebe V1[R2]
                LD R11,R3,R0            ;;R11 recebe V2[R3]
                SUB R12,R11,R10         ;;Se R10 > R11 então TROCA
                JMPND #TROCA
        VOLTATROCA:
                ADDI R2,#01H            ;;Adiciona 1 em R2
                ADDI R3,#01H            ;;Adiciona 1 em R3
                SUBI R4,#01H            ;;Subtrai 1 de R4
                JMPD #LOOPINTERNO       ;;Salta incondicionalmente
        FIMLOOPINTERNO:
                SUBI R1,#01H            ;;Subtrai 1 de R1
                ADDI R5,#01H            ;;Adiciona 1 em R5
                JMPD #LOOP              ;;Salta incondicionalmente
        FIMBUBBLE:                      ;;Termina a execução do algoritmo de bubble
                LDL R15,#FDH            ;;Carrega R15 com FFFD - FFFD indica notify
                LDH R15,#FFH
                LDL R14,#01H            ;;Carrega 01 em R14 - indica o núcleo destino do notify
                LDH R14,#00H
                ST R14,R0,R15           ;;Envia o notify ao núcleo 01
        FIM:
                HALT

        TROCA:
                ST R11,R2,R0            ;;Grava R11 na posição de memória armazenada em R2
                ST R10,R3,R0            ;;Grava R10 na posição de memória armazenada em R3
                JMPD #VOLTATROCA        ;;Salta incondicionalmente
.ENDCODE
.DATA
        TAMV1:          DB #0FFH
        .ORG #0500H
        V1:             DB #00H
.ENDDATA
```

17 ANEXO 4 – UCF (User Constraints File)

```
net clock          loc=a11;
net sw1            loc=d7;
net txd            loc=b7;
net rxd            loc=a7;

net display<0>     loc=f9;
net display<1>     loc=f11;
net display<2>     loc=f10;
net display<3>     loc=c10;
net display<4>     loc=c9;
net display<5>     loc=d10;
net display<6>     loc=d9;
net display<7>     loc=e7;
net display<8>     loc=b8;
net display<9>     loc=e8;
net display<10>    loc=e9;
net display<11>    loc=a8;
net display<12>    loc=e10;
net display<13>    loc=b9;

#BLOCKRAM 1
INST "NM0_WB_RAM0" LOC = "RAMB16_X0Y0";
INST "NM0_WB_RAM1" LOC = "RAMB16_X1Y0";

#BLOCKRAM 2
INST "NM1_WB_RAM0" LOC = "RAMB16_X2Y0";
INST "NM1_WB_RAM1" LOC = "RAMB16_X3Y0";

#BLOCKRAM 3
#descomentar as linhas abaixo para utilizar o sistema com três memória
#INST "NM2_WB_RAM0" LOC = "RAMB16_X0Y1";
#INST "NM2_WB_RAM1" LOC = "RAMB16_X1Y1";
```