Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Fábio Gonçalves Pessanha

Arquitetura de uma rede de interconexão com memória compartilhada baseada na topologia *crossbar*

Rio de Janeiro
2013

Fábio Gonçalves Pessanha

Arquitetura de uma rede de interconexão com memória compartilhada baseada na topologia *crossbar*

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Orientadora: Prof.ª Dr.ª Luiza de Macedo Mourelle
Coorientadora: Prof.ª Dr.ª Nadia Nedjah

Rio de Janeiro
2013

CATALOGAÇÃO NA FONTE
UERJ / REDE SIRIUS / BIBLIOTECA CTC / B

S586 Pessanha., Fábio Gonçalves
Arquitetura de uma rede de interconexão com memória compartilhada baseada na topologia *crossbar* /Fábio Gonçalves Pessanha. – 2013.
139f.

Orientadora: Luiza de Macedo Mourelle.
Coorientadora: Nadia Nedjah.
Dissertação (Mestrado) – Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia.

1. Engenharia Eletrônica - Dissertações. 2. Rede de Interconexão - Dissertações. I. Mourelle, Luiza de Macedo. II. Nedjah, Nadia. III. Universidade do Estado do Rio de Janeiro. Faculdade de Engenharia. IV. Título.

CDU 621.38:004.7

Assinatura Data

Fábio Gonçalves Pessanha

Arquitetura de uma rede de interconexão com memória compartilhada baseada na topologia *crossbar*

Dissertação apresentada, como requisito parcial para obtenção do título de Mestre, ao Programa de Pós-Graduação em Engenharia Eletrônica, da Universidade do Estado do Rio de Janeiro. Área de concentração: Sistemas Inteligentes e Automação.

Aprovado em: 22 de março de 2013.

Banca Examinadora:

Prof.ª Dr.ª Luiza de Macedo Mourelle (Orientadora)
Faculdade de Engenharia - UERJ

Prof.ª Dr.ª Nadia Nedjah (Coorientadora)
Faculdade de Engenharia - UERJ

Prof. Dr. Eugene Francis Vinod Rebello
Instituto de Computação - UFF

Prof. Dr. Wang Jiang Chau
Escola Politécnica - USP

Rio de Janeiro
2013

AGRADECIMENTOS

Agradeço, primeiramente, a Deus pela minha vida, dos meus familiares e dos meus amigos e, por sua infinita misericórdia, permitir que eu chegasse até aqui.

Agradeço à minha mãe, pela educação que me deu ao longo da minha vida, minha irmã pela força que me deu durante a minha vida e a minha namorada pela força e paciência durante todo o curso de mestrado.

Agradeço ao CNPq pelo apoio financeiro, durante o curso de mestrado.

Agradeço aos professores do PEL-UERJ pelas lições aprendidas ao longo do curso. Em especial às professoras Luiza de Macedo Mourelle e Nadia Nedjah pela orientação nesta dissertação e por terem me recebido como orientando.

Agradeço aos amigos do curso de mestrado, Luneque, Rafael, Rogério, Paulo, Leandro, Nicolas, Heloísa, pelas conversas e troca de ideias durante o curso e pela pizza das terças-feiras. Agradeço também a todos os amigos do trabalho e obrigado também pelos momentos de descontração.

”A mente que se abre a uma nova ideia jamais voltará a seu tamanho original.”

(Oliver Wendell Holmes)

RESUMO

Pessanha, Fábio Gonçalves. *Arquitetura de uma rede de interconexão com memória compartilhada baseada na topologia crossbar*. 2013. 139f. Dissertação (Mestrado em Engenharia Eletrônica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2013.

Multi-Processor System-on-Chip (MPSoC) possui vários processadores, em um único *chip*. Várias aplicações podem ser executadas de maneira paralela ou uma aplicação paralelizável pode ser particionada e alocada em cada processador, a fim de acelerar a sua execução. Um problema em MPSoCs é a comunicação entre os processadores, necessária para a execução destas aplicações. Neste trabalho, propomos uma arquitetura de rede de interconexão baseada na topologia *crossbar*, com memória compartilhada. Esta arquitetura é parametrizável, possuindo N processadores e N módulos de memórias. A troca de informação entre os processadores é feita via memória compartilhada. Neste tipo de implementação cada processador executa a sua aplicação em seu próprio módulo de memória. Através da rede, todos os processadores têm completo acesso a seus módulos de memória simultaneamente, permitindo que cada aplicação seja executada concorrentemente. Além disso, um processador pode acessar outros módulos de memória, sempre que necessite obter dados gerados por outro processador. A arquitetura proposta é modelada em VHDL e seu desempenho é analisado através da execução paralela de uma aplicação, em comparação à sua respectiva execução sequencial. A aplicação escolhida consiste na otimização de funções objetivo através do método de Otimização por Enxame de Partículas (*Particle Swarm Optimization* - PSO). Neste método, um enxame de partículas é distribuído igualmente entre os processadores da rede e, ao final de cada iteração, um processador acessa o módulo de memória de outro processador, a fim de obter a melhor posição encontrada pelo enxame alocado neste. A comunicação entre processadores é baseada em três estratégias: anel, vizinhança e *broadcast*. Essa aplicação foi escolhida por ser computacionalmente intensiva e, dessa forma, uma forte candidata a paralelização.

Palavras-chave: Redes de Interconexão. Memória Compartilhada. Redes *Crossbar*.

ABSTRACT

Multi-Processor System-on-Chip (MPSoC) has multiple processors in a single chip. Multiple applications can be executed in parallel or a parallelizable application can be partitioned and allocated to each processor in order to accelerate their execution. One problem in MPSoCs is the communication between the processors required to implement these applications. In this work, we propose the architecture of an interconnection network based on the crossbar topology, with shared memory. This architecture is parameterizable, having N processors and N memory modules. The exchange of information between processors is done via shared memory. In this type of implementation each processor executes its application stored in its own memory module. Through the network, all processors have complete access to their own memory modules simultaneously allowing each application to run concurrently. Moreover, a processor can access other memory modules, whenever it needs to retrieve data generated by another processor. The proposed architecture is modelled in VHDL and its performance is analysed by the execution of a parallel aplication, in comparison to its sequencial one. The chosen application consists of optimizing some objetive functions by using the Particle Swarm Optimization method. In this method, particles of a swarm are distributed among the processors and, at the end of each iteration, a processor accesses the memory module of another one in order to obtain the best position found in the swarm. The communication between processors is based on three strategies: ring, neighbourhood and broadcast. This application was chosen due to its computational intensive characteristic and, therefore, a strong candidate for parallelization.

Keywords: Interconnection Network. Shared Memory. Crossbar Switch.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

CISC	Complex Instruction Set Computer
DSP	Digital Signal Processor
FIFO	First-In-First-Out
IB	Input Buffer
IC	Input Controler
IFC	Input Flow Controller
IP	Intellectual Property
IRS	Input Read Switch
MCNoC	Multi-Cluster Network-on-Chip
MDN	MultiDirectional Network
MPSoC	MultiProcessor Systems-on-Chip
NoC	Network-on-Chip
NPoC	Network Processor on Chip
NUMA	Non-Uniform Memory Access
OC	Output Controller
ODS	Output Data Switch
ORS	Output Read Switch
PSO	Particle Swarm Optimization
RANoC	Router Architecture for NoC
RASoC	Router Architecture for SoC
RCS-NR	Reconfigurable Crossbar Switch for NoC Router
RISC	Reduced Instruction Set Computer
SoC	System-on-Chip
SoCIN	SoC Interconection Network

UMA	Uniform Memory Access
VHDL	Very high-speed integrated circuit Hardware Description Language
VLSI	Very Large Scale Integration

SUMÁRIO

INTRODUÇÃO

O AUMENTO na complexidade de circuitos integrados e a busca por maneiras eficazes de diminuir o tempo e o custo de projeto levaram a indústria eletrônica, a partir da segunda metade do século passado, a aplicar, de maneira cada vez maior, o uso de microprocessadores em seus projetos. A organização deste em unidade lógica e aritmética, unidade de controle e memória o tornou flexível para todo o tipo de aplicação em eletrônica. Através de um conjunto de comandos básicos armazenados em uma memória externa (*software*) e acessos a esta memória, pode-se controlar as saídas de diferentes tipos de dispositivos. Nos anos seguintes, presenciamos o aumento da capacidade interna do circuito integrado onde obtivemos uma série de arquiteturas baseadas neste tipo de implementação, como arquiteturas CISCs (*Complex Instruction Set Computer*), RISCs (*Reduced Instruction Set Computer*), DSPs (*Digital Signal Processor*) e ASICs (*Application Specific Integrated Circuit*) (SCHLETT, 1998). Entretanto a capacidade interna do circuito continuou aumentando, o que permitiu que não só processadores, mas várias unidades funcionais fossem colocadas dentro de um *chip*, surgindo o conceito de SoCs (System-on-Chip) (IVANOV; MICHELI, 2005).

SoCs são sistemas completos encapsulados em um único *chip*, possuindo todos os componentes necessários para um projeto de *hardware* como processador, memória e periféricos. Entretanto existe a necessidade destes componentes se comunicarem. Uma forma de comunicação entre componentes do sistema é utilizando canais ponto-a-ponto, onde os componentes se comunicam através de canais dedicados. Uma vantagem destes canais é o maior desempenho, pois para cada comunicação já existe previamente um canal estabelecido. Uma desvantagem deste tipo de implementação é o custo, pois cada arquitetura necessita de um projeto específico. O uso de barramentos é uma alternativa, pois apresenta baixo custo uma vez que este pode ser utilizado em vários projetos, entretanto o tempo de arbitragem cresce com o acréscimo de processadores (BJERREGAARD; MAHADEVAN, 2006). O uso de SoCs traz vantagens como a diminuição da área e do tempo

de projeto, além de aumentar o desempenho, já que esta arquitetura geralmente possui blocos dedicados para realizar diferentes funções (YABARRENA, 2011).

Uma evolução dos SoCs são os chamados MPSoC (*MultiProcessor Systems-on-Chip*), que, além de memória e periféricos, possuem vários processadores implementados em um único *chip* com o objetivo de atingir maior poder computacional. MPSoCs, atualmente, têm surgido como uma alternativa viável para a maioria da aplicações envolvendo multimídia, comunicações, processamento de sinais, entre outras (WOLF; JERRAYA; MARTIN, 2008). Estas aplicações exigem grande quantidade de computação e algoritmos complexos que não podem ser suportados por um *hardware* simples. Além disso, muitas destas aplicações requererem o uso de baterias o que exige também que estes sistemas sejam eficientes do ponto de vista energético (WOLF, 2004).

Em MPSoCs também deve existir a definição de como os processadores irão se comunicar. A definição de como os núcleos de processamento se comunicarão é a primeira etapa do projeto de MPSoCs. O meio de interconexão mais tradicional é o barramento, porém não são escaláveis (GUERRIER; GREINER, 2000). O barramento é compartilhado por todos os componentes existentes na arquitetura e apenas um componente pode usá-lo em um determinado intervalo de tempo, gerando gargalo na comunicação (BAINES, 1995). Uma alternativa para conectar os núcleos em MPSoCs são as denominadas NoC(*Network-on-Chip*) (BENINI; MICHELI, 2002). Estas redes de conexão são escaláveis, entretanto os projetistas necessitam adquirir novos conceitos, devido à não trivialidade desta rede, o que aumenta o tempo necessário para o projeto (BJERREGAARD; MAHADEVAN, 2006). Uma outra maneira de conectar núcleos de processamento, que será investigada neste trabalho, é através da rede *crossbar* (TAMIR; CHI, 1993) (FRANKLIN, 1981). Estas redes possuem alto desempenho, pois permitem conexões simultâneas entre todas as entradas e saídas.

O objetivo desta dissertação é desenvolver uma arquitetura de uma rede de interconexão com memória compartilhada, baseada na topologia *crossbar*, para interligar processadores a módulos de memória. A troca de informações entre os núcleos será realizada através de memória compartilhada. Com a arquitetura funcionalmente validada executaremos uma aplicação paralela, a fim de analisarmos o desempenho oferecido pela rede, em comparação com a respectiva execução sequencial. A escolha desta rede de interconexão é motivada pelo fato de termos poucos trabalhos em MPSoCs utilizando redes *crossbar* para interconectar núcleos de processamento diretamente a módulos de

memória. Atualmente nos trabalhos relacionados ao tema tem-se o uso de redes de interconexão utilizando troca de mensagem, onde cada núcleo de processamento possui sua própria memória.

A arquitetura proposta é escalável com N processadores e uma memória compartilhada dividida em N módulos, onde cada faixa de endereço define um módulo de memória. Cada módulo de memória contém a parte da aplicação que será executada por um processador. Entretanto qualquer processador pode acessar qualquer módulo de memória. Esta arquitetura é modelada em VHDL (*Very high-speed integrated circuit Hardware Description Language*) (NAVABI, 1997) e o modelo é funcionalmente validado através de simulação utilizando a ferramenta ModelSim (MODEL, 2012). Com a arquitetura operacional executamos uma aplicação paralela que é distribuída entre os processadores da rede. A avaliação do desempenho da arquitetura será feita em termos do tempo de execução, comparando-se a execução sequencial e a paralela, para diferentes números de processadores. Para atingir os objetivos acima expostos esta dissertação está estruturada em quatro capítulos e dois apêndices

No capítulo 1, apresentaremos os conceitos de sistemas embutidos, sistemas embutidos multiprocessados e redes de interconexão. Veremos a classificação das redes de interconexão e apresentaremos também os trabalhos relacionados ao tema.

No capítulo 2, introduzimos a arquitetura proposta, descrevendo a função de cada módulo. O modelo é validado funcionalmente através de simulação e diagramas de tempo são apresentados para alguns exemplos de comunicação entre processadores e módulos de memória.

No capítulo 3, a partir de uma aplicação paralela, analisamos o desempenho oferecido pela rede, em comparação com a respectiva execução sequencial. Ao final do capítulo são avaliados os resultados obtidos nas simulações.

No capítulo 4, fazemos uma síntese do trabalho desenvolvido com as conclusões e perspectivas para trabalhos futuros.

No Apêndice A temos os modelos, em VHDL, dos componentes utilizados na arquitetura. O Apêndice B apresenta as funções utilizadas na execução da aplicação, descritas em linguagem de montagem e linguagem C.

Capítulo 1

REDES DE INTERCONEXÃO

ESTE capítulo abordará os principais tópicos relacionados a redes de interconexão e apresentará alguns exemplos de implementação. O uso das redes de interconexão é a solução adotada para garantir uma comunicação eficaz entre os diversos elementos da rede. Neste capítulo também apresentaremos os trabalhos relacionados à este tema.

1.1 Sistemas embutidos

Um sistema embutido ou SoC (System-on-Chip) é um sistema completo em um único encapsulamento que implementa funcionalidades para um propósito específico. Estes sistemas aproveitam a tecnologia VLSI (*Very-Large-Scale Integration*), que proporciona *chips* de alto desempenho ocupando, através da evolução tecnológica, uma área de silício cada vez menor (SCHALLER, 1997). SoCs possuem internamente microprocessadores, memórias, interface de entrada e saída e componentes que realizam funções digitais específicas como, por exemplo, os chamados DSPs (*Digital Signal Processors*). Todos estes componentes trocam informações através de um meio de interconexão. Um exemplo de SoC genérico pode ser visto na Figura 1 (MADISETTI; SHEN, 1997).

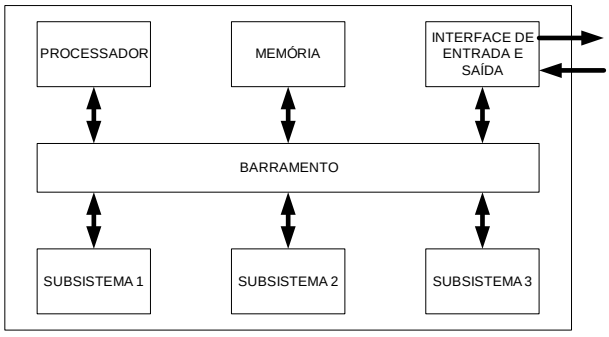


Figura 1: Estrutura Simplificada de um SoC

A fim de reduzir o tempo de projeto de um SoC, normalmente utilizam-se blocos de propriedade intelectual ou IPs (*Intelectual Property*) (GUPTA; ZORIAN, 1997) (LACH; MANGIONE-SMITH; POTKONJAK, 1998). IPs são componentes que podem ser reutilizados em vários projetos de sistemas embutidos com poucas adaptações, sendo essa a ideia básica. Desenvolvedores podem adquirir IPs de terceiros ou projetá-los de maneira flexível para que estes possam ser utilizados em vários projetos. O conceito de propriedade intelectual é aplicado em outras áreas além da eletrônica (GALLINI; SCOTCHMER, 2002).

Uma forma do projetista disponibilizar um IP é através do código fonte em uma linguagem de descrição de *hardware*, que pode ser editado antes da síntese possibilitando a adaptação do IP ao projeto. A segunda forma é denominada *netlist*, sem a possibilidade de alteração antes da síntese. Uma terceira opção são os núcleos de *hardware*, onde os IPs são descritos para uma implementação física na camada do circuito integrado, também sem possibilidade de alteração (PALMA *et al.*, 2002).

Sistemas embutidos necessitam de um meio de interconexão para interligar todos os IPs. O meio de interconexão mais encontrado em tais sistemas é denominado barramento compartilhado. No barramento existe um árbitro que é a entidade que vai gerenciar qual é o componente que pode usar o barramento naquele momento. Um componente, para se tornar o mestre do barramento, deve solicitar ao árbitro. Quando o barramento estiver disponível, o árbitro então sinaliza, indicando esta situação. Uma das vantagens deste tipo de arquitetura é a compatibilidade com diferentes IPs. Uma desvantagem é o tempo de arbitragem que cresce com o acréscimo de processadores (BJERREGAARD; MAHADEVAN, 2006).

1.2 Sistemas embutidos multiprocessados

A demanda por dispositivos cada vez mais rápidos, em virtude de novas aplicações, principalmente aplicações multimídia, pressionou a indústria a buscar novas soluções. Encontrar o paralelismo nas aplicações é uma destas soluções, ou seja, particionar as aplicações em partes menores que possam ser executadas em paralelo. Uma maneira de fazer isso é através de sistemas multiprocessados, denominados Sistemas Embutidos Multiprocessados ou MPSoCs (*Multi-Processor System-on-Chip*).

Um exemplo de MPSoC pode ser visto na Figura 2. Nestes sistemas temos processadores e outros componentes interconectados através de um meio de interconexão. O

barramento não é uma opção para o meio de interconexão nestes sistemas, pois apenas um componente possui o controle do barramento em cada intervalo de tempo, o que restringe comunicações paralelas (WOSZEZENKI, 2007). Diante do exposto surgiu a necessidade dos projetistas vislumbrarem outros meios de conexão, utilizando o conceito de redes de interconexão.

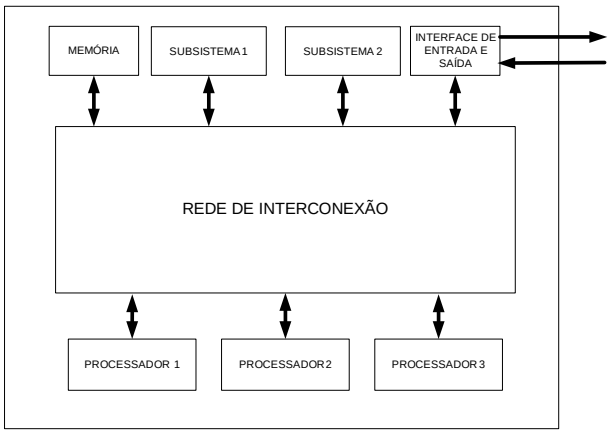


Figura 2: Estrutura Simplificada de um MPSoC

Em MPSoCs existe a necessidade de cada núcleo de processamento se comunicar com os demais. Esta comunicação pode ser feita através de duas maneiras, por acessos a uma memória compartilhada ou através de uma memória distribuída entre os processadores, neste caso a comunicação é feita através de troca de mensagens (DUNCAN, 1990).

Na comunicação através de memória compartilhada (CULLER; SINGH; GUPTA, 1999), todos os processadores compartilham uma mesma memória e por consequência um mesmo espaço de endereçamento. Neste caso a comunicação entre os processadores é realizada através de acesso a variáveis compartilhadas em memória. Como todos os processadores são capazes de acessar qualquer local da memória, a troca de informação entre os processadores ocorre simplesmente por meio de operações de *load* e *store* (PATTERSON; HENNESSY, 2009). Este tipo de implementação possui duas ramificações denominadas: UMA (*Uniform Memory Access*) e NUMA (*Non-Uniform Memory Access*). Nos multiprocessadores UMA mostrados na Figura 3 todos os processadores levam o mesmo tempo para acessar a memória principal independente de qual processador a requisita. Nos processadores NUMA, mostrado na Figura 4, o processador possui sua memória local que é agregada ao endereço global da arquitetura. Cada processador pode acessar a sua memória local e a memória local dos outros processadores. Entretanto quando um processador acessa a sua

memória local, este acesso é realizado de maneira mais rápida do que quando o acesso é realizado a memória local de outro processador.

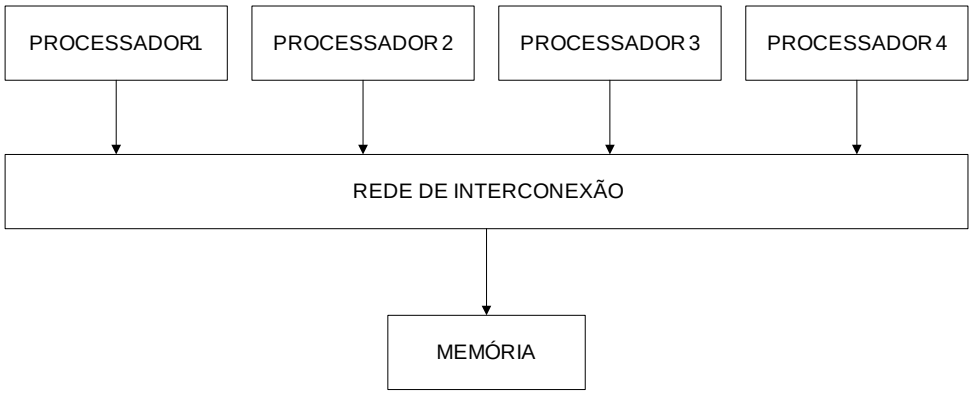


Figura 3: Multiprocessador UMA

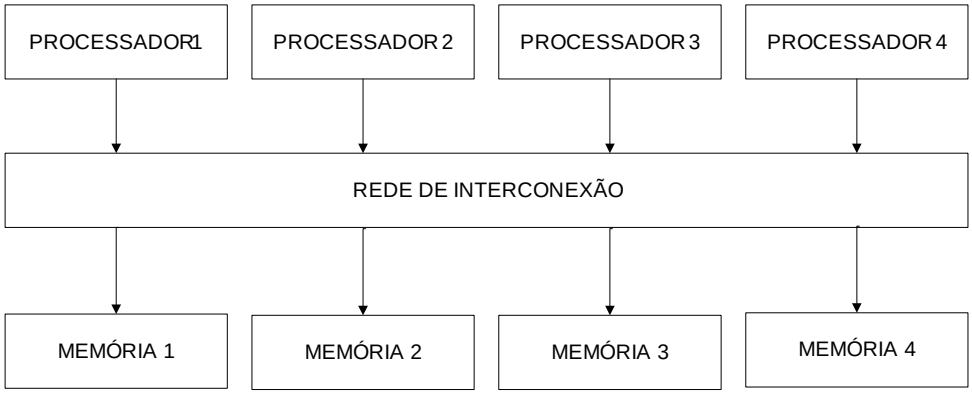


Figura 4: Multiprocessador NUMA

O modelo de memória distribuída (CYBENKO, 1989) mostrado na Figura 5 usa a troca de mensagens para comunicação entre processadores. A troca de mensagens é necessária, pois cada núcleo de processamento tem sua memória privada, sendo assim um processador não pode acessar diretamente a memória do outro. Um processador quando necessitar de um dado produzido por outro deverá enviar uma mensagem para este solicitando o dado e o processador responderá com outra mensagem contendo o dado solicitado. Ao final, o processador que recebeu a mensagem acusará tal recebimento e a comunicação será encerrada.

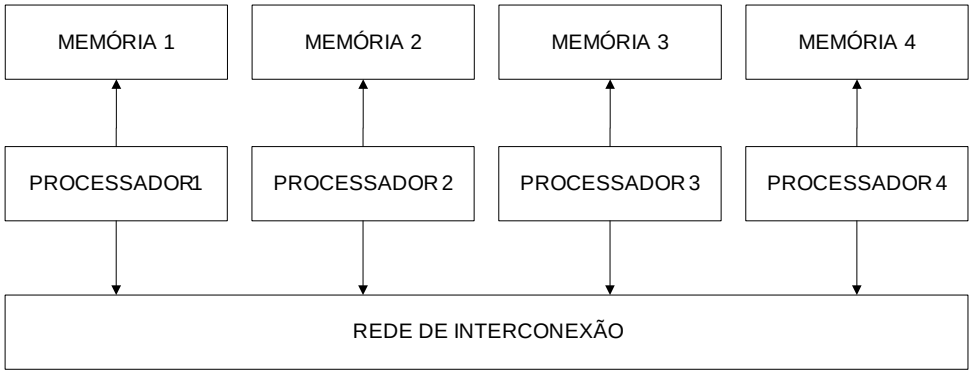


Figura 5: Multiprocessador de memória distribuída

1.3 Redes de interconexão

Redes de interconexão provêem um meio de comunicação entre os diferentes recursos de um sistema multiprocessado. Recurso é um termo genérico utilizado para designar os diferentes subsistemas, tais como memória, processador, interface de entrada e saída entre outros.

1.3.1 Considerações sobre projetos de redes de interconexão

Para a escolha de uma rede de interconexão é necessário avaliar alguns fatores tais como: desempenho, escalabilidade, simplicidade, reusabilidade e custo (DUATO; YALAMANCHILI; LIONEL, 2002).

O desempenho de uma rede de interconexão é definido por dois fatores principais. O primeiro fator é o tempo necessário para que uma mensagem seja gerada em uma origem e chegue a um destino. Esta medida também é chamada latência da rede (PATTERSON; HENNESSY, 2009). Outro fator é o *throughput* (LI; HARMS; HOLTE, 2005) que é definido como a quantidade de informação que trafega na rede por unidade de tempo. Em outras palavras quanto menor a latência da rede e maior o *throughput*, maior será o desempenho da rede.

Escalabilidade reflete quantos recursos podemos conectar na rede com a finalidade de aumentar o desempenho. Idealmente, o desempenho de um sistema deve aumentar proporcionalmente com o acréscimo de recursos. Quanto mais recursos acrescentam-se ao sistema com o aumento de desempenho maior é a escalabilidade. Por outro lado em redes não escaláveis o acréscimo de componentes torna-se um gargalo para o resto do sistema fazendo com que o desempenho global diminua com o acréscimo de recursos.

A simplicidade do sistema pode fazer com que o mesmo alcance maior frequência de *clock* o que aumenta também o desempenho. Adicionalmente, projetistas deste tipo de sistema apreciam redes de interconexão que são fáceis de entender, pois isto torna mais fácil explorar toda a funcionalidade da rede, além de diminuir o tempo de projeto.

Reusabilidade é o quanto a rede pode ser usada em diferentes projetos. Uma rede de interconexão precisa ser flexível para que diferentes IPs possam ser agregados a ela com o mínimo de modificações possíveis. Desta maneira uma mesma estrutura pode ser utilizada em outros projetos com o mínimo de alterações possível.

O custo também é um fator que deve ser analisado em projetos de redes de interconexão. Entretanto existe a necessidade de se fazer um compromisso entre o custo e o desempenho. Felizmente, o custo nem sempre é diretamente proporcional ao desempenho. Usando IPs de maneira adequada poderemos ter uma redução global do custo com o aumento do desempenho.

1.3.2 Classificação das redes de interconexão

As redes de interconexão podem ser classificadas de acordo com sua topologia. Topologias são definidas pela maneira com que os recursos são interconectados na rede (FENG, 1981). Estas topologias podem ser agrupadas em duas classes principais: redes diretas e redes indiretas (ZEFERINO, 2003).

1.3.2.1 Redes diretas

Nas redes diretas cada nó possui pelo menos um recurso conectado a ele. Geralmente estes recursos são processadores que utilizam uma memória local e outras unidades funcionais. Os processadores podem ser do mesmo tipo, formando uma rede homogênea, ou de diferentes tipos formando uma rede heterogênea. Uma arquitetura utilizada para interconectar recursos, em redes diretas, é a denominada NoC (*Network-on-Chip*) (BENINI; BERTOZZI, 2005) (DALLY; TOWLES, 2001). Um elemento comum das redes diretas é denominado roteador, sendo que o conjunto destes roteadores definiriam uma NoC. Na Figura 6 temos um exemplo de nó utilizado nas redes diretas.

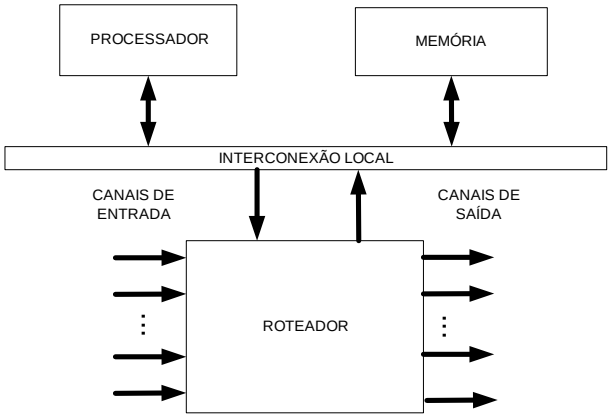


Figura 6: Arquitetura genérica do nó da rede

NoCs utilizam conceitos de redes de computadores e comunicação de dados. Uma NoC consiste de roteadores, adaptadores de redes e *links* de comunicação. Tais redes podem implementar uma série de topologias, incluindo malha 2D, toroide e hipercubo (FENG, 1981) (DUNCAN, 1990). A Figura 7 mostra o exemplo de um roteador utilizado em uma NoC. Nesta figura podemos observar que o roteador é composto por quatro componentes principais: *buffer*, chave, o bloco de arbitragem e o bloco de roteamento. Os *buffers* são necessários para armazenar uma informação enquanto uma entrada não obteve a permissão para alcançar uma saída. A chave é responsável pela conexão das entradas com uma saída, geralmente este bloco é composto por uma chave *crossbar* NxN, onde N é o número de entradas e saídas do roteador (HU; MARCULESCU, 2003). O bloco de arbitragem é responsável por selecionar, caso duas ou mais entradas solicitem uma mesma porta de saída, qual delas a alcançará. A lógica de roteamento é responsável por definir o caminho que a informação seguirá da origem até o destino. A motivação para o estudo e aplicação de NoCs em sistemas embutidos multiprocessados são: confiabilidade, eficiência energética, escalabilidade e reusabilidade (TAYAN, 2009) (MORAES *et al.*, 2004).

Geralmente as informações trocadas entre os recursos são implementadas por meio de troca de mensagens, que são divididas em porções menores denominadas pacotes. Um pacote deve conter informações sobre o nó de destino da mensagem. Esta informação se encontra no início do pacote e é denominada cabeçalho. Com a informação do destino do pacote computada, deverá ser definido o percurso que a mensagem seguirá da origem até o destino, sendo a função do algoritmo de roteamento definir este caminho. Com a finalidade de diminuir a latência da rede, os pacotes podem ser ainda divididos em porções menores denominas *flits* (NI; MCKINLEY, 1993).

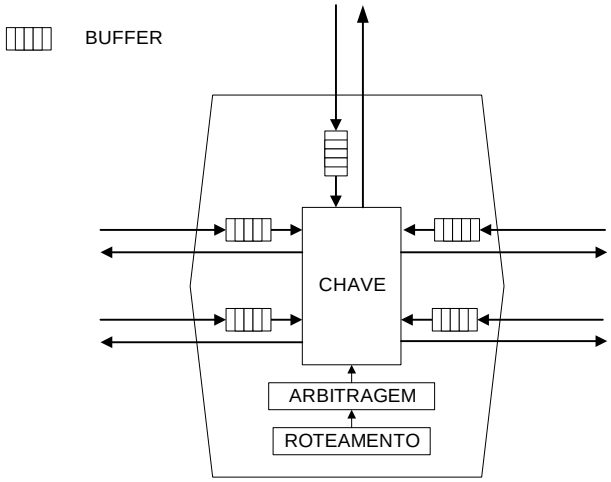


Figura 7: Roteador utilizado em NoCs

As redes diretas podem ser divididas em topologias, de acordo com duas propriedades: diâmetro da rede e grau do nó. O diâmetro da rede é a distância máxima entre dois nós. O grau do nó é o número máximo de conexões que um nó pode implementar com os nós vizinhos. Na Figura 8 tem-se alguns exemplos de topologias de redes diretas. Em (a), temos o tipo de conexão mais simples que é a linear, na qual todos os nós estão conectados por roteadores de grau 2, exceto os roteadores das extremidades que só estão conectados a um roteador. A topologia em anel, mostrada em (b), é semelhante à topologia linear exceto pelos roteadores das extremidades que são conectados. Na topologia malha 2D, apresentada em (c), um nó se conecta a 4 nós adjacentes exceto as extremidades da malha, que se conectam a apenas 3 nós adjacentes. Em (d), temos a rede toroide que é semelhante à rede em malha 2D exceto pelos roteadores da extremidade que são interconectados. Na rede hipercubo, vista em (e), cada nó é conectado a outros três. Já na topologia totalmente conectada, mostrada em (f), todos os nós são interconectados, onde cada roteador tem grau $N - 1$, sendo N o número de roteadores existentes na rede.

1.3.2.2 Redes indiretas

Nas redes indiretas o número de roteadores é maior que o número de recursos, isto é, não existe a necessidade de cada roteador estar conectado a um recurso como nas redes diretas. Cada recurso pode ser conectado a outro através de um ou mais roteadores intermediários. Em virtude disto apenas os roteadores onde existem recursos conectados podem ser a origem ou o destino de uma mensagem. De maneira similar às redes diretas, redes indiretas podem ser caracterizadas também por topologias. Podemos interligar re-

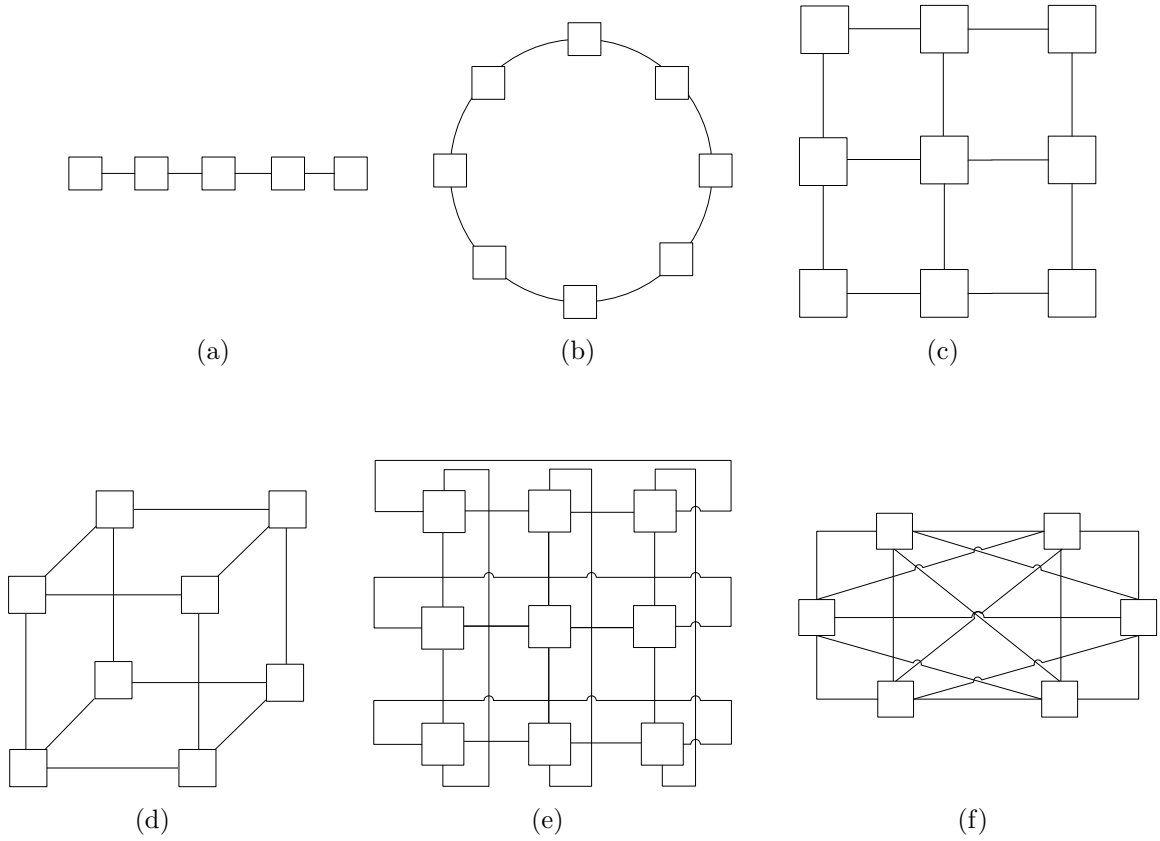


Figura 8: topologias de redes de interconexão direta: linear(a), Anel(b), Malha 2D(c), Hipercubo(d), Toróide(e) e completamente conectada(f)

cursos nas redes indiretas utilizando as topologias *crossbar* (GUPTA; MCKEOWN, 1999) e multiestágio (DUATO; YALAMANCHILI; LIONEL, 2002). A topologia define como os roteadores estão interligados com os demais e pode ser caracterizada por um grafo G(V,A), onde cada vértice V_i representa um roteador e cada aresta A_i representa a ligação entre os roteadores.

A topologia ideal é a de estágio único com N roteadores onde cada roteador possui uma chave *crossbar* NxN, ligando todas as entradas a todas as saídas. Esta rede pode ser concebida para quantidades pequenas e médias de recursos conectados a ela. Entretanto para uma maior quantidade de recursos devem-se usar roteadores intermediários para conectar a origem ao destino, definindo uma topologia multiestágio. Geralmente estes roteadores são organizados de maneira que cada estágio seja conectado a outros estágios a fim de que qualquer entrada possa alcançar uma saída.

A rede *crossbar*, mostrada na Figura 9, permite que qualquer entrada possa ser conectada a uma saída simultaneamente sem contenções. Por este motivo ela é uma rede

não bloqueante. Esta rede é composta por barramentos horizontais e verticais, onde cada ponto de cruzamento entre estas linhas possui uma chave que pode ser aberta e fechada eletricamente (TANENBAUM, 1998). Uma rede *crossbar* é mais econômica do que uma rede direta completamente conectada, que necessitaria de N roteadores cada qual contendo uma chave crossbar NxN (ZEFERINO, 2003). O custo da *crossbar*, no entanto, ainda é proibitivo, pois cresce em uma taxa de C^2 onde C é o custo de cada chave existente na rede. Entretanto esta rede possui alto desempenho, pois permite N comunicações simultâneas.

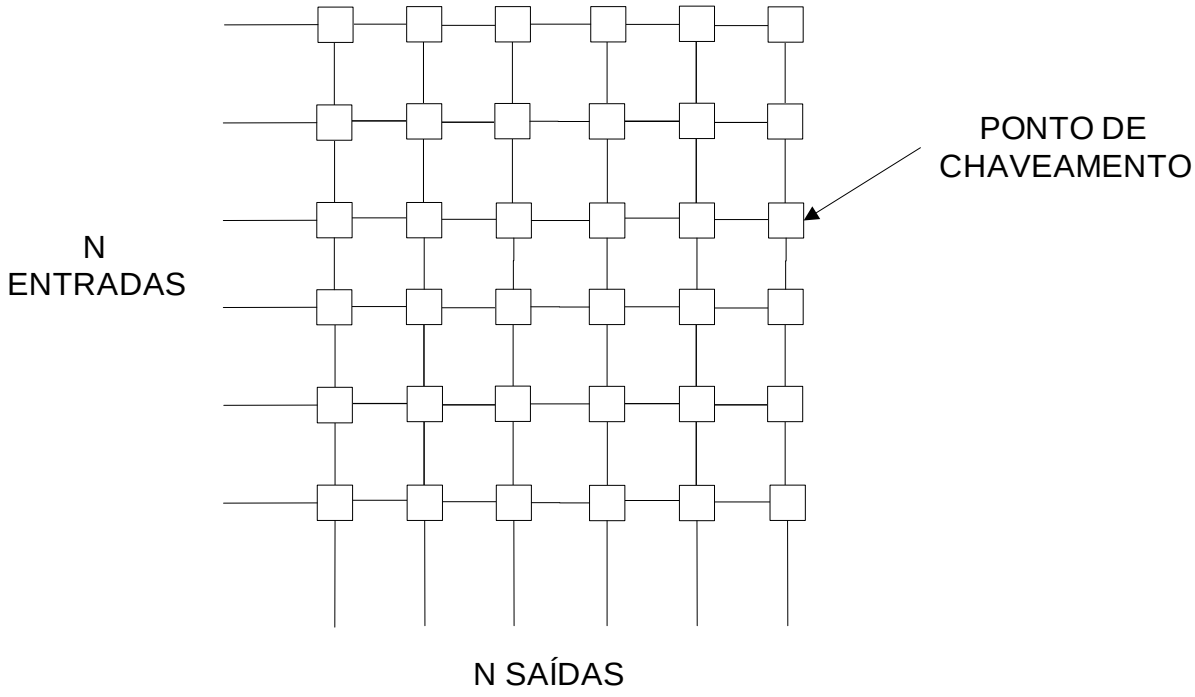


Figura 9: Rede *crossbar*

Outra topologia possível para as redes indiretas é a rede multiestágio (KRUSKAL; SNIR, 1983). Nesta rede, cada roteador é composto de uma chave *crossbar* 2x2. Esta rede é mais econômica que a *crossbar*. Na rede *crossbar* são necessárias N^2 chaves, onde N é o número de processadores conectados na rede. Na topologia multiestágio, para uma rede com N processadores, são necessários $log_2 n$ estágios, cada um com $N/2$ roteadores. Como cada roteador possui 4 chaves, o número necessário de chaves seria $2nlog_2 n$. Por exemplo, para a rede multiestágio mostrada na Figura 10 são necessárias 48 chaves para conectarmos 8 processadores com 8 memórias. Utilizando uma rede *crossbar* seriam necessárias 64 chaves para conectarmos processadores a memórias. Neste tipo de rede cada entrada acessa uma saída através de um único caminho utilizando os estágios intermediários da

rede.

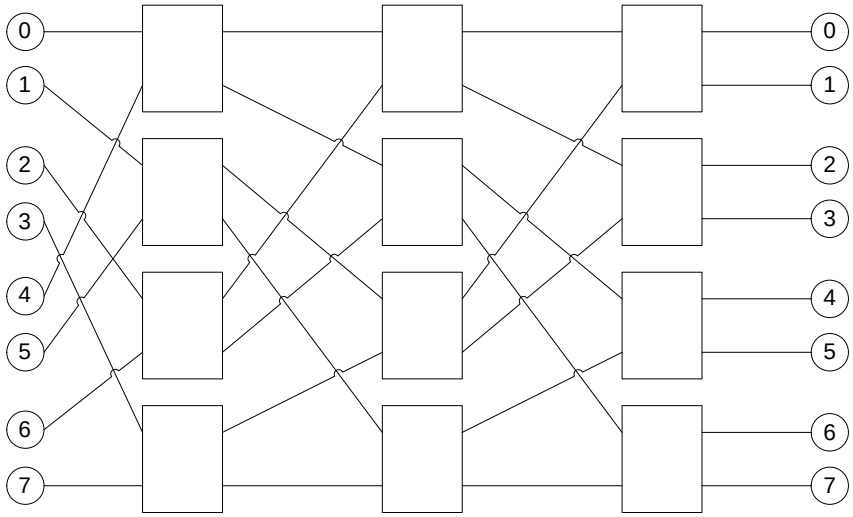


Figura 10: Rede multiestágio

1.4 Trabalhos correlatos

A necessidade de estabelecer a comunicação entre os processadores em um MPSoC gera várias alternativas para a pesquisa de formas eficientes de realizá-la. Estas pesquisas utilizam conceitos de redes de computadores para implementar a comunicação entre os núcleos. Entretanto existe a limitação de área dos *chips*, exigindo que estas redes de interconexão sejam pequenas e rápidas. Em virtude disto existem vários trabalhos relacionados ao tema. Nesta seção serão apresentados alguns destes trabalhos.

1.4.1 Multi-Cluster Network-on-Chip

Em (FREITAS, 2009) é apresentada uma rede de interconexão baseada em rede *crossbar* denomina de MCNoC (*Multi-Cluster Network-on-Chip*). Esta rede pode ter vários *clusters* de processamento interligados. No interior de cada *cluster* existem oito núcleos de processamento, interligados por uma rede *crossbar* 8x8. Na Figura 11 é apresentado um *cluster* da arquitetura proposta. Cada *cluster* implementa uma topologia no espaço (e.g. malha 2D e anel). O *cluster* apresenta os seguintes componentes internos: 8 *cores*, um processador de rede denominado NPoC (*Network Processor on Chip*), uma chave *crossbar* reconfigurável, denominada RCS-NR (*Reconfigurable Crossbar Switch for NoC Router*), e um conjunto de *buffers* de entrada.

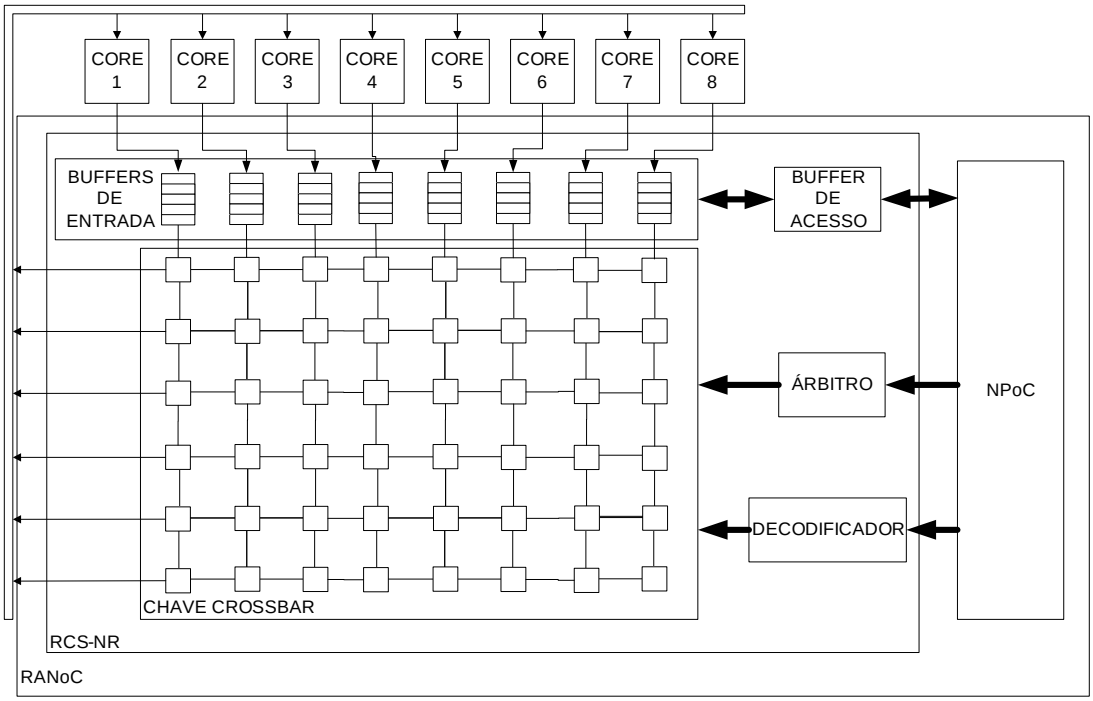


Figura 11: Arquitetura do roteador em chip programável (FREITAS, 2009)

O processador de rede NPoC tem a responsabilidade de monitorar o tráfego na rede através do componente *buffer* de acesso e, quando necessário, reconfigurar a topologia espacial da rede em virtude dos padrões de comunicação. Esta reconfiguração é feita através de registradores internos ao processador denominados registradores de topologia. Estes registradores alimentam o componente decodificador que realizará o chaveamento das entradas com as saídas da rede.

A RCS-NR é o componente responsável por executar o chaveamento de uma entrada com uma saída. Diferentemente de chaves *crossbar* tradicionais, uma entrada pode ser chaveada para uma ou mais saídas diferentes. O árbitro é o componente responsável por fazer as seleções temporais das entradas que alcançarão as saídas. O processador da rede atualiza uma tabela de prioridades, que, ciclo a ciclo é lida pelo árbitro. Cada linha da tabela possui 32 *bits* e cada conjunto de 4 *bits* é responsável por liberar os pacotes de uma entrada para uma saída. Os *buffers* de entrada armazenam o pacote, até que uma entrada seja liberada para uma saída e também oferecem a oportunidade de avaliação do tráfego da rede através do processador da rede. Os componentes RCS-NR e o NPoC formam a RANoC (*Router Architecture for NoC*) .

1.4.2 SoC Interconection Network

Em (ZEFERINO; SUSIN, 2003) é apresentado o SoCIN (*SoC Interconection Network*) que é uma arquitetura que possui um roteador parametrizável para ser usado na elaboração de NoCs de baixo custo. A arquitetura SoCIN implementa um roteamento *wormhole* baseado no algoritmo XY (NI; MCKINLEY, 1993) (GLASS; NI, 1992). O bloco básico de construção é denominado RASoC (*Router Architecture for SoC*). O roteador RASoC pode ser escalável em algumas dimensões: grau do nó (até o grau 5), largura do canal (8, 16, 32 *bits* ou maior), largura do *buffer* (1, 2, 3 palavras ou maior). Tais fatores podem ser adaptados automaticamente dependendo da aplicação. A rede SoCIN utiliza a topologia direta baseada em malha 2D ou toroide. Estas topologias foram escolhidas devido à primeira apresentar baixo custo e à segunda permitir uma menor latência.

Os *links* da rede SoCin incluem dois canais unidirecionais, cada qual estabelece a comunicação em um sentido e possuem sinais de controle independentes. Estes canais possuem *n bits* para dados e dois *bits* para enquadramento do pacote: BOP (*begin-of-packet*), ativado apenas no cabeçalho do pacote e o sinal EOP (*end-of-packet*) é ativado no final da carga útil do pacote. Cada *link* possui também dois sinais de controle de fluxo. O primeiro, denominado VAL, é utilizado para validar os dados no canal. O segundo, denominado ACK, é utilizado para reconhecimento do recebimento dos dados. Estes dois sinais implementam um protocolo denominado *handshake* (HSIAO; JIANG, 1989). Neste protocolo o emissor informa, ao receptor, a necessidade de enviar um dado e o receptor confirma se pode receber este dado. A linha VAL se torna ativa quando um dado existente no *buffer* de saída está pronto para ser enviado. A linha ACK é ativa quando a linha VAL está ativa e o *buffer* de entrada está pronto para receber o dado (ZEFERINO, 2003).

O RASoC possui cinco portas bidirecionais denominadas L (*Local*), N (*North*), E (*East*), S (*South*) e W(*West*). Por se tratar de portas bidirecionais, cada uma apresenta dois módulos: um de entrada, denominado *in*, e outro de saída, denominado *out*, conforme mostrado na Figura 12. Não é permitido uma porta de entrada ser direcionada para uma mesma porta de saída. Por exemplo, a porta Lin não pode ser direcionada para a porta Lout.

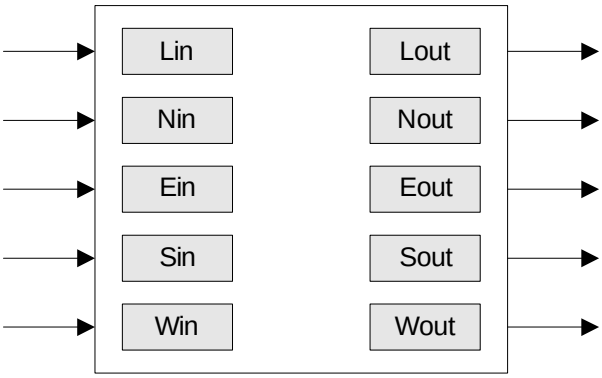


Figura 12: Sinais de interface de RASoC

O módulo de entrada, mostrado na Figura 13, é composto por quatro componentes denominados IFC (*Input Flow Controller*), IB (*Input Buffer*), IC (*Input Controller*) e IRS (*Input Read Switch*). Os sinais de interface do módulo do módulo de entrada com o *link* SoCIN possuem o prefixo IN_ e os sinais de interface do módulo de entrada com o módulo de saída possuem o prefixo X_.

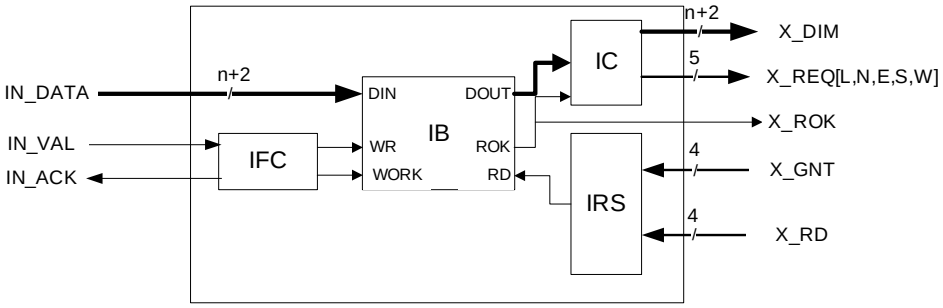


Figura 13: Canal de entrada de RASoC

O componente IFC faz a transição entre o protocolo *handshake* e o protocolo de controle de fluxo existente na FIFO (*First-In-First-Out*). O componente IB é um *buffer* FIFO, responsável pelo armazenamento de pacotes enquanto estes não são encaminhados para uma saída. O componente IC executa a função de roteamento através das informações do *flit* de cabeçalho armazenado em IB. O componente IRS recebe quatro pares de X_RD e de X_GNT vindo de cada canal de saída da rede e encaminha o sinal X_RD para a entrada do bloco IB. Este sinal é o comando para leitura de um dado armazenado no componente IB.

O módulo de saída, mostrado na Figura 14, é composto por quatro componentes denominados OC (*Output Controller*), ODS (*Output Data Switch*), ORS (*Output Read*

Switch) e OFS (*Output Flow Controller*). Os sinais com o prefixo OUT_ são sinais de interface deste componente com o *link* SoCIN.

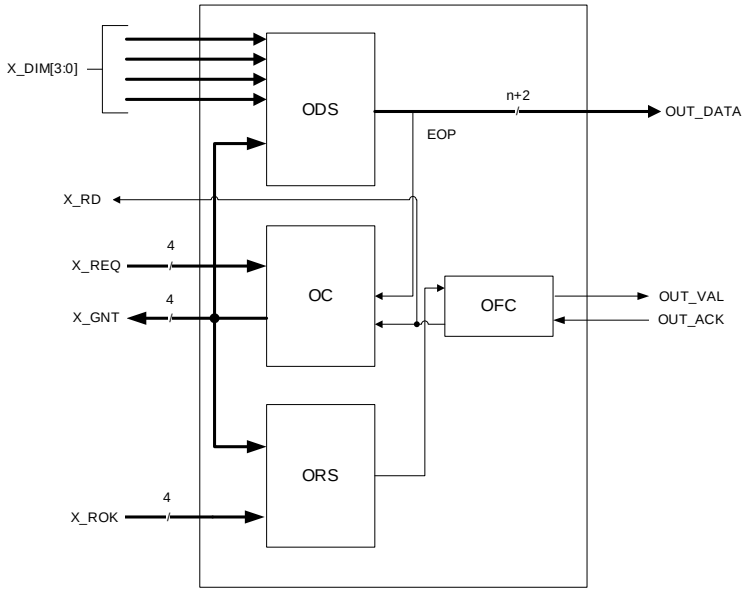


Figura 14: Canal de saída de RASoC

O componente OC implementa o algoritmo *round-robin* (PASRICHA; DUTT, 2008), selecionando quais dos requisitantes poderão acessar o canal de saída. De acordo com o sinal X_GNT de concessão, é definida qual entrada X_DIN será encaminhada para a saída do circuito ODS e qual a entrada X_ROK será encaminhada para a saída do circuito ORS. Não há nenhuma diferença funcional entre o protocolo *handshake* e o protocolo FIFO, no lado do emissor. Em virtude disto o bloco OFC conecta diretamente a saída X_ROK selecionada ao canal OUT_VAL e OUT_ACK a saída X_RD. O Bloco OC também é responsável por monitorar o sinal EOP, com a finalidade de, ao final da transmissão, desfazer o enlace de comunicação.

1.4.3 Rede intrachip HERMES

Em (MORAES *et al.*, 2003) é apresentada a rede intra-*chip* HERMES. Esta rede utiliza a topologia em malha 2D. Semelhante à rede SoCIN, cada roteador, denominado chave HERMES e mostrado na Figura 15, possui 5 portas: E (*East*), W (*West*), N (*North*), S (*South*) e L (*Local*). Cada porta possui *buffers* para armazenamento temporário de informação. A chave implementa o algoritmo de roteamento XY. Cada pacote é dividido em partes menores denominadas *flits*, sendo cada um de 8 *bits*. O primeiro flit contém informação sobre roteamento e os outros *flits* seguem a rota definida pelo primeiro.

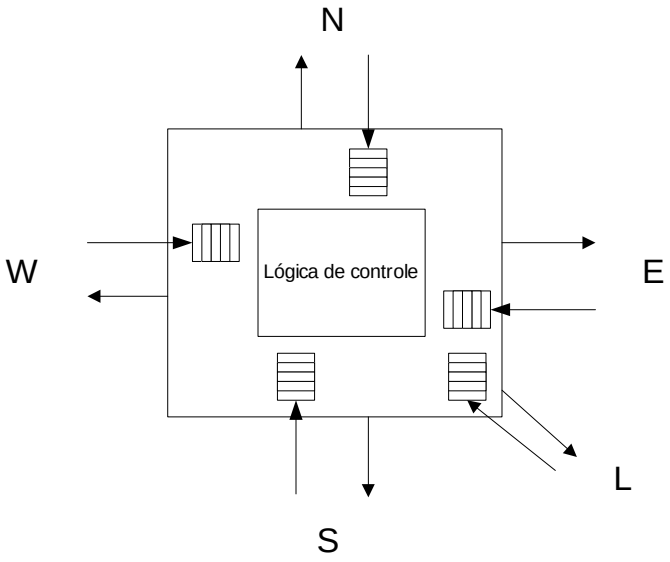
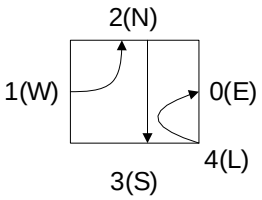


Figura 15: Chave HERMES

A lógica de controle é constituída por dois blocos: roteamento e arbitragem. Quando uma chave recebe o *flit* de cabeçalho do pacote, o módulo de roteamento o encaminha para a porta de saída correspondente. Cada chave intermediária compara o endereço atual do pacote com o endereço armazenado no cabeçalho, utilizando o algoritmo XY. Quando o pacote chega ao destino (coordenada de destino menos coordenada da chave igual a zero) é encaminhado para a porta local da chave onde o recurso está conectado.

Quando uma conexão entre uma porta de entrada e uma porta de saída é estabelecida, uma tabela de roteamento é atualizada. Esta tabela possui três vetores: *free*, *in* e *out*. O vetor *free* indica se a porta de saída está livre (*free*=1) ou se está ocupada (*free*=0). O vetor *in* indica a conexão de uma porta de entrada com uma porta de saída, sendo preenchido com a identificação da porta de saída conectada. O vetor *out* indica a conexão de uma porta de saída com uma porta de entrada, sendo preenchido com a identificação da porta de entrada conectada. Na Figura 16 temos um exemplo de como, a partir de conexões implementadas na chave, a tabela de roteamento é preenchida.



	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 16: Vetores de roteamento

Cada chave pode implementar cinco conexões simultaneamente, desde que duas ou mais entradas não solicitem uma mesma saída ao mesmo tempo. Caso isto ocorra, o árbitro é o componente que irá garantir o acesso de somente uma delas à porta de saída, em cada intervalo de tempo. O árbitro implementado utiliza o esquema de prioridade rotativa no qual a prioridade de uma porta depende da última porta que teve a conexão atendida. Se, por exemplo, a porta atendida foi a porta com o índice 3 (*North*), a próxima que será atendida é aquela com índice 4 (*South*).

Cada chave HERMES possui 5 *buffers* de entrada que funcionam como filas, em que cada uma possui dois ponteiros denominados *first* e *last*. O ponteiro *first* aponta para a posição onde se encontra o *flit* que está pronto para ser transmitido. O ponteiro *last* aponta para a posição onde deve ser acrescentado o próximo *flit* a ser armazenado. A função dos *buffers* é armazenar um pacote até que a porta de saída seja liberada.

1.4.4 Rede *crossbar* reconfigurável com controle de banda adaptativo

Em (KIM *et al.*, 2005) é proposta uma rede *crossbar* que, através de um barramento extra, provê uma maior largura de banda para uma determinada entrada. Este projeto se baseia no conceito de que diferentes IPs podem requerer, em certos momentos, uma largura de banda extra para o tráfego da rede. A chave *crossbar* tradicional provê sempre a mesma largura de banda para todos os IPs conectados a ela. A implementação proposta encontra-se na Figura 17. Nesta figura verifica-se o barramento adicional da rede, que pode ser conectado a qualquer IP da rede para fornecer a largura de banda extra. O barramento extra tem a mesma largura de banda dos outros barramentos da rede. Por este motivo a largura de banda total da rede é $(N + 1)/N$ vezes maior que a largura de banda da rede *crossbar* tradicional.

Para decidir se um IP necessita do barramento extra é necessário verificar a vazão de *bits* na rede. Esta decisão é tomada através da avaliação dos *buffers* de entrada da rede. Se estiverem cheios, a razão de *bits* que chegam à rede é maior que a razão de *bits* que são encaminhados para uma saída. Neste caso, o componente denominado escalonador de barramento detecta tal situação e ativa o barramento adicional, acelerando a transferência de pacotes. Quando duas ou mais portas necessitam de uma maior largura de banda, a concessão para uso do barramento extra é controlada através de um árbitro. No caso em

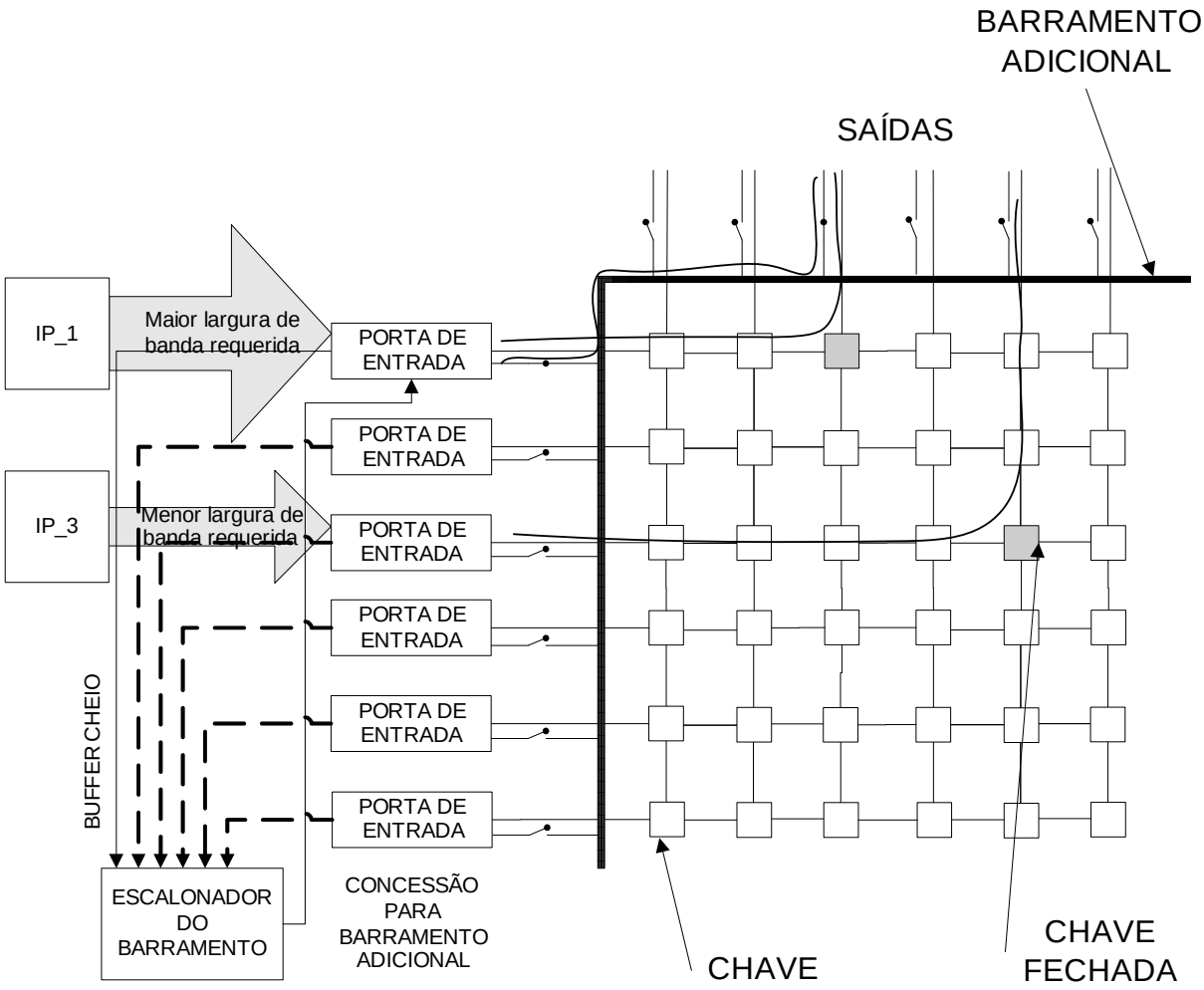


Figura 17: Chave *crossbar* proposta com barramento adicional (KIM *et al.*, 2005)

que o IP tem o controle do barramento extra e do barramento da rede *crossbar* ao mesmo tempo, a largura de banda torna-se duas vezes maior.

1.4.5 Rede *crossbar* utilizando algoritmo de arbitragem NA-MOO

Em (LEE; LEE; YOO, 2003) é proposto um algoritmo de arbitragem distribuído embutido em uma rede *crossbar*. Neste algoritmo, o caminho entre as entradas e a saída é composto por $log_2 N$ estágios de multiplexadores, onde N é o número de recursos conectados à rede. Cada barramento da rede *crossbar* possui N entradas e uma saída. O circuito para $N = 8$, que implementa um barramento da rede *crossbar*, é mostrado na Figura 18.

Ao contrário do algoritmo de arbitragem *round-robin*, o NA-MOO é um algoritmo distribuído através dos nós da *crossbar*. A prioridade e determinada pelo valor n de cada macro-escalonador NA-MOO, que é chamado de valor preferencial. Cada nó da

chave possui um macro-escalonador e cada macro-escalonador NA-MOO possui um valor preferencial, que pode ser 1 ou 0. Este valor vai definir qual das duas entradas de cada multiplexador serão encaminhadas para a saída, em cada nó da *crossbar*. Este valor preferencial só é utilizado quando duas portas de entrada do multiplexador estão requisitando a saída ao mesmo tempo. Se apenas uma requisitar a saída, o valor preferencial será ignorado e esta porta será escalonada para a saída.

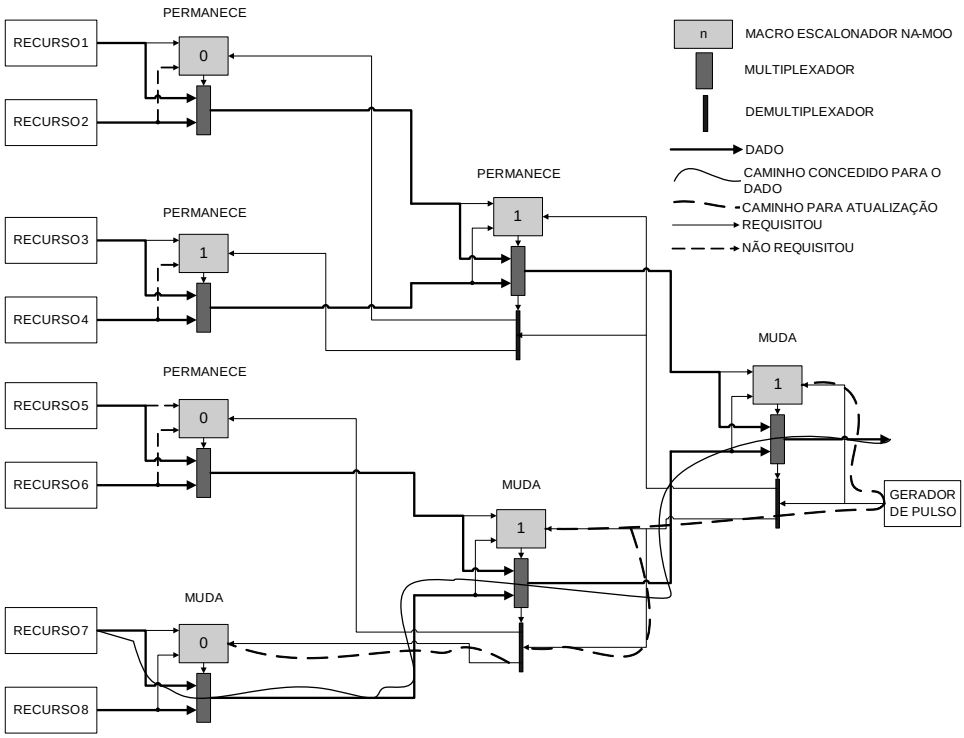


Figura 18: Circuito NA-MOO proposto (LEE; LEE; YOO, 2003)

Na Figura 18 vemos a propagação de um dado em direção à saída do barramento, onde cada macro-escalonador NA-MOO definiu a saída de cada multiplexador da rede, de maneira a interligar uma das entradas à saída. Neste exemplo observa-se que os recursos 1, 4, 7, e 8 solicitaram o uso do barramento. Entretanto, pelos valores preferenciais dos escalonadores NA-MOO da rede, a entrada 7 foi a que alcançou a saída. Após a concessão do caminho, um pulso percorre a rede, através do mesmo caminho do dado, em sentido oposto, mudando o valor dos macro-escalonadores NA-MOO envolvidos no roteamento, permitindo que uma outra entrada possa alcançar uma saída. O algoritmo NA-MOO é desleal dando mais concessões para as portas longe daquelas ativas, porém a área ocupada pela implementação em *hardware* deste algoritmo é menor do que a área ocupada pela implementação em *hardware* do algoritmo *round-robin*.

1.4.6 Rede *crossbar* multidirecional baseada em NoC

Em (MHAMDI; GOOSSENS; SENIN, 2010) é proposta uma rede *crossbar* com $(N/4) * (N/4)$ roteadores, onde N é o número de recursos conectados à rede. Os recursos podem ser conectados nos quatro lados da rede. Esta rede é mostrada na Figura 19 e é denominada Rede *crossbar* Multidirecional baseada em NoC ou MDN (*MultiDirectional Network*). Cada roteador da rede possui quatro entradas e quatro saídas. As entradas possuem um *buffer* FIFO para armazenamento temporário do pacote, enquanto uma entrada não é liberada para uma saída. Cada roteador utiliza o algoritmo de roteamento *round-robin* para solucionar possíveis conflitos.

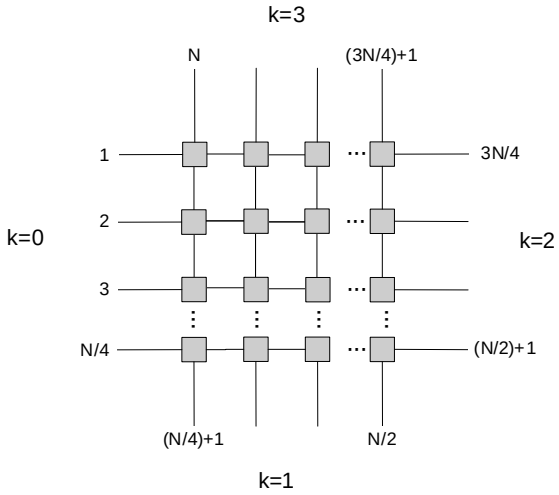


Figura 19: Rede baseada em roteadores MDN (MHAMDI; GOOSSENS; SENIN, 2010)

A arquitetura MDN é escalável. Cada estrutura da rede pode ser empilhada de maneira que a arquitetura proposta tenha múltiplos planos P, conforme mostrado na Figura 20, que são verticalmente conectados apenas nos roteadores de borda. As entradas e saídas da rede estão conectadas apenas no plano central ($PM = \lfloor P/2 \rfloor$). Dependendo da posição roteador nesta implementação, este pode ter diferentes graus. Os roteadores dos planos 0 e $P-1$ possuem grau 4 exceto os roteadores dos vértices do cubo que possuem grau 3. Os roteadores dos planos $P(P \in \{0, ..., PM-1\} \cup \{PM+1, ...P-1\})$ tem grau 4, os roteadores de bordas tem grau 5. Os roteadores de borda do plano central tem grau 6.

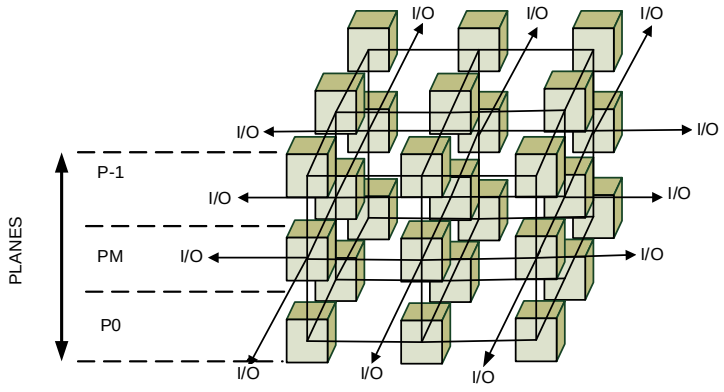


Figura 20: A arquitetura MDN com P planos (MHAMDI; GOOSSENS; SENIN, 2010)

O roteamento desta chave é feito por dois algoritmos: o algoritmo XY e o algoritmo XY balanceado. O algoritmo XY é utilizado quando uma entrada e uma saída estão em dois lados perpendiculares da malha. O algoritmo XY balanceado é usado para balancear o tráfego na rede, onde uma volta extra é dada em uma das colunas anteriores. Um pacote com destino a uma saída x pode ser encaminhado para as portas sul ou norte se a origem for uma porta leste ou oeste quando $x = (N/4 - i + j)\ mod\ (N/4) + k(N/4)$, $(k \in [0, 3])$, ou pode ser encaminhado para as portas leste ou oeste se a origem for uma porta sul ou norte, quando $x = j$, onde i e j corresponde à posição do roteador na rede, N é o número de entradas/saídas da rede e k são os 4 lados da rede.

No caso da implementação utilizando vários planos, as portas de entradas e saídas encontram-se apenas no plano do meio. Dependendo da porta de saída desejada, os pacotes podem ser encaminhados para o plano de cima ou para o plano de baixo. Se a porta de destino do pacote for ímpar, o pacote é encaminhado para o plano de cima. Caso a porta de saída seja par, o pacote é encaminhado para o plano de baixo. É preciso determinar por qual plano P_z o pacote será roteado. Isto é feito quando $x\ mod\ (P/2) = (N/4 - i((k+1)\ mod\ 2) - (j(k\ mod\ 2)) + P_z)\ \mod\ [P/2]$. A operação módulo distribui o tráfego através dos planos e evita congestionamentos.

1.4.7 Arquitetura NoC com *crossbar* dupla

Em (ZHANG; MORRIS; KODI, 2011), é proposto um roteador para uma NoC, implementado com duas redes *crossbar*, denomina DXbar. Com esta implementação, é esperada uma redução da potência requerida pela rede e um aumento do seu desempenho, utilizando os *buffers* de entrada apenas quando existe a retenção de pacotes. A seleção destas entradas é feita utilizando demultiplexadores.

Figura 21 mostra a chave proposta. Ela é composta por 6 componentes principais: *buffers*, chave primária, chave secundária, computação de roteamento, alocação da chave e o elemento de processamento. Os *buffers* são utilizados para armazenar temporariamente os pacotes que não tiveram a permissão para alcançar uma saída. A chave primária é responsável por encaminhar os *flits* que não foram armazenados nos *buffers*. A chave secundária é o elemento responsável por encaminhar os *flits* que foram armazenados nos *buffers*. Estes *flits* não obtiveram permissão para acessar a saída e por este motivo foram armazenados temporariamente nos *buffers*. O componente alocação da chave controla a saída dos demultiplexadores, decide qual a chave será usada no roteamento e qual das entradas alcançará a saída dos multiplexadores, este componente é responsável também pela arbitragem da rede. O componente cálculo de roteamento é o componente que armazena o algoritmo de roteamento da chave. O componente elemento processador é o processador da rede propriamente dito.

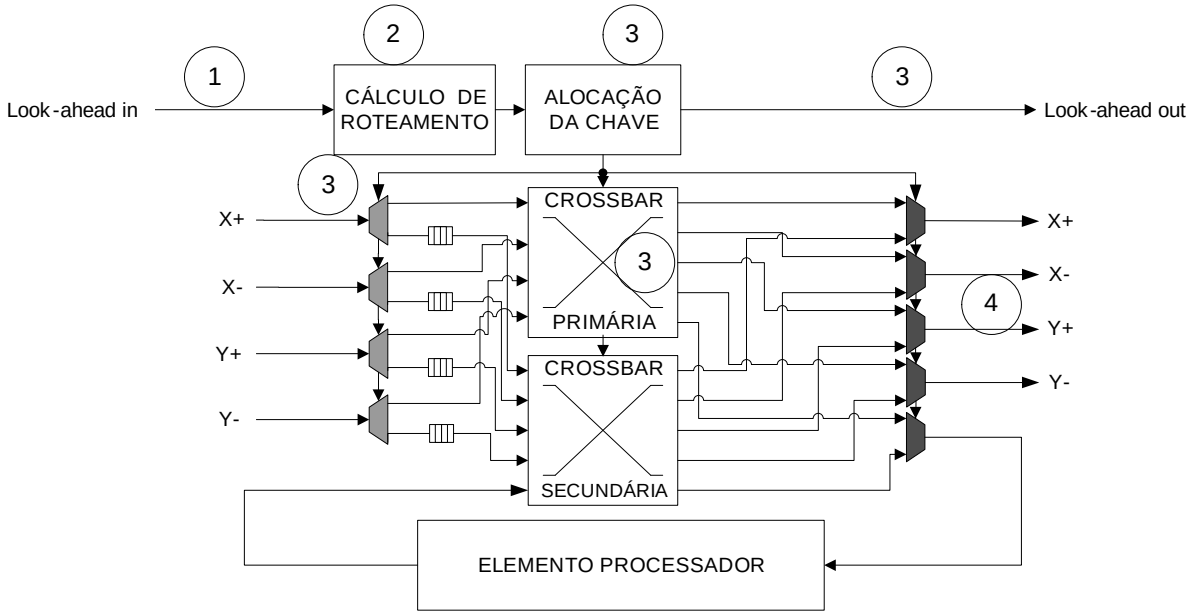


Figura 21: Roteador DXbar (ZHANG; MORRIS; KODI, 2011)

A propagação da mensagem, através dos roteadores da rede, é feita utilizando um *pipeline* de 4 estágios, conforme mostrado na Figura 22. Os componentes que realizam cada etapa do pipeline estão indicados na Figura 21. Em 1 o sinal de *look ahead in* chega ao roteador. Em 2 é realizado o cálculo do roteamento. Em 3 é realizado o chaveamento das entradas da rede, a alocação da chave *crossbar* que realizará o roteamento e o sinal *look ahead out* é encaminhado para o próximo roteador. Em 4 o *flit* alcança a saída.

No estágio LA é realizada a leitura do sinal de *look ahead in* do roteador origem, que contém informações de roteamento. Os estágios SA e ST são executados em paralelo e são responsáveis pela alocação da chave e pelo chaveamento. Paralelamente, a saída *look ahead out* é encaminhada para o próximo roteador, permitindo que o este possa executar a computação do roteamento um ciclo antes da chegada dos dados. No estado LT o dado que está na saída do roteador é encaminhado para o próximo roteador. Observa-se que através do sinal *look ahead out* a informação de roteamento é encaminhada para o próximo roteador antes da chegada do dado, o que acelera o roteamento em um ciclo de *clock* por chave.

Em caso de conflito, ou seja, duas entradas desejarem acessar uma saída ao mesmo tempo, a arbitragem é feita pelo tempo que o pacote está na rede. Pacotes mais antigos na rede têm maior prioridade. Entretanto os pacotes que não necessitam ser armazenados em *buffers*, mesmo sendo mais recentes, tem prioridade no roteamento.

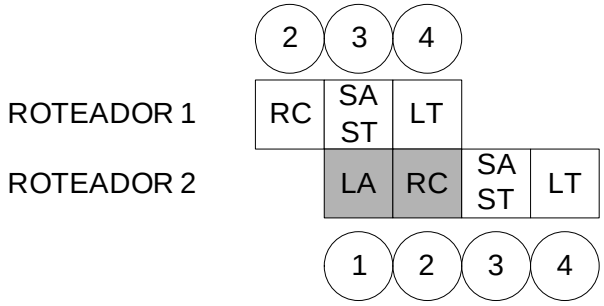


Figura 22: *Pipeline* do Roteador DXbar

O roteamento dos pacotes na rede pode ser feito utilizando os algoritmos XY e o *west-first* adaptativo (GLASS; NI, 1992). No algoritmo *west-first*, os pacotes são roteados através da direção oeste da rede e se necessário podendo ser depois roteados tanto para a direção norte, leste ou sul. A adaptação acontece apenas quando a posição X de início do roteamento está a oeste da posição X de destino. Nesta situação os pacotes podem transitar livremente do norte para o leste ou do sul para o leste. Entretanto as curvas do norte para o oeste e do sul para o oeste estão proibidas o que torna este algoritmo livre de dependência cíclica. Na Figura 23 temos as direções permitidas para este algoritmo em linhas sólidas e as não permitidas em linhas pontilhadas.

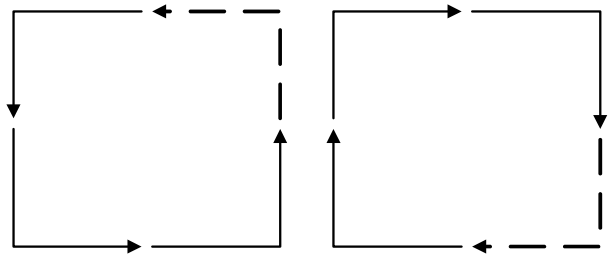


Figura 23: As seis voltas permitidas para o roteamento *west-first*

1.5 Considerações finais do capítulo

Neste capítulo foi apresentado o estado da arte das redes de interconexão, foram apresentadas as redes diretas e indiretas, as topologias utilizadas em cada uma destas redes e os trabalhos relacionados com este assunto. No próximo capítulo será apresentada a arquitetura proposta nesta dissertação.

Capítulo 2

ARQUITETURA DA REDE

A MACROARQUITETURA proposta é baseada na topologia *crossbar* (DUATO; YALAMANCHILI; LIONEL, 2002). No projeto utilizamos a rede *crossbar* para conectar processadores a módulos de memória em uma arquitetura escalável NxN, isto é, a arquitetura possui N processadores a N módulos de memória. Esta macroarquitetura pode ser vista na Figura 24, onde identificamos os quatro componentes básicos: processador, módulo de memória, controlador do barramento e chave. O controlador do barramento é o componente que realiza a conexão do processador com o módulo de memória habilitando a chave correspondente.

Os barramentos possuem um índice j que varia de 0 a $N-1$ e os processadores possuem um índice i que varia de 0 a $N-1$. As chaves cujo índice do BARRAMENTO(j) é igual ao índice do PROCESSADOR(i) estão frequentemente conectadas, pois o programa a ser executado no PROCESSADOR(i) está no BARRAMENTO(j), com $i = j$. O PROCESSADOR(i) só será desconectado do BARRAMENTO(j) quando o PROCESSADOR(i) requisita o uso de um BARRAMENTO(j) com $i \neq j$, ou quando o PROCESSADOR(i) atinge o tempo máximo de conexão ao módulo de memória i e existe um outro processador solicitando o uso desse barramento. Na Figura 24, os nós que conectam os processadores aos seus respectivos barramentos encontram-se destacados.

2.1 O processador da rede

O processador da rede é responsável pela execução do programa, armazenado no módulo de memória. Esse componente é dividido em duas partes: o núcleo, denominado MLite_CPU, e o controle de acesso ao barramento.

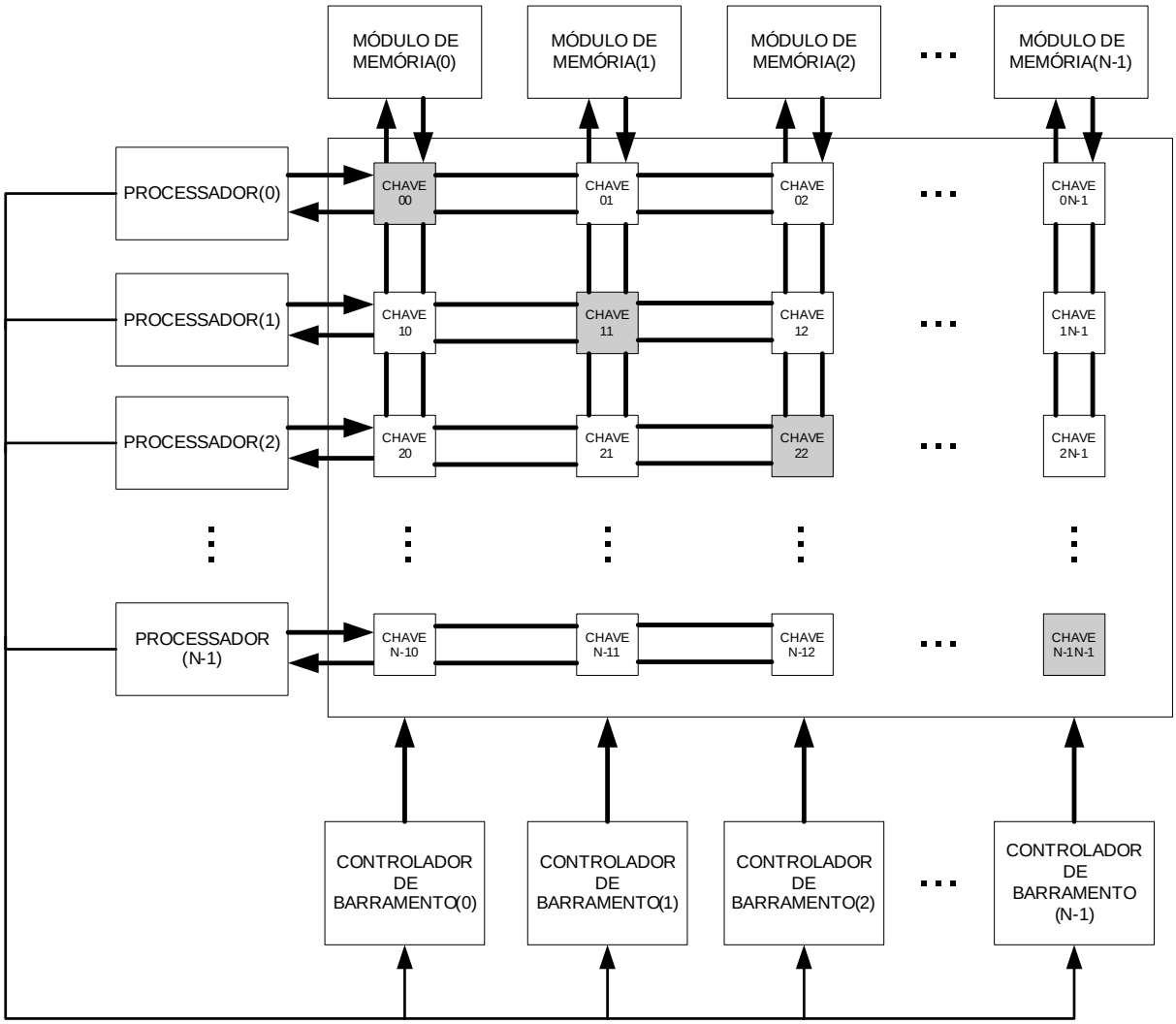


Figura 24: Modelo da rede

2.1.1 MLite_CPU

O MLite_CPU consiste do núcleo do processador PLASMA (RHOADS, 2007), cuja a arquitetura é baseada no processador MIPS (PATTERSON; HENNESSY, 2009). A escolha pelo PLASMA foi feita devido a disponibilidade de seu código VHDL (OPENCORES, 1999), viabilizando, se necessário, adequações de sua arquitetura ao projeto.

O MLite_CPU executa todas as instruções presentes no conjunto de instruções MIPS I, exceto operações com ponto flutuante. O coprocessador aritmético não foi implementado e também não foram implementadas instruções de *load* e *store* não alinhadas.

Este núcleo foi implementado com o *pipeline* diferente do MIPS clássico possuindo 2 ou 3 estados que podem ser configurados através do código VHDL do processador. Além destes citados existe um estágio extra para operações de *load* e *store*.

O MLite_CPU pode endereçar 2^{30} posições de memória, entretanto os *bits* 29 a

$29 - (log_2 N - 1)$ do endereço são utilizados para selecionar o módulo de memória j. Na Tabela 1 temos um exemplo de como, a partir dos *bits* de endereço, o PROCESSADOR(i) endereça os módulos de memória para uma rede com 4 processadores. Dessa forma, com $N = 4$ e $log_2 N = 2$, os *bits* ADDR(29:28) são os responsáveis por endereçar o módulo de memória.

Tabela 1: Conexões dos processadores com os módulos de memória para a rede com $N = 4$

ADDR(29:28)	Módulo
00	memória(0)
01	memória(1)
10	memória(2)
11	memória(3)

O contador de programa do MLite_CPU é inicializado em zero, entretanto cada módulo de memória possui uma faixa de endereço, com os *bits* mais significativos deste conforme mostrados na Tabela 1. Dessa forma, foi necessário alterar o contador de programa de MLite_CPU acrescentando um *offset* em sua saída. Este *offset* é composto do índice i do processador, deslocado à esquerda de $29 - (log_2 N - 1)$ *bits*, somado ao valor real do contador de programa. Com isso, o contador de programa de cada processador da arquitetura apontará para o endereço do módulo de memória onde está o seu respectivo programa.

Na Figura 25 estão os sinais de interface de MLite_CPU e na Tabela 2 encontra-se a descrição destes sinais. O sinal MEM_PAUSE, faz com que o contador de programa de processador MLITE_CPU pare de ser incrementado. Com isso, o processador para de buscar instruções, interrompendo portanto o processamento. O sinal de BYTE_WE indica quantos *bytes* serão escritos na memória principal e qual a posição deles dentro dos 32 *bits*, que é a largura do barramento de dados.

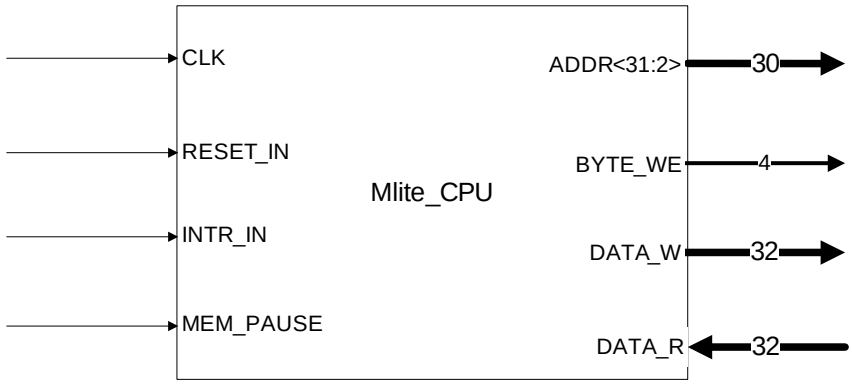


Figura 25: Sinais de interface de MLite_CPU

Tabela 2: Descrição dos sinais de MLite_CPU

Sinal	Número de bits	Tipo	Descrição
CLK	1	Entrada	Sinal de *clock*
RESET_IN	1	Entrada	Sinal de reset
INTR_IN	1	Entrada	Sinal de interrupção
MEM_PAUSE	1	Entrada	Suspende o funcionamento do processador
DATA_R	32	Entrada	Barramento de leitura
ADDR	30	Saída	Barramento de endereços
BYTE_WE	4	Saída	Habilita a Escrita de dados na memória
DATA_W	32	Saída	Barramento de escrita

2.1.2 O Controle de acesso ao barramento

O controle de acesso ao barramento é mostrado na Figura 26. A função deste componente é pausar o processador que não está conectado a nenhum barramento e decodificar os *bits* do endereço, fornecido pelo núcleo, a fim de solicitar uma requisição de uso do barramento correspondente ao módulo de memória endereçado (REQ(i,j)). Essa requisição é enviada para o controle de barramento, que determinará quando o processador terá acesso à memória.

O sinal BUS_REQ vai para '1' quando o endereço fornecido pelo núcleo corresponde a um módulo de memória em que $i \neq j$. Isto faz com que o núcleo pare até que o controlador do barramento correspondente conceda o uso do mesmo. Neste momento, o sinal DIS_EN é enviado pelo controlador, reativando o processador, conforme a Figura 26. Após utilizar o módulo de memória j, com $i \neq j$, obrigatoriamente MLite_CPU endereçará o seu próprio módulo, para buscar a próxima instrução a ser executada. Neste caso a saída do comparador vai para zero e consequentemente o sinal MEM_PAUSE vai para

'0', ativando o processador. Entretanto, o PROCESSADOR(i) está conectado ao módulo de memória j com $i \neq j$. Portanto antes de ativar o processador é necessário conectar PROCESSADOR(i) ao seu próprio módulo de memória. O *flip-flop*, permite que a saída permaneça em '1' até que o PROCESSADOR(i) seja conectado ao seu próprio módulo de memória. Quando isto ocorre, o controlador ativa o sinal CLEAR(i), tornando a saída BUS_REQ igual a zero e consequentemente MEM_PAUSE igual a zero.

O sinal PAUSE_CONT(i), fornecido pelo controlador do barramento, vai para '1' quando outro processador está requisitando o uso do barramento em que $i \neq j$. Dessa forma, o PROCESSADOR(i), com $i = j$, é pausado e, em seguida, desconectado desse barramento.

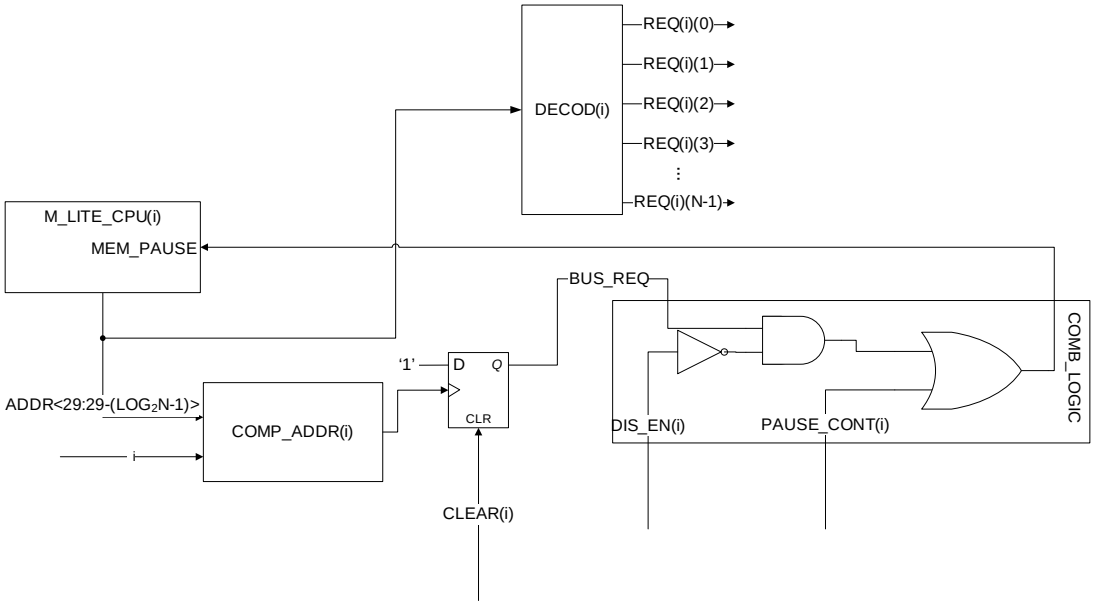


Figura 26: Controle de acesso ao barramento

2.2 A rede *crossbar*

A rede *crossbar* é uma matriz de interconexão que possui barramentos paralelos verticais e horizontais. No ponto de cruzamento destes barramentos temos uma chave que é fechada eletricamente. A rede *crossbar* proposta é composta de N barramentos, iguais ao mostrado na Figura 27, uma vez há N processadores, cada qual com seu respectivo módulo de memória.

A chave, que pode ser vista na Figura 28, e composta de quatro 4 portas *tris-tate* que englobam todos os sinais necessários para a comunicação do processador com o barramento.

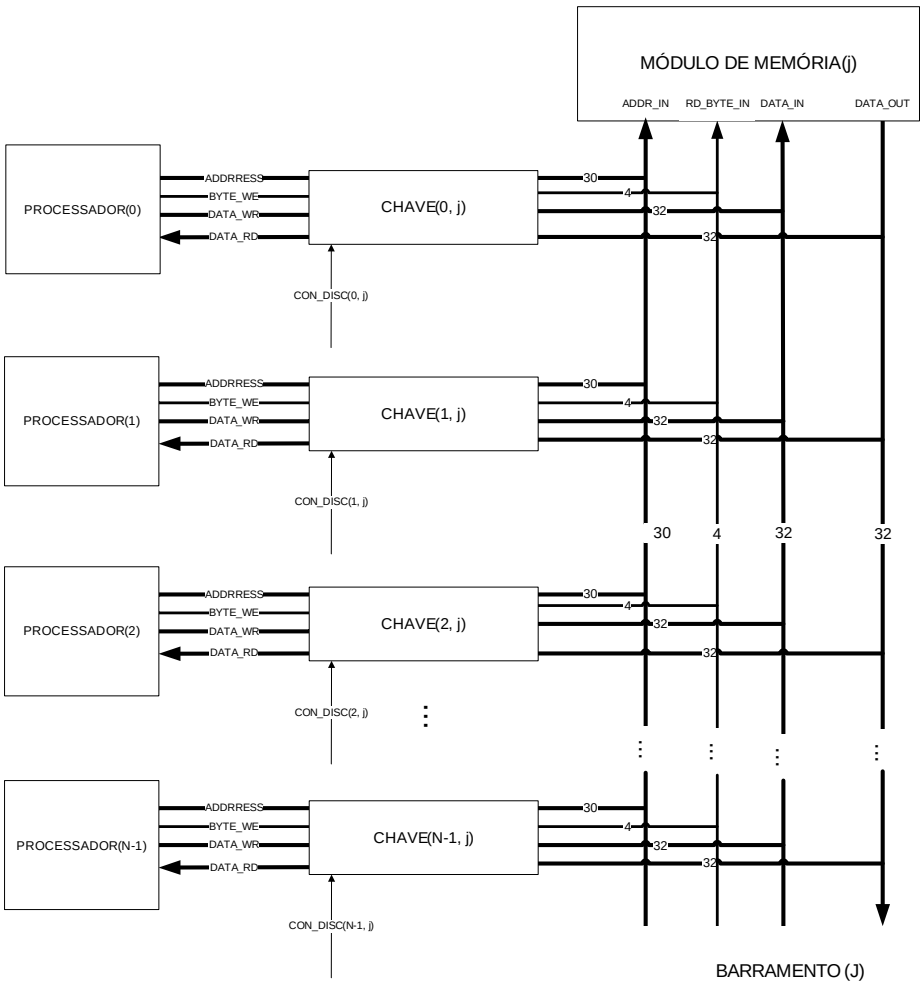


Figura 27: Barramento utilizado na Rede *crossbar*

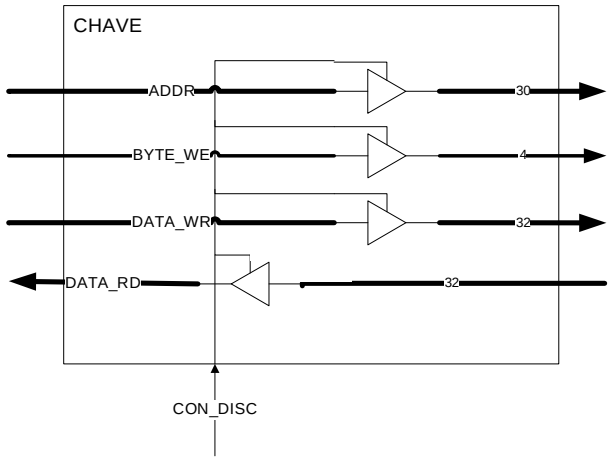


Figura 28: Descrição da chave utilizada na Rede *crossbar*

2.3 Controlador do barramento

O CONTROLADOR DE BARRAMENTO possui dois componentes principais: o árbitro denominado ARBITER e as máquinas de estados denominadas SM. Na arquitetura, existe um controlador de barramento para cada BARRAMENTO(j), perfazendo um total de N controladores, e N máquinas de estados para cada barramento em um total de N^2 máquinas de estados. A macroarquitetura do controlador é apresentada na Figura 29.

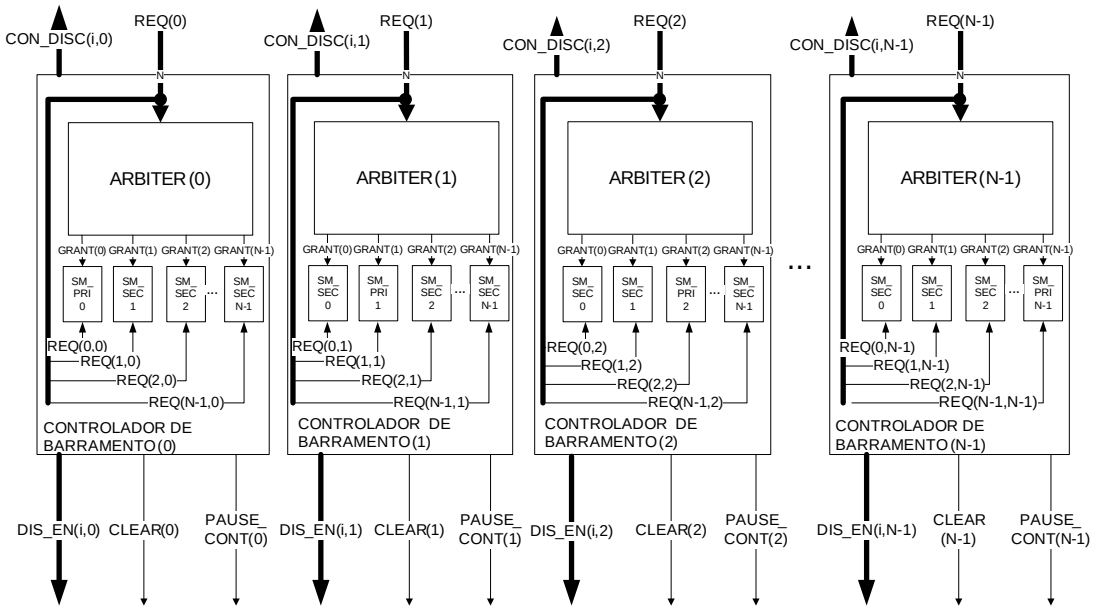


Figura 29: Macroarquitetura dos controladores

2.3.1 O árbitro

O árbitro tem a função de evitar que mais de um processador utilize um barramento em um mesmo intervalo de tempo. Como dito anteriormente, a rede *crossbar* da arquitetura proposta é composta por N barramentos. O conflito ocorre quando mais de um processador deseja usar um mesmo barramento ao mesmo tempo. O algoritmo utilizado neste árbitro é denominado *round-robin* (WEBER, 2001).

O circuito correspondente ao árbitro é dividido em duas partes: uma que implementa o algoritmo de arbitragem, denominada RR_ARBITER, e um contador, denominado CONTADORMOD_32, para implementar a verificação de requisições pendentes. O circuito de RR_ARBITER encontra-se na Figura 30.

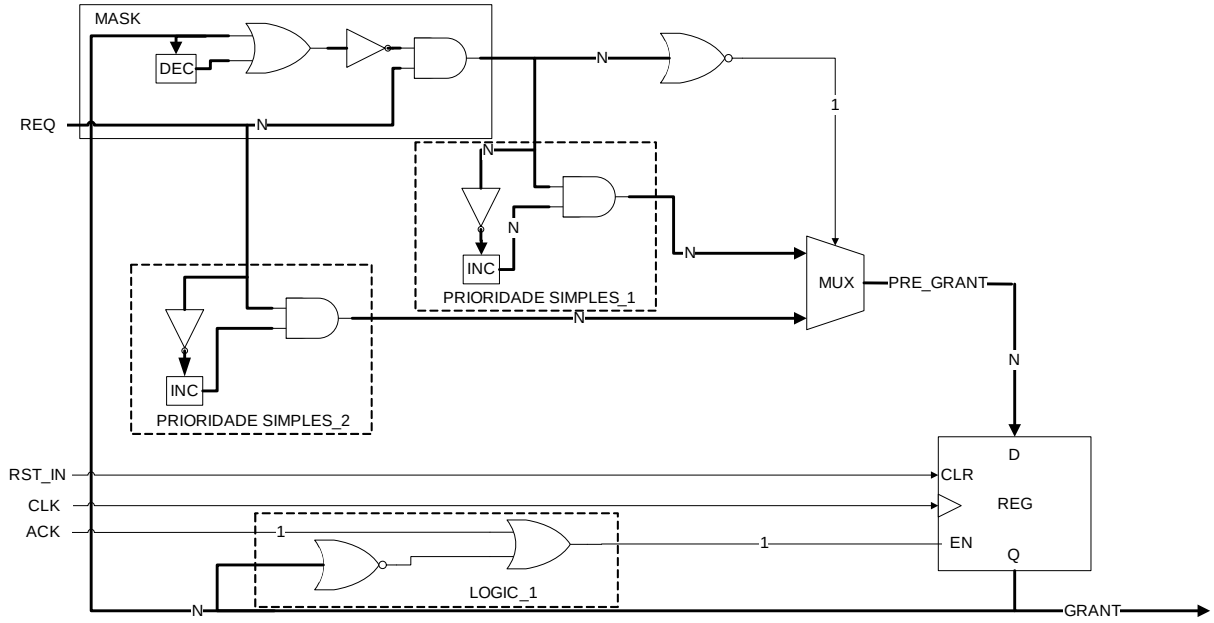


Figura 30: Circuito lógico que implementa o algoritmo *round-robin*

.

O circuito da Figura 30 possui dois subcomponentes denominados PRI_SIMPLES_1 e PRI_SIMPLES_2. Nestes subcomponentes, para cada solicitante é fixada uma prioridade. O vencedor será o requisitante, entre aqueles que estiverem ativos, com maior prioridade, sendo o *bit* zero o de maior prioridade. Este circuito possui como entradas, um vetor de requisição de N *bits*, designado REQ e, como saída, um vetor de concessão de N *bits* designado GRANT. No vetor de concessão apenas um *bit* pode ficar ativo em um mesmo intervalo de tempo. Por exemplo, se $N = 4$ e a entrada for igual a "0011", a saída será igual a "0001". O conjunto incrementador e porta inversora mantêm em '1' o *bit* associado ao requisitante de maior prioridade e coloca em '0' os *bits* associados aos outros requisitantes. A porta AND faz com que o circuito de prioridade simples tenha, em sua saída, apenas o *bit* de maior prioridade ativo em '1'. Um exemplo de funcionamento do circuito de prioridade simples pode ser visto na Figura 31. Nesta figura mostramos, para $N = 4$, como a partir de um vetor de requisição, neste caso igual a "0111", temos, na saída um vetor de concessão com apenas o *bit* de maior prioridade igual a '1' (saída $GRANT\_SEC =$ "0001").

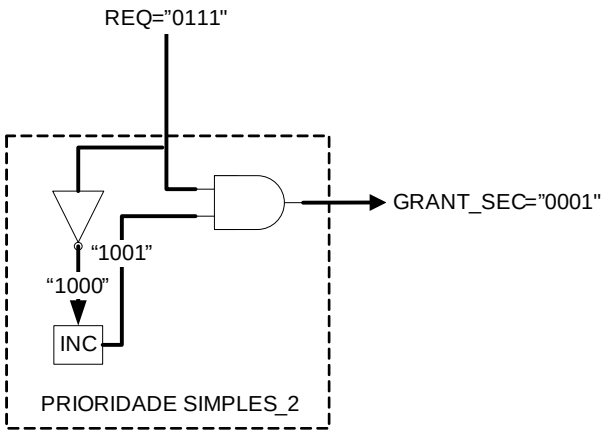


Figura 31: Exemplo de funcionamento do circuito PRI_SIMPLES_2

No algoritmo *round-robin*, vencedores anteriores não podem participar da arbitragem e o circuito MASK é responsável por garantir que isto ocorra. Este circuito possui um decrementador denominado DEC, que tem a função de mascarar o *bit* associado ao último ganhador, e circuitos auxiliares. Se há duas ou mais requisições e entre elas está incluída a vencedora anterior, o circuito MASK mascara esta requisição. Nesta situação a saída de MASK é diferente de zero, logo o *bit* de seleção do componente MUX é igual a '0' e a saída PRE_GRANT selecionada será a saída do circuito PRI_SIMPLES_1. Quando o vencedor anterior não está requisitando o uso do barramento ou quando ele já tem a prioridade menor que a dos outros requisitantes, o problema fica reduzido ao de prioridade simples, em que a saída de MASK é zero e o *bit* de seleção do componente MUX é '1'. Logo, a saída do MUX é a saída do circuito PRI_SIMPLES_2. Na Figura 32 estão expostas as duas situações possíveis para o circuito MASK. Em (a) temos um exemplo, para $N = 4$, da situação em que o algoritmo de arbitragem não pode ser reduzido a um problema de prioridade simples. Em (b) temos o exemplo em que um algoritmo de prioridade simples é capaz de resolver a arbitragem.

Se a entrada ACK do circuito da Figura 30 for igual a '1', EN='1' e a saída GRANT é atualizada com o valor do sinal PRE_GRANT. Em outras palavras, quando ACK='1' a saída GRANT associada ao atual vencedor vai para zero e a saída GRANT associada ao próximo requisitante vai para '1'. Se a saída GRANT for igual a zero a escrita no registrador também é habilitada. Este caso ocorre quando não houve requerentes no ciclo anterior, portanto não houve ganhadores. Então se no próximo ciclo existirem requerentes a escrita neste registrador estará habilitada para receber um ganhador entre os requerentes.

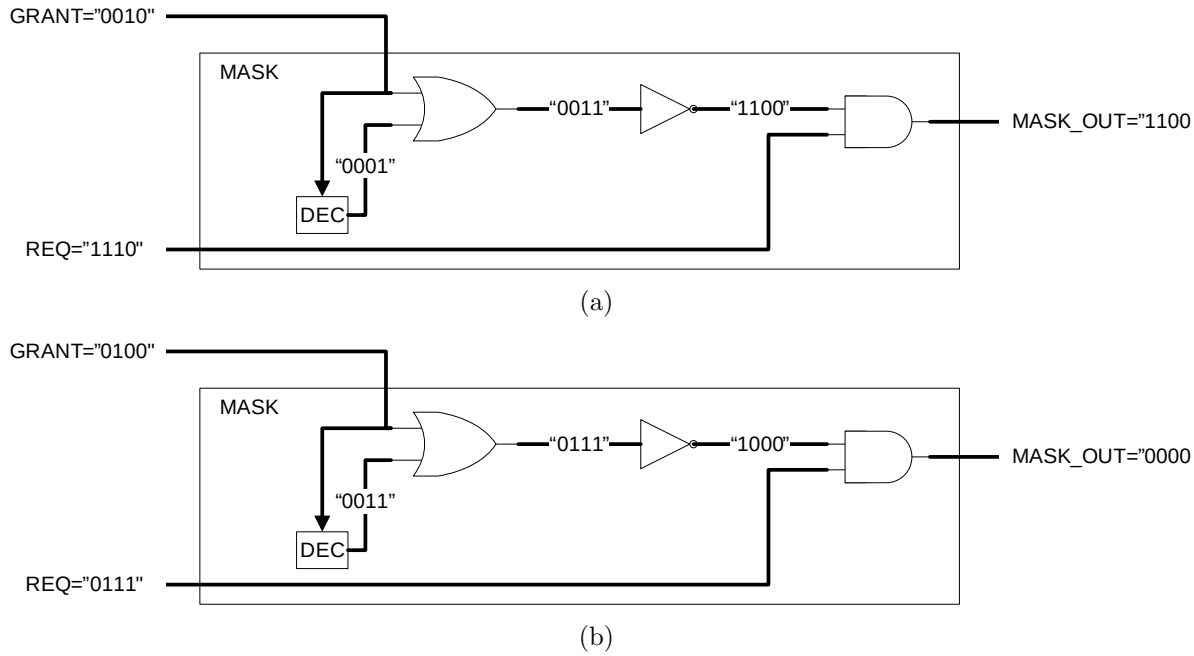


Figura 32: Exemplo de funcionamento do circuito MASK

O componente CONTADORMOD_32 funciona como um circuito de inspeção, verificando a cada 32 ciclos de *clock* se existem solicitações de conexão pendentes. Esta verificação é executada apenas quando o PROCESSADOR(i) está conectado ao BARRAMENTO(j), com $i = j$, pois os outros processadores utilizarão o barramento por somente dois ciclos de *clock*. Este é o tempo necessário para que, após atendida a requisição, a operação de acesso ao módulo de memória se complete. O sinal INIB_COUNT vem da máquina de estados e impede que seja realizada a verificação de conexões pendentes quando um PROCESSADOR(i) está conectado ao BARRAMENTO(j). O árbitro implementado no projeto encontra-se na Figura 33. Na Tabela 3, temos os sinais de interface do árbitro.

Um exemplo de uma situação onde os processadores competem pelo uso do BARRAMENTO(1) está descrita na Figura 34. A entrada REQ="1111", implica que todos os processadores estão solicitando o uso do barramento, e o PROCESSADOR(1) está conectado ao BARRAMENTO(1). Quando é realizada a verificação de requisições pendentes, a saída ACK_COUNT de CONTADORMOD_32 vai para '1' e a entrada ACK de RR_ARBITER vai para '1'. Neste momento, o árbitro concede ao PROCESSADOR(2) o uso do BARRAMENTO(1). Quando o PROCESSADOR(2) completa o acesso ao BARRAMENTO(1) o sinal ACK_ME vai para '1'. A partir daí o PROCESSADOR(2) não está mais solicitando o uso do BARRAMENTO(1). O PROCESSADOR(3) se conecta

ao BARRAMENTO(1) e assim por diante. Quando todos os requisitantes terminam de usar o BARRAMENTO(1), o PROCESSADOR(1) é conectado novamente ao BARRAMENTO(1).

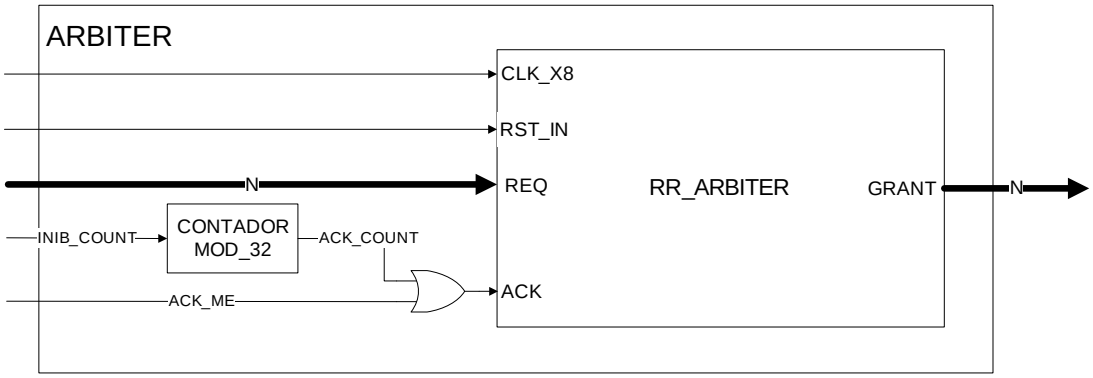


Figura 33: Árbitro *round-robin* implementado

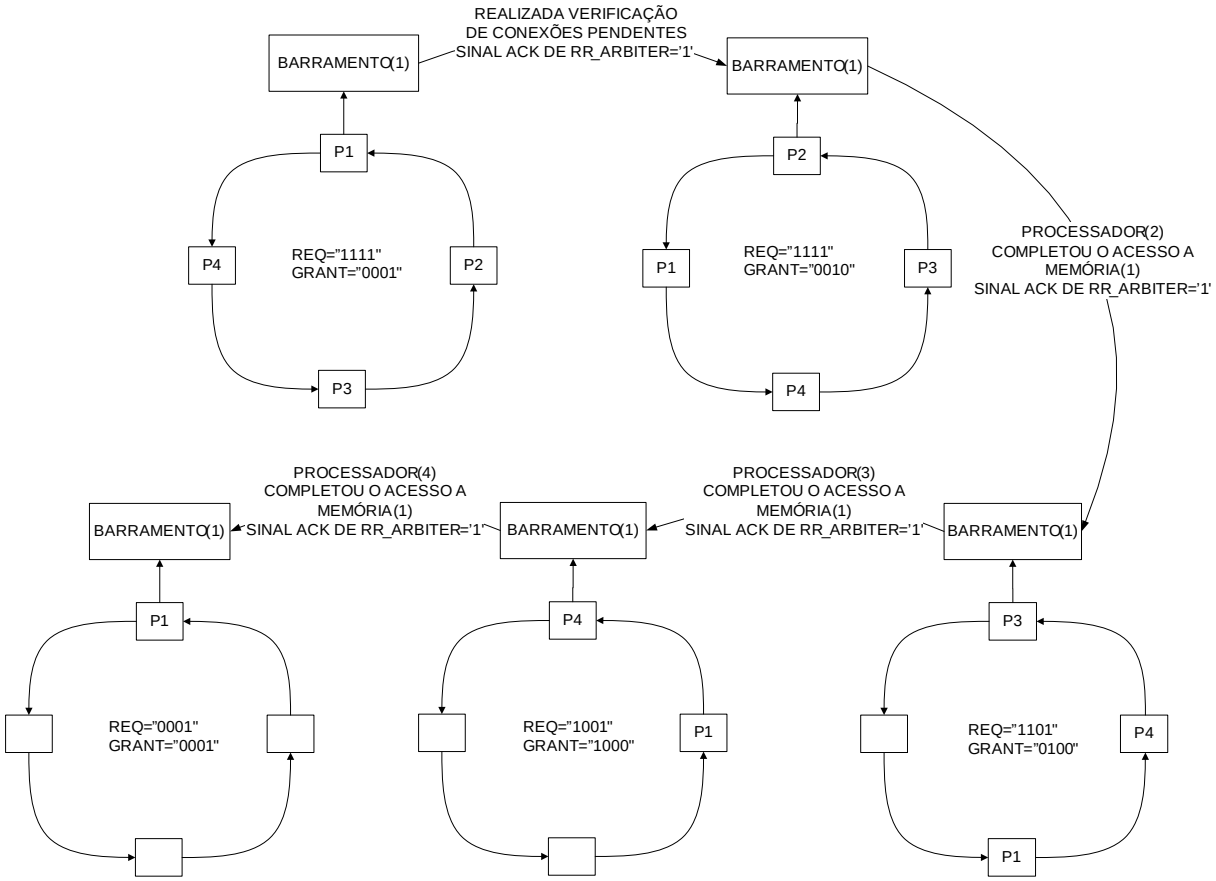


Figura 34: Algoritmo *round-robin*

Tabela 3: Sinais do *round-robin*

Sinal	N^o de bits	Tipo	Descrição
CLK_X8	1	Entrada	Sinal de *clock*
RST_IN	1	Entrada	Sinal de *reset*
REQ	N	Entrada	Sinal de requisição
INIB_COUNT	1	Entrada	Quando ativo desabilita a verificação de requisições pendentes
ACK_ME	1	Entrada	Atualiza a saída GRANT do árbitro
GRANT	N	Entrada	Sinal de concessão

2.3.2 As máquinas de estados

O árbitro fornece para o sistema um sinal de concessão denominado GRANT(i,j), entretanto este sinal não é suficiente para controlar todos os eventos necessários para conectar o PROCESSADOR(i) ao BARRAMENTO(j), por isso existiu a necessidade de implementarmos máquinas de estados. No projeto proposto temos uma máquina de estado por *switch* totalizando N^2 máquina de estados. Na arquitetura temos as máquinas de estados primárias, denominadas SM_PRI, são responsáveis por conectar o PROCESSADOR(i) ao BARRAMENTO(j), quando $i = j$. As máquinas de estados secundárias, denominadas SM_SEC, são responsáveis por conectar o PROCESSADOR(i) ao BARRAMENTO(j), quando $i \neq j$.

A justificativa para termos duas máquinas de estados diferentes no projeto é que a primária geralmente está conectando o processador ao barramento, pois o programa para ser executado pelo PROCESSADOR(i) está no módulo de memória j, quando $i = j$. Esta máquina desconecta o processador do barramento em duas situações: quando é realizada a verificação de conexões pendentes e existe outro processador desejando utilizar o barramento ou quando o PROCESSADOR(i) solicita a conexão com outro BARRAMENTO(j). A máquina de estados secundária conecta um PROCESSADOR(i) a outro BARRAMENTO(j), com $i \neq j$, apenas pelo tempo necessário para que o dado existente na MEMÓRIA(j) possa ser acessado pelo PROCESSADOR(i).

Na Figura 35 é apresentada a máquina de estados SM_PRI. Durante o *reset* do sistema a máquina de estados primária está no estado *Reset*. Neste estado todos os sinais da máquina de estados estão em zero e o PROCESSADOR(i) está inoperante. Quando o sinal de RESET do sistema vai para zero a máquina de estados vai para o estado *Con_1*. O sinal de RESET igual a zero também faz o PROCESSADOR(i) tornar-se ope-

rante, entretanto este processador está desconectado do BARRAMENTO(j). Neste caso o processador precisa ser pausado para se conectar ao barramento. No estado *Con_1*, SM_PRI pausa e conecta o PROCESSADOR(i) ao BARRAMENTO(j), com $i = j$, através dos sinais PAUSE_CONT(i) =‘1’ e CON_DISC(i,j)=‘1’, conforme pode ser visto no Algoritmo 1. No estado *Cont*, o sinal PAUSE_CONT(i) vai para zero habilitando o PROCESSADOR(i). O PROCESSADOR(i) permanece habilitado e conectado ao BARRAMENTO(j), com $i = j$, até que a verificação de requisições pendentes seja realizada ou o PROCESSADOR(i) solicite o uso de outro BARRAMENTO(j), com $i \neq j$.

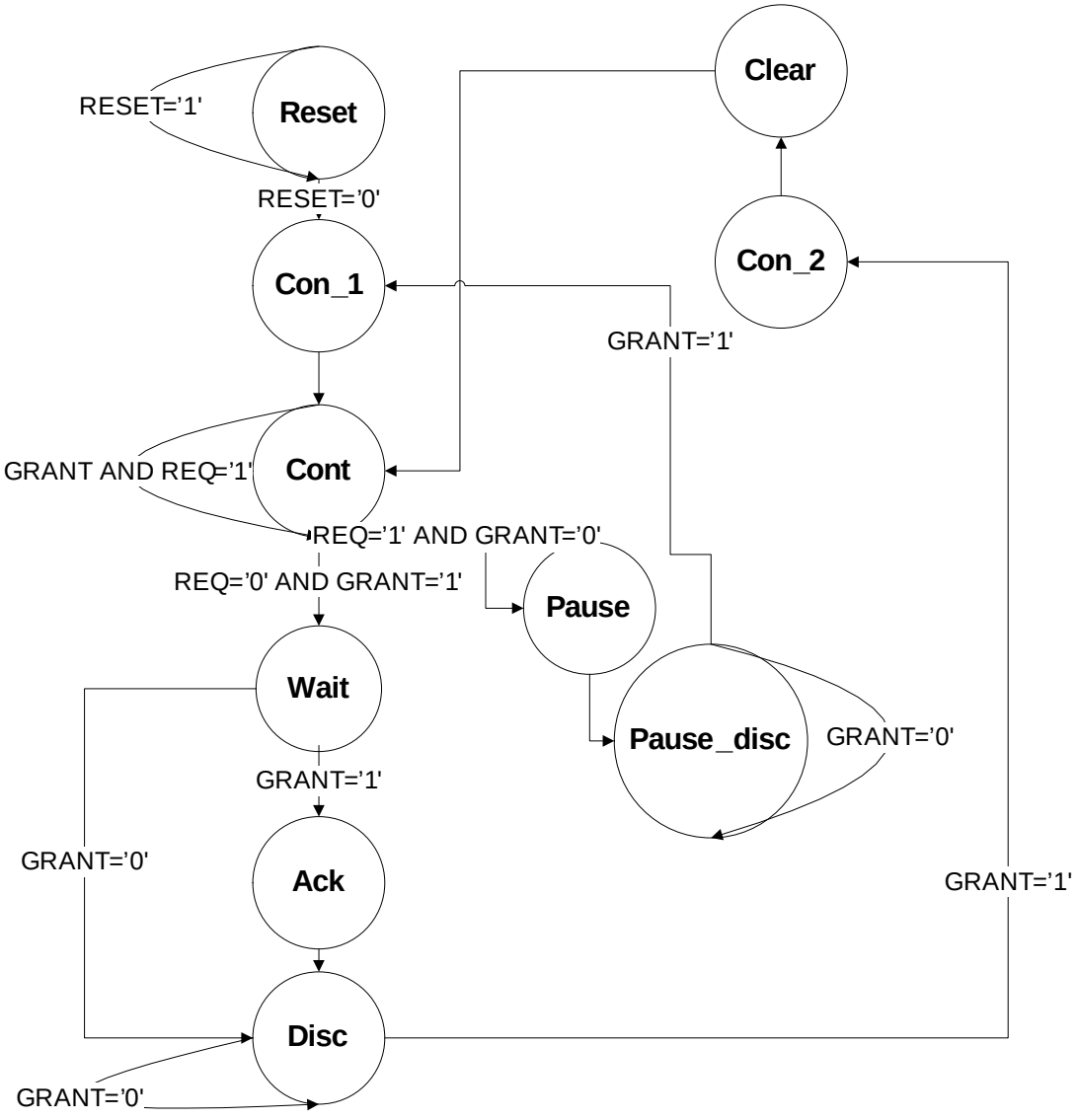


Figura 35: Estados da máquina de estados SM_PRI

Algoritmo 1 Máquina de estados SM_PRI

Reset: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 0$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;
se $RESET = 1$ **então** vá para **Reset**;

Con_1: $PAUSE\_CONT \leftarrow 1$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;

Cont: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;
se $GRANT = 1$ **e** $REQ = 0$ **então** vá para **Wait**;
se $GRANT = 0$ **e** $REQ = 1$ **então** vá para **Pause**;

Wait: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;
se $GRANT = 1$ **então** vá para **Ack senão** vá para **Disc**;

Ack: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 1$;

Disc: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 0$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;
se $GRANT = 0$ **então** vá para **Disc**;

Con_2: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;

Clear: $PAUSE\_CONT \leftarrow 0$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 1$; $ACK\_ME \leftarrow 0$;
vá para **Cont**;

Pause: $PAUSE\_CONT \leftarrow 1$; $CON\_DISC \leftarrow 1$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;

Pause_disc: $PAUSE\_CONT \leftarrow 1$; $CON\_DISC \leftarrow 0$; $CLEAR \leftarrow 0$; $ACK\_ME \leftarrow 0$;
se $GRANT = 0$ **então** vá para **Pause_disc senão** vá para **Con_1**;

Na primeira situação o sinal GRANT(i,j), com $i = j$, vai para '0' significando que a verificação de requisições pendentes foi realizada e existe outro processador solicitando o uso do barramento. Nesta situação a máquina de estados vai para o estado *Pause*. Neste estado o PROCESSADOR(i) é pausado, através do sinal PAUSE_CONT(i)='1'. No estado *Pause_disc* o PROCESSADOR(i) é desconectado do BARRAMENTO(j), com $i = j$, através do sinal CON_DISC(i,j)='0'. O processador é pausado e desconectado do barramento, porém continua requisitando o uso do mesmo através do sinal REQ(i,j). Quando o processador que requisitou o barramento termina de usá-lo, GRANT(i,j) para $i = j$ vai para '1' e a máquina de estados vai para o estado *Con_1*. No estado *Con_1* o PROCESSADOR(i) volta a se conectar ao BARRAMENTO(j), com $i = j$, através do sinal CON_DISC(i,j)='1'. No estado *Cont* sinal PAUSE_CONT(i) vai para zero. Dessa forma o PROCESSADOR(i) está apto a utilizar o BARRAMENTO(j).

Na segunda situação o sinal REQ(i,j), com $i = j$, vai para zero, isto significa que o PROCESSADOR(i) solicitou o uso de um BARRAMENTO(j) com $i \neq j$. Neste caso, a

máquina de estados verifica, no estado *Wait* se GRANT foi para zero. Esta situação pode ocorrer quando o PROCESSADOR(i) solicita o uso do BARRAMENTO(j) ao mesmo tempo em que a verificação de requisições pendentes é realizada (ACK_COUNT='1'). Neste caso GRANT(i,j), com $i = j$, no próximo subciclo, é desativado. Com isso, a concessão para o uso do barramento, será dada para o próximo requisitante não necessitando passar pelo estado *Ack*. Se, no estado *Wait*, GRANT(i,j) for igual a '1', significa que o sinal REQ(i,j) foi para zero em um ciclo diferente daquele onde é realizada a verificação de solicitações pendentes. Nesta situação, o sinal GRANT(i,j) do árbitro precisa ser atualizado, então a máquina de estados vai para o estado *Ack*. No estado *Ack* a saída ACK_ME(j) da máquina de estados vai para '1', desativando o sinal GRANT(i,j), com $i = j$, do árbitro, fornecendo a concessão para o próximo requisitante. Neste momento, se existir outro processador solicitando o uso do barramento, ele então é conectado, se não existir, o árbitro fica a disposição para liberar o acesso do BARRAMENTO(j) a um requisitante. No estado *Disc* o PROCESSADOR(i) é desconectado do BARRAMENTO(j) através do sinal CON_DISC(i,j)='0'. Quando o PROCESSADOR(i) volta a requisitar o BARRAMENTO(j), com $i = j$, o árbitro concede o uso do barramento ao processador. A máquina de estados vai para *Con_2* conectando o processador ao barramento através do sinal CON_DISC(i,j)='1'. Com o PROCESSADOR(i) conectado ao BARRAMENTO(j) SM_PRI vai para o estado *Clear*, onde o sinal CLEAR(i)='1' torna o processador operante. Na Figura 36 são apresentados os sinais de interface da máquina de estados SM_PRI.

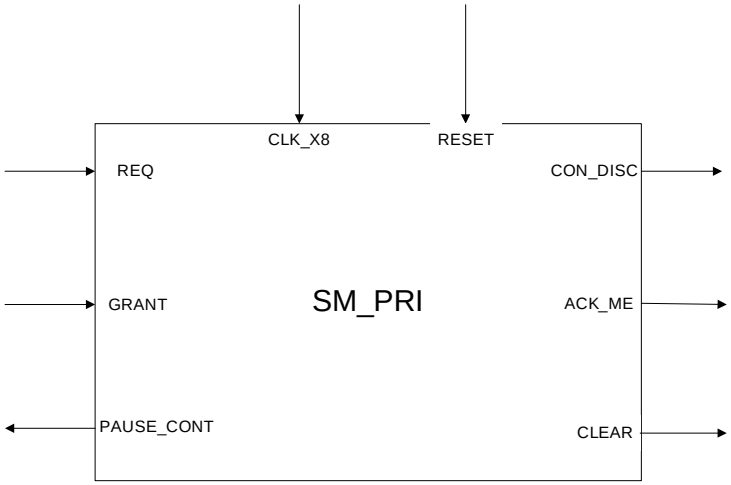


Figura 36: Sinais de interface da máquina de estados SM_PRI

Na Figura 37 é apresentada a máquina de estados SM_SEC. SM_SEC permanece no estado *Reset*, aguardando o sinal GRANT(i,j) de concessão do árbitro. Quando

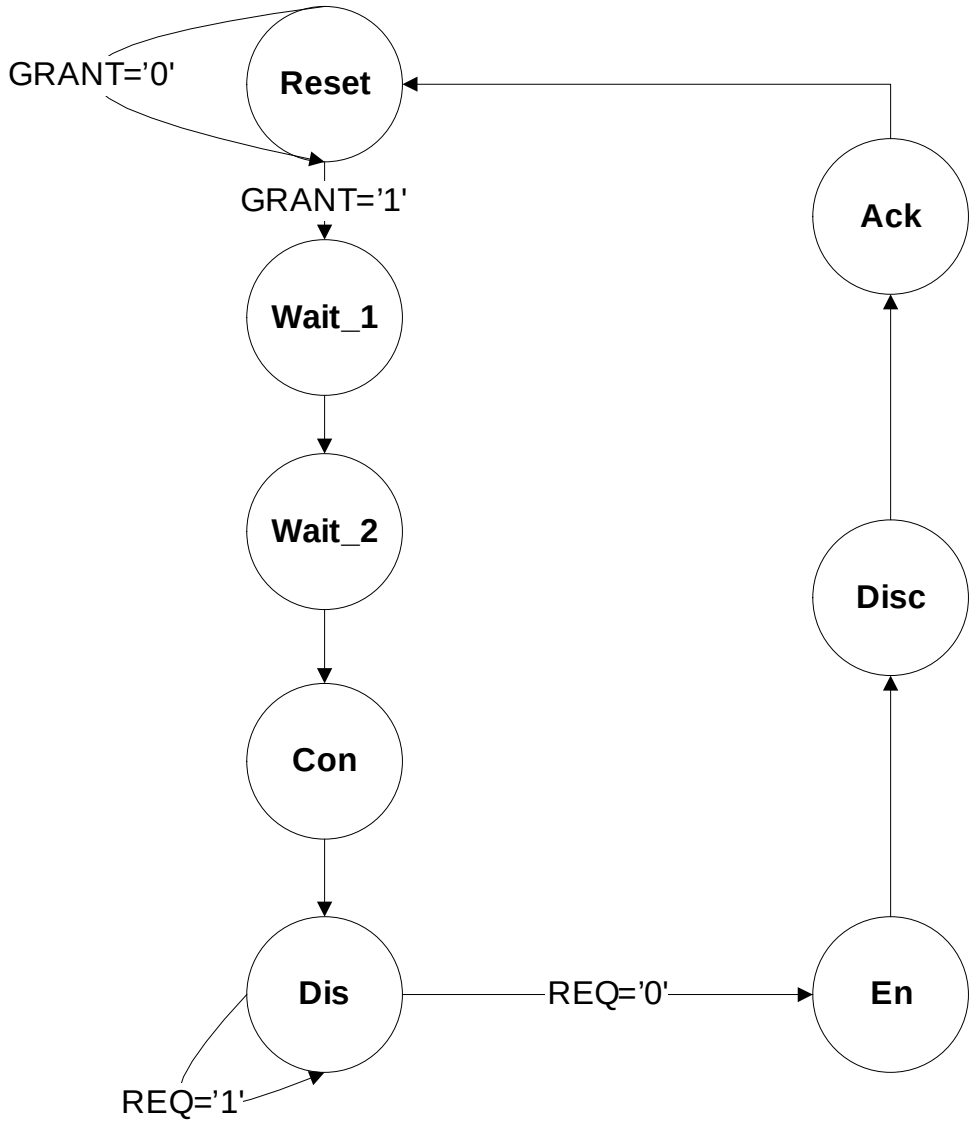


Figura 37: Estados da máquina de estados SM_SEC

GRANT(i,j) vai para '1', a máquina de estados vai para os estados *Wait_1* e *Wait_2*. Estes estados de espera foram implementados com o objetivo de dar tempo para SM_PRI(i) pausar o PROCESSADOR(i) e desconectá-lo do BARRAMENTO(j), para $i = j$. Após estes estados de espera SM_SEC vai para o estado *Con*, onde o PROCESSADOR(i) é conectado ao BARRAMENTO(j), com $i \neq j$, através do sinal CON_DISC(i,j)='1'. No estado *Dis*, DIS_EN(i,j) vai vai para '1' fazendo com que o sinal MEM_PAUSE do PROCESSADOR(i) vá para zero, de acordo com Figura 26. O processador permanece conectado ao barramento e ativado (sinal MEM_PAUSE='0') até que ele pare de requisitar o barramento, isto é, até que o sinal REQ(i,j), para $i \neq j$, vá para zero. Após isto ocorrer a operação de acesso a memória compartilhada está concluída e SM_SEC vai para o estado *En*. Neste estado o sinal DIS_EN(i,j) é igual a zero fazendo com que o PROCESSADOR(i)

seja pausado (MEM_PAUSE='1'), conforme Figura 26. A máquina de estados vai para o estado *Disc*, quando então o processador é desconectado do BARRAMENTO(j). Depois disto temos o estado *Ack*, onde o sinal ACK_ME é igual a '1', fazendo com que o árbitro verifique o próximo mestre do barramento. Se não houver mais solicitantes o PROCESSADOR(i) volta a usar o BARRAMENTO(j), para $i = j$. Em todos os estados da máquina de estados exceto no estado *Reset*, temos a saída INIB_COUNT(i,j)='1', impedindo que o sinal de saída CONT_MOD32 vá para '1', conforme a Figura 33, enquanto SM_SEC(i,j) está conectando o processador ao barramento. Isto impede que o árbitro conceda o uso do barramento ao próximo requisitante de maneira prematura, isto é, antes do PROCESSADOR(i) completar a operação de acesso ao módulo de memória j.

No Algoritmo 2 é apresentada a descrição da máquina em termos de suas saídas. Na Figura 38 são apresentados os sinais de interface da máquina de estados SM_SEC.

Algoritmo 2 Máquina de estados SM_SEC

Reset: $INIB\_COUNT \leftarrow 0$; $CON\_DISC \leftarrow 0$; $DIS\_EN \leftarrow 0$; $ACK\_ME \leftarrow 0$
se $GRANT = 0$ **então** vá para **Reset**;

Wait_1: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 0$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 0$

Wait_2: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 0$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 0$

Con: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 1$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 0$

Dis: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 1$; $EN\_DIS \leftarrow 1$; $ACK\_ME \leftarrow 0$
se $REQ = 1$ **então** vá para **Dis**;

En: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 1$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 0$

Disc: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 0$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 0$

Ack: $INIB\_COUNT \leftarrow 1$; $CON\_DISC \leftarrow 0$; $EN\_DIS \leftarrow 0$; $ACK\_ME \leftarrow 1$
vá para **Reset**;

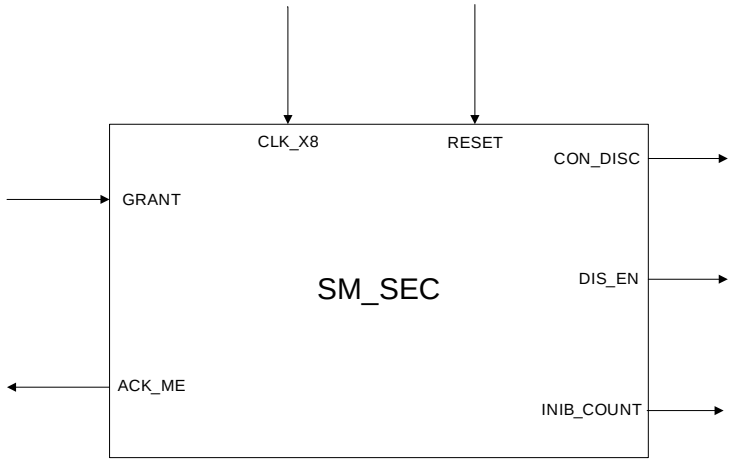


Figura 38: Sinais de interface da máquina de estados SM_SEC

A Figura 39 ilustra um exemplo do PROCESSADOR(0) solicitando o uso do BARRAMENTO(1). Nesta figura verifica-se como ocorre a interação das máquinas de estados SM_PRI(0,0), SM_PRI(1,1) e SM_SEC(0,1). Em um primeiro momento, o PROCESSADOR(0) solicita o uso do BARRAMENTO(1) através do sinal REQ(0,1). A verificação de requisições pendentes é realizada e o sinal ACK vai para '1'. Neste momento o árbitro concede o uso do BARRAMENTO(1) ao PROCESSADOR(0). A partir daí começa a sequência de eventos necessária para conectar o PROCESSADOR(0) AO BARRAMENTO(1). A máquina de estados SM_PRI(0,0) desconecta o PROCESSADOR(0) do BARRAMENTO(0), através do estado *Disc*. Simultaneamente a máquina de estados SM_PRI(1,1) pausa o PROCESSADOR(1) e o desconecta do BARRAMENTO(1), através dos estados *Pause* e *Pause_disc*. No próximo ciclo, a máquina de estados SM_SEC(0,1) conecta o PROCESSADOR(0) ao BARRAMENTO(1). Quando o acesso ao módulo de memória 1 é realizado o PROCESSADOR(0) é desconectado do BARRAMENTO(1) através do estado *Disc*. O PROCESSADOR(1) é conectado ao BARRAMENTO(1) através do estado **Con_1** e o PROCESSADOR(0) é reconectado ao BARRAMENTO(0) através do estado *Con_2*.

Na Figura 39 também identifica-se o *clock* dos processadores e dos controladores. Verifica-se que o *clock* dos controladores é oito vezes mais rápido que o dos processadores. Isto acelera a sequência de eventos necessários para a conexão de um processador com o barramento solicitado.

As conexões de interface entre a máquina de estados e o árbitro, para o controle do BARRAMENTO(2) com $N = 4$, são mostradas na Figura 40. Na Figura 41 podemos

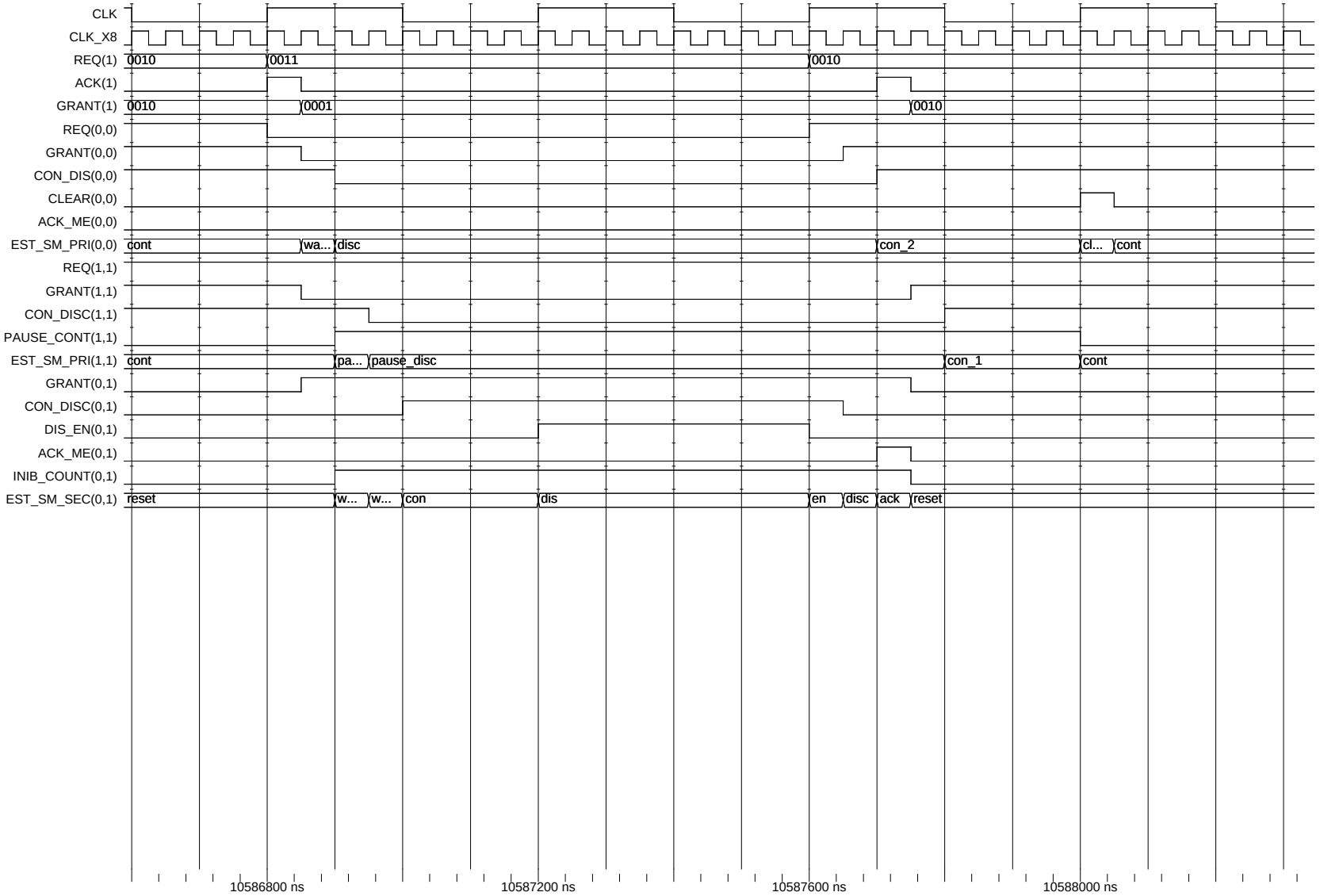


Figura 39: Interação entre as máquinas de estados envolvidas na conexão do PROCESSADOR(0) com o BARRAMENTO(1)

verificar como estão arranjados os CONTROLADORES DE BARRAMENTO para $N = 4$. Temos também, ilustrado nesta figura, como é composta a saída DIS_EN que alimenta os controles dos processadores.

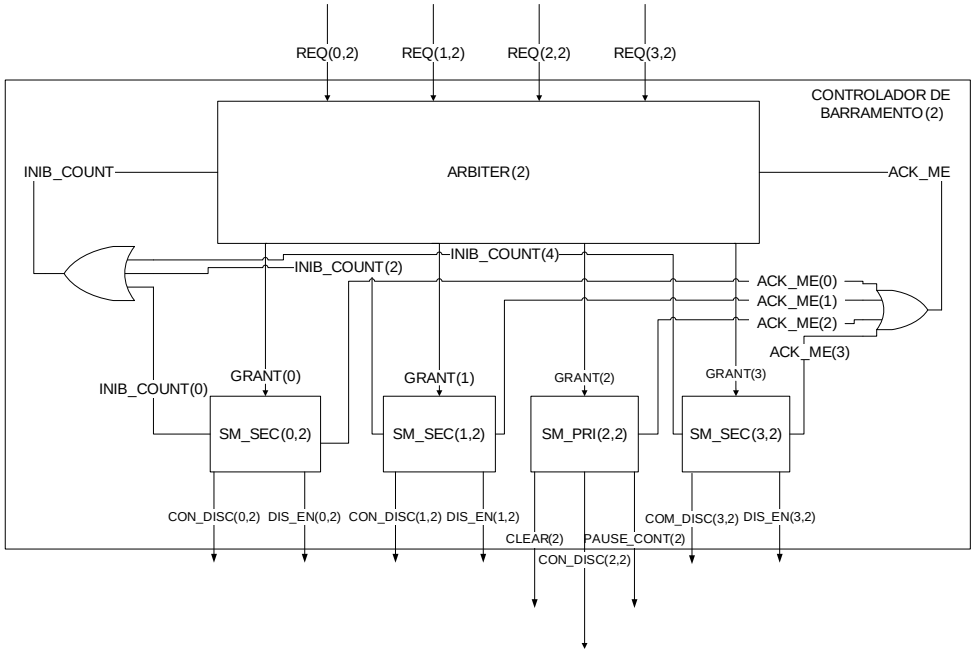


Figura 40: Sinais de interface do árbitro e das máquinas de estados para o controle do barramento 2, com $N = 4$

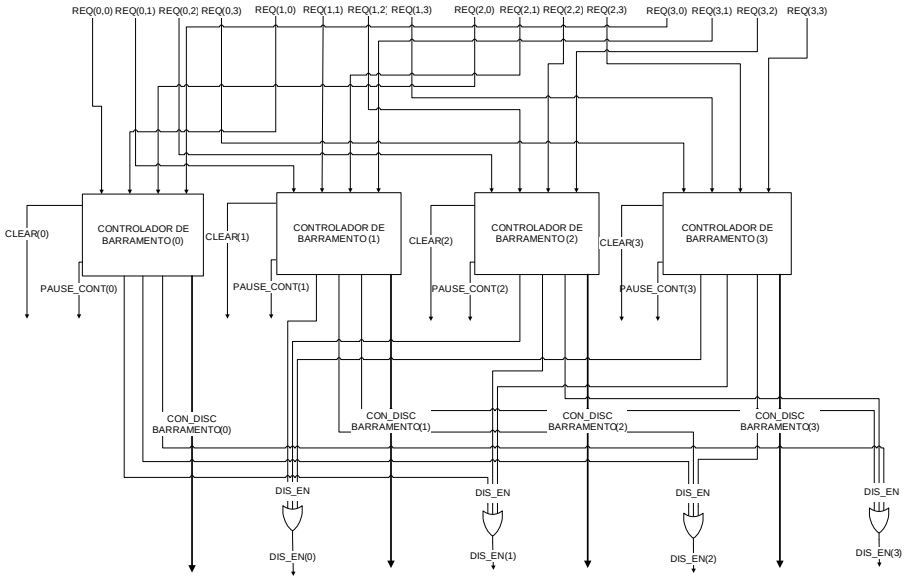


Figura 41: Controlador da rede para $N = 4$

2.4 O funcionamento da rede

Na Figura 42 podemos verificar a simulação de funcionamento da rede com 4 processadores. Na Figura 43 é mostrado os sinais de arbitragem para o BARRAMENTO(0). Nesta simulação temos os 4 processadores da rede solicitando o uso do BARRAMENTO(0), isto pode ser verificado através dos sinais ADDR<29:28>="00" dos processadores. Verifica-se que o sinal MEM_PAUSE dos três processadores 1, 2 e 3 foi para '1', indicando que os mesmos estão pausados e aguardando o sinal de GRANT do árbitro. O PROCESSADOR(0), em um primeiro momento, é mestre do BARRAMENTO(0), conforme pode ser verificado na Figura 43 (GRANT(0)="0001"). Após isso o sinal ACK do árbitro vai para 1, implicando que a verificação de conexões pendentes foi realizada. Neste momento GRANT(0)="0010", logo o PROCESSADOR(0) não é mais o mestre do BARRAMENTO(0). Os sinais MEM_PAUSE(0) e CON_DISC(0,0) vão para zero pausando e desconectando o PROCESSADOR(0) do BARRAMENTO(0). Através de GRANT(0)="0010", o PROCESSADOR(1) torna-se mestre do BARRAMENTO(0). Este processador é conectado ao barramento, através de CON_DISC(1,0)='1' e ativado (MEM_PAUSE(1)='0'). Quando o PROCESSADOR(1) completa o acesso ao BARRAMENTO(0), o mesmo precisa se reconectar ao BARRAMENTO(1) para buscar a próxima instrução, sendo então pausado através do sinal MEM_PAUSE(1)='1' e desconectado do barramento através do sinal CON_DISC(1,0). O sinal CON_DISC(1,1) vai para '1' e, com isso, o PROCESSADOR(1) é reconectado ao BARRAMENTO(1). Na Figura 43 vemos que o sinal REQ(1,0) vai para 0, indicando que o processador não está mais requisitando o uso do barramento e o sinal ACK vai para '1', indicando que o próximo requisitante pode utilizar o barramento. O PROCESSADOR(2), que é o próximo requisitante, torna-se mestre do barramento (GRANT(0)="0100") e assim por diante até o PROCESSADOR(3) acessar o BARRAMENTO(0). Quando o PROCESSADOR(3) termina de usar o BARRAMENTO(0), o PROCESSADOR(0) volta a se conectar ao mesmo. O sinal CON_DISC(0,0) vai para '1', conectando o processador ao barramento e o sinal MEM_PAUSE(0) vai para zero reativando o PROCESSADOR(0).

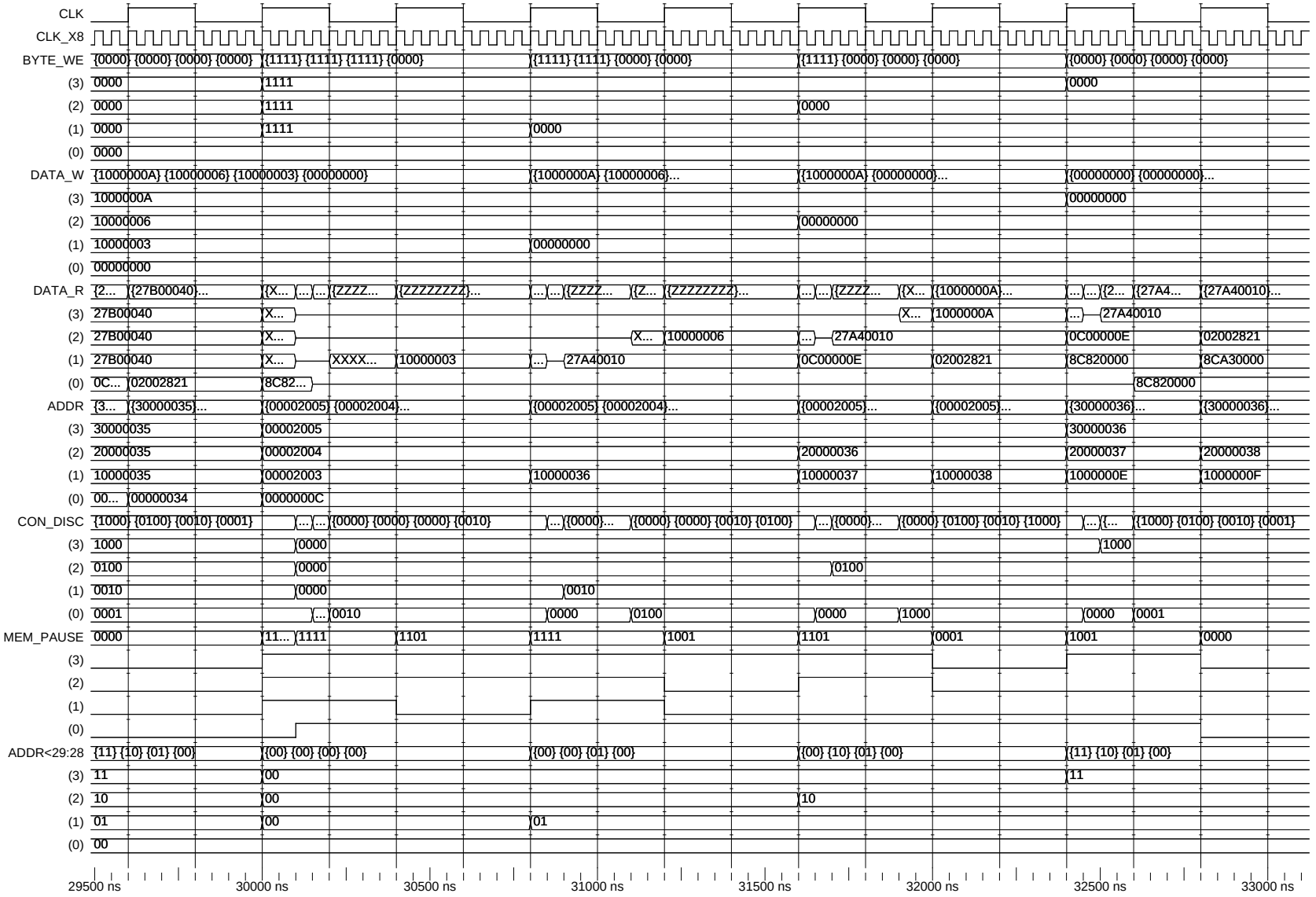


Figura 42: Processadores 1, 2, 3 e 4 solicitando o uso do BARRAMENTO(0)

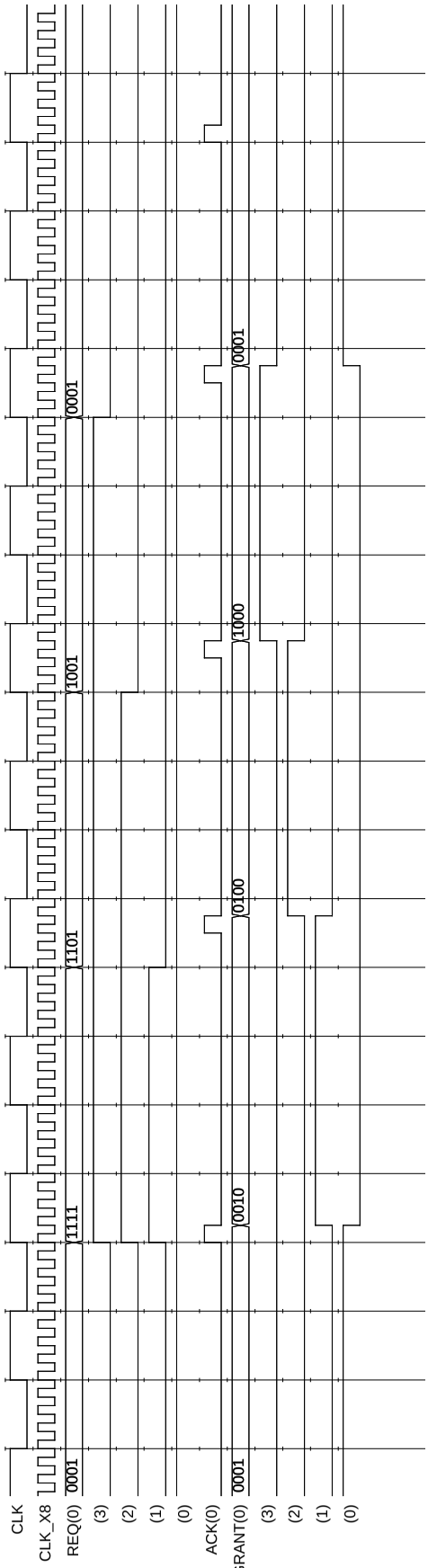


Figura 43: Arbitragem atendendo solicitação de conexão dos processadores 1, 2, 3 e 4 ao BARRAMENTO(0)

2.5 Considerações finais do capítulo

Neste capítulo descrevemos a arquitetura da rede implementada. Cada processador faz acesso a um módulo de memória compartilhada via um barramento, cuja conexão é estabelecida através de uma chave. A chave de interconexão foi definida, simplesmente, como um conjunto de portas *tristate*. O componente mais elaborado é o controlador da rede, que determina que processador será o mestre do barramento e fornece os sinais necessário para a conexão deste com o módulo de memória requerido. Dessa forma, temos N controladores da rede, sendo um para cada barramento. Cada controlador da rede é constituído de um árbitro e N máquinas de estados, cada qual responsável por emitir os sinais de controle necessários para o respectivo processador e a chave de interconexão, a fim de estabelecer a conexão processador-memória requisitada. O modelo foi validado funcionalmente através de simulação e diagramas de tempos foram apresentados para alguns exemplos de comunicação entre processador e módulo de memória. No capítulo 5, executaremos uma aplicação paralela, a fim de analisarmos o desempenho oferecido pela rede, em comparação com a respectiva execução sequencial.

Capítulo 3

AVALIAÇÃO DE DESEMPENHO

NESTE capítulo são analisados os resultados de algumas simulações, com o objetivo de testarmos e avaliarmos o desempenho da arquitetura proposta. Para cada simulação, serão observados os seguintes fatores. Tempo de execução, *speedup* e eficiência. Nestas simulações utilizaremos um PSO paralelo e avaliaremos o desempenho utilizando as topologias de migração anel, vizinhança e *broadcast*.

3.1 Infraestrutura de *software*

Como mencionado no capítulo anterior a arquitetura escalável proposta possui N processadores e N módulos de memória. Para validação e análise de desempenho da arquitetura uma aplicação deverá ser particionada em N partes, de maneira que estas partes possam ser executadas em paralelo. Cada parte da aplicação executada em paralelo é denominada processo (TANENBAUM, 2007). Nesta dissertação, consideramos que a aplicação está particionada em processos e que cada processador da rede está com um processo alocado em seu módulo de memória.

Em alguns momentos, durante a execução da aplicação, é necessária a comunicação entre os processos, para troca de informações. Isto é realizado através da memória compartilhada. Cada módulo de memória tem uma região de variáveis compartilhadas. Nesta região estão todas as variáveis que o processo i, que está alocado no módulo de memória i, está compartilhando com outros processos. Por exemplo, o PROCESSADOR(1) que está executando o processo 1 necessita compartilhar uma variável com outros processos. O PROCESSADOR(1) cria então uma cópia desta variável na região de variáveis compartilhadas do módulo de memória 1, para que outros processos possam acessá-la. Se outro processador necessitar deste dado gerado pelo PROCESSADOR(i), deverá realizar

uma leitura na região de variáveis compartilhadas do módulo de memória i. Por exemplo se o PROCESSADOR(1), precisar de um dado gerado pelo PROCESSADOR(3), o PROCESSADOR(1) deverá realizar uma operação de leitura no módulo de memória 3. As funções utilizadas para cópia de um dado para a região compartilhada e para leitura de um dado da região compartilhada são MOVE_RC e RD_RC.

A Função $MOVE\_RC(*a, b)$ é responsável por mover os dados para a região compartilhada da memória. O ponteiro $*a$ aponta para o endereço de memória do dado a ser movido e a variável b indica a posição da região de variáveis compartilhadas para onde será movido o dado.

A função $RD\_RC(*a, c, b)$ lê um dado da posição c da região de variáveis compartilhadas do módulo de memória b e a variável existente no endereço $*a$ recebe este dado.

Para gerarmos o código executável em cada processador da rede, utilizamos um compilador cruzado. A função do compilador cruzado é gerar um código executável para uma plataforma diferente daquela onde o compilador é executado. Os compiladores cruzados são utilizados para gerar o código executável para sistemas que possuem recursos limitados de memória RAM e, por sua vez, não permitem abrigar seus próprios compiladores. O compilador cruzado utilizado neste trabalho foi o GCC-MIPS (GNU, 2009).

A aplicação escolhida para ser executada é o algoritmo de otimização por enxame de partículas (PSO). Este algoritmo pode ser paralelizado através da divisão do enxame, onde cada processador receberá uma parcela deste enxame.

3.2 Otimização por enxames de partículas

A otimização por enxame de partículas (*Particle Swarm Optimization* - PSO)(KENNEDY; EBERHART, 1995) é um método de otimização natural. Este método é baseado na interação e aprendizado social (ENGELBRECHT, 2006) de indivíduos, onde aqueles de maior sucesso influenciam os demais.

O algoritmo PSO mantém um exame de partículas onde cada uma representa uma solução em potencial para um dado problema. Estas partículas transitam em um espaço de busca onde podem ser encontradas as soluções para o problema em questão. Cada partícula tende a ser atraída para a região do espaço de busca onde foram encontradas as melhores soluções para o problema.

A posição de cada partícula é atualizada através do fator velocidade $v_i(t)$, de acordo com a Equação 1:

$$x_i(t+1) = x_i(t) + v_i(t+1). \tag{1}$$

Cada partícula possui sua própria velocidade. A velocidade da partícula dirige o processo de otimização guiando a partícula através do espaço de busca. Esta velocidade depende do seu próprio desempenho, denominado componente cognitivo, e da troca de informações entre a sua vizinhança, denominado componente social. O componente cognitivo quantifica o desempenho da partícula i em relação ao seu desempenho em iterações anteriores. Este componente é proporcional à distância entre a melhor posição, designada $Pbest_i$, encontrada pela partícula e a posição atual (ENGELBRECHT, 2006). O componente social quantifica o desempenho da partícula i em relação à sua vizinhança. Este componente é proporcional a distancia entre a melhor posição encontrada pelo enxame, designada $Gbest$, e a posição atual da partícula. Na Equação 2 temos a definição da velocidade atual do PSO em termos dos componentes cognitivo e social da partícula.

$$v_i(t+1) = v_i(t) * w(t) + c_1 * r_1(Pbest_i - x_i(t)) + c_2 * r_2(Gbest - x_i(t)) \tag{2}$$

Os componentes r_1 e r_2 controlam a aleatoriedade do algoritmo. As componentes c_1 e c_2 são chamados coeficiente cognitivo e coeficiente social, controlando a confiança nas componentes cognitiva e social da partícula. A maioria das aplicações utiliza $c_1 = c_2$, fazendo com que as componentes cognitiva e social coexistam em harmonia. Se $c_1 \gg c_2$, tem-se uma movimentação excessiva da partícula, dificultando a convergência. Se $c_2 \gg c_1$, pode-se ter uma convergência prematura do algoritmo, facilitando a convergência do algoritmo para um valor de mínimo local.

O componente w, na Equação 2 é chamado de coeficiente de inércia e define como a velocidade anterior influenciará na velocidade atual da partícula. O valor deste fator é importante para a convergência do PSO. Um valor baixo de w promove uma exploração local da partícula. Por outro lado, um valor alto promove uma exploração global do espaço. Em geral são utilizados valores próximos a 1, porém não muito próximos de 0. Valores de w maiores do que 1 deixam a partícula com uma aceleração muito alta, o que pode dificultar a convergência do algoritmo. Valores de w próximos a 0 podem fazer com que a busca se torne mais demorada, acarretando um custo computacional desnecessário.

Uma alternativa é atualizar a cada iteração o valor de w, de acordo com a Equação 3:

$$w(t+1) = w(t) - \frac{w(0)}{n_{ite}}, \tag{3}$$

onde n_{ite} é o número total de iterações. No início das iterações temos um $w \approx 1$, aumentando o caráter exploratório do algoritmo. Com o passar das iterações diminuímos linearmente w, fazendo com que o algoritmo realize uma busca mais refinada. Outros parâmetros do PSO são o tamanho do enxame e o número de iterações. O tamanho do enxame é o número de partículas existentes. Um número alto de partículas permite que mais partes do espaço de busca sejam inspecionadas por iteração. Em outras palavras faz com que melhores soluções sejam encontradas se comparadas com soluções encontradas por enxames pequenos. Entretanto aumentam o custo computacional, consequentemente aumentando o tempo de execução. O número de interações é dependente do problema, sendo que poucas iterações pode levar ao término do algoritmo prematuramente, fazendo com que não encontremos uma solução aceitável. Entretanto um número alto de iterações pode ocasionar um custo computacional desnecessário.

O Algoritmo 3 mostra a execução do PSO. Neste algoritmo podemos verificar três estágios: inicialização das partículas, determinação de $Pbest_i$ e $Gbest$, atualização da velocidade e posição da partícula.

Algoritmo 3 PSO

```
Crie e inicialize um enxame com n partículas;
Repita
  Para i = 1 → n Faça
    Calcula o fitness da particula_i;
    Se Fitness_i ≤ Pbest_i Então
      Atualiza Pbest_i com a nova posição;
    Fim Se;
    Se Pbest_i ≤ Gbest Então
      Atualiza Gbest com a nova posição;
    Fim Se;
    Atualiza a velocidade da partícula;
    Atualiza a posição da partícula;
  Fim Para;
  Verifica critério de parada;
Até Critério de parada = verdadeiro;
```

3.3 Comunicação entre os processos

Para análise de desempenho da arquitetura, executaremos o PSO paralelo, onde cada processador executará o PSO descrito no Algoritmo 3, utilizando uma parcela P_N da população. A parcela da população que cada processador irá executar é definida pela equação $P_N = P_t/N$, onde P_t é o total de partículas no enxame e N é o número de processadores existentes na arquitetura. Estes enxames evoluem de maneira independente e periodicamente a variável *Gbest* é trocada entre os processadores da rede. Esta troca é feita de acordo com a topologia de comunicação escolhida.

Neste trabalho utilizamos três topologias de comunicação: anel, vizinhança e *broadcast*. Na topologia de comunicação em anel, vista na Figura 44 o melhor indivíduo obtido pelo PROCESSADOR(i) é lido pelo vizinho da direita. A topologia de comunicação vizinhança é mostrada na Figura 45, onde o melhor indivíduo obtido pelo PROCESSADOR(i) é lido pelo vizinho da direita e da esquerda. A terceira topologia de comunicação é a *broadcast*, mostrada na Figura 46, onde o melhor indivíduo do PROCESSADOR(i) é lido por todos os processadores existentes na arquitetura.

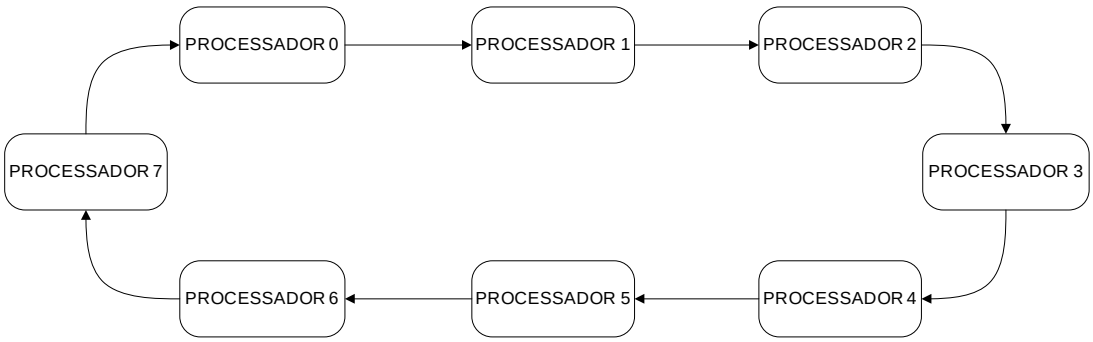


Figura 44: Topologia de comunicação em anel

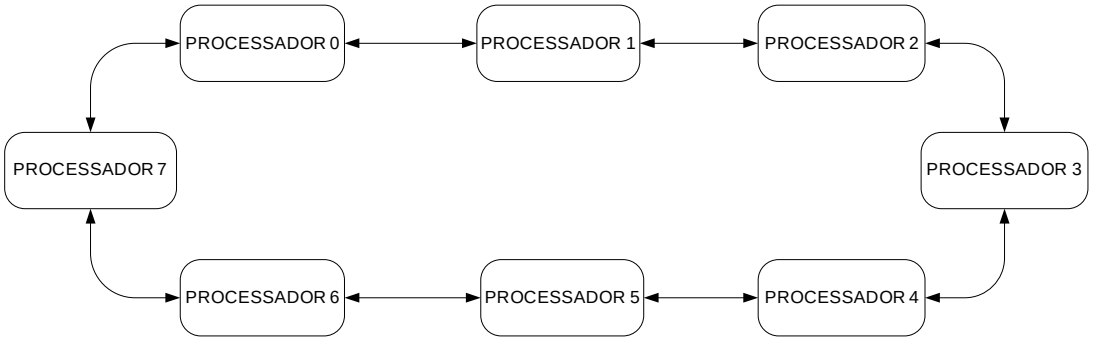


Figura 45: Topologia de comunicação em vizinhança

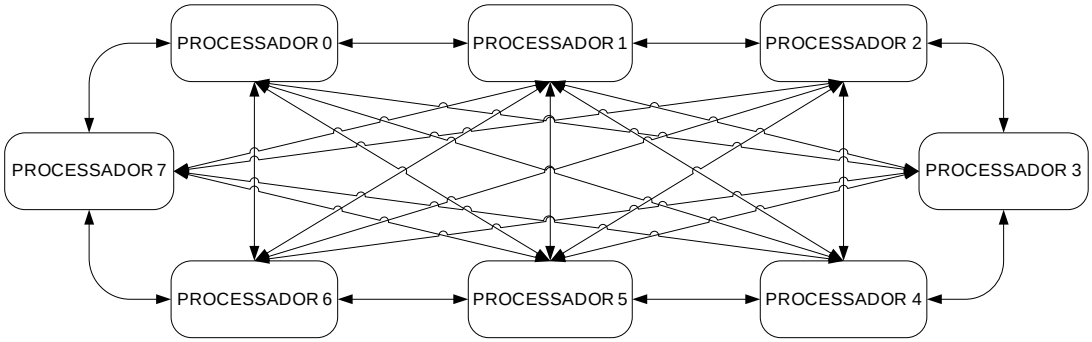


Figura 46: Topologia de comunicação *broadcast*

A variável *Gbest* representa a melhor partícula encontrada pelo PROCESSADOR(i). No final de cada iteração esta variável tem seu valor armazenado na região de variáveis compartilhadas do módulo de memória i. Neste momento esta variável pode ser acessada por outros processadores, de acordo com a topologia de comunicação escolhida. Os Algoritmos 4, 5 e 6 implementam estas topologias de migração. Nestes algoritmos a constante *id* representa o índice do processador na rede. A variável *tmp_id* indica o índice do módulo de memória em que o PROCESSADOR(id) realizará o acesso.

No Algoritmo 4 temos a implementação da topologia de comunicação em anel. Nesta topologia a variável *tmp_id* recebe o valor de *id* decrementado (linha 9), indicando que o PROCESSADOR(id) lê *Gbest* do seu vizinho da direita, ou seja, do PROCESSADOR(id-1). O Algoritmo 5 representa a topologia de migração vizinhança. Neste caso o PROCESSADOR(id) lê informações do PROCESSADOR(id+1) e do PROCESSADOR(id-1)(linhas 19 e 20, linhas 24 e 25). No caso da topologia *broadcast* o PROCESSADOR(id) lê informações de todos os processadores, conforme mostrado no Algoritmo 6.

O PROCESSADOR(0) é o processador responsável por computar os *Gbest* dos outros processadores e armazenar, na variável *Best*, o melhor valor encontrado. Cada processador ao terminar seu processo sinaliza tal situação através da variável $fim\_processo$. Quando a variável $fim\_processo = 0$, significa que o processador ainda não terminou de executar a seu processo e portanto não tem um valor de *Gbest* definitivo. A variável $fim\_processo = 1$ significa que o processador já terminou a execução de seu processo e portanto já tem um valor definitivo para *Gbest*. A indicação de término do processo serve para sinalizar ao PROCESSADOR(0) que um processo terminou de ser executado. O PROCESSADOR(0) não possui esta variável, pois não existe a necessidade do PROCESSADOR(0) sinalizar término de seu processo a nenhum outro processador da rede.

Ao final de suas iterações, o PROCESSADOR(0) executa uma inspeção nos outros processadores a fim de descobrir se estes já terminaram de executar os seus processos. Neste caso o PROCESSADOR(0) só lê a variável *Gbest* de outro processador quando a variável $fim\_processo$ deste processador for igual a '1'.

Algoritmo 4 Algoritmo PSO para a Comunicação em Anel

```
1: Crie e inicialize um enxame com n partículas;
2: Seja id:=identificação do processador;
3: Seja tmp_id índice do processador onde será realizada a leitura de Gbest;
4: Seja n_proc:=número de processadores da rede;
5: Se id ≠ 0 Então
6:    Seja fim_processo(id) indica final do processo;
7:    fim_processo(id):=0;
8: Fim Se;
9: tmp_id := id − 1;
10: Repita
11:    Para j = 1 → n Faça
12:       Cálculo da função fitness;
13:       Atualiza Gbest(id) e Pbest(id)
14:       Atualiza a velocidade da partícula;
15:       Atualiza a posição da partícula;
16:    Fim Para;
17:    Copia Gbest(id) para região compartilhada de memória;
18:    Lê Gbest do processador(tmp_id);
19:    Se Gbest(tmp_id) ≤ Gbest(id) Então
20:       Gbest(id):=Gbest(tmp_id);
21:    Fim Se;
22:    Verifica critério de parada;
23: Até Critério de parada = verdadeiro;
24: Se id = 0 Então
25:    Best := Gbest(id);
26:    tmp_id := id + 1;
27:    Para k = 1 → k < n_proc − 1 Faça
28:       Lê fim_processo(tmp_id);
29:       Enquanto fim_processo(tmp_id) = 0 Faça
30:          Lê fim_processo(tmp_id);
31:       Fim Enquanto;
32:       Lê Gbest do processador(tmp_id);
33:       Se Gbest(tmp_id) ≤ Best Então
34:          Best := Gbest(tmp_id);
35:       Fim Se;
36:       tmp_id := tmp_id + 1;
37:    Fim Para;
38: Senão
39:    fim_processo(id):=1;
40: Fim Se;
```

Algoritmo 5 Algoritmo PSO para a Comunicação em Vizinhança

```
1: Crie e inicialize um enxame com n partículas;
2: Seja id:=identificação do processador;
3: Seja tmp_id:=índice do processador onde será realizada a leitura de Gbest;
4: Seja n_proc:=número de processadores da rede;
5: Se id ≠ 0 Então
6:    Seja fim_processo(id):=indica final do processo;
7:    fim_processo(id):=0;
8: Fim Se;
9: Repita
10:   Para j = 1 → n Faça
11:      Cálculo da função fitness;
12:      Atualiza Gbest(id) e Pbest(id)
13:      Atualiza a velocidade da partícula;
14:      Atualiza a posição da partícula;
15:   Fim Para;
16:   Copia Gbest(id) para região compartilhada de memória;
17:   tmp_id := id + 1;
18:   Lê Gbest do processador(tmp_id);
19:   Se Gbest(tmp_id) ≤ Gbest(id) Então
20:      Gbest(id):=Gbest(tmp_id);
21:   Fim Se;
22:   tmp_id := id − 1;
23:   Lê Gbest do processador(tmp_id);
24:   Se Gbest(tmp_id) ≤ Gbest(id) Então
25:      Gbest(id):=Gbest(tmp_id);
26:   Fim Se;
27:   Verifica critério de parada;
28: Até Critério de parada = verdadeiro;
29: Se id = 0 Então
30:   Best := Gbest(id);
31:   tmp_id := id + 1;
32:   Para k = 1 → k < n_proc − 1 Faça
33:      Lê fim_processo(tmp_id);
34:      Enquanto fim_processo(tmp_id) = falso Faça
35:         Lê fim_processo(tmp_id);
36:      Fim Enquanto;
37:      Lê Gbest do processador(tmp_id);
38:      Se Gbest(tmp_id) ≤ Best Então
39:         Best := Gbest(tmp_id);
40:      Fim Se;
41:      tmp_id := tmp_id + 1;
42:   Fim Para;
43: Senão
44:   fim_processo(id):=1;
45: Fim Se;
```

Algoritmo 6 Algoritmo PSO para a Comunicação *Broadcast*

```
1: Crie e inicialize um enxame com n partículas;
2: Seja id:=identificação do processador;
3: Seja tmp_id:=índice do processador onde será realizada a leitura de Gbest;
4: Seja n_proc:=número de processadores da rede;
5: Se id ≠ 0 Então
6:    Seja fim_processo(id):=indica final do processo;
7:    fim_processo(id):=0;
8: Fim Se;
9: Repita
10:   Para j = 1 → n Faça
11:      Cálculo da função fitness;
12:      Atualiza Gbest(id) e Pbest(id)
13:      Atualiza a velocidade da partícula;
14:      Atualiza a posição da partícula;
15:   Fim Para;
16:   Copia Gbest(id) para região compartilhada de memória;
17:   tmp_id := id + 1;
18:   Para k = 1 → k < n_proc − 1 Faça
19:      Lê Gbest do processador(tmp_id);
20:      Se Gbest(tmp_id) ≤ Gbest(id) Então
21:         Gbest(id):=Gbest(tmp_id);
22:      Fim Se;
23:      Se tmp_id = n_proc − 1 Então
24:         tmp_id := 0;
25:      Senão
26:         tmp_id := tmp_id + 1;
27:      Fim Se;
28:   Fim Para;
29:   Verifica critério de parada;
30: Até Critério de parada = verdadeiro;
31: Se id = 0 Então
32:   Best := Gbest(id);
33:   tmp_id := id + 1;
34:   Para k = 1 → k < n_proc − 1 Faça
35:      Lê fim_processo(tmp_id);
36:      Enquanto fim_processo(tmp_id) = 0 Faça
37:         Lê fim_processo(tmp_id);
38:      Fim Enquanto;
39:      Lê Gbest do processador(tmp_id);
40:      Se Gbest(tmp_id) ≤ Best Então
41:         Best := Gbest(tmp_id);
42:      Fim Se;
43:      tmp_id := tmp_id + 1;
44:   Fim Para;
45: Senão
46:   fim_processo(id):=1;
47: Fim Se;
```

3.4 Métricas para avaliação de desempenho

Para avaliarmos o desempenho utilizamos como métrica o *speedup*, definido pela lei de Amdahl (AMDAHL, 1967) e expresso pela Equação 4:

$$S = \frac{T_2}{T_1}, \tag{4}$$

onde T_1 é o tempo de execução de uma aplicação na arquitetura original e T_2 é o tempo de execução dessa mesma aplicação na arquitetura com melhorias.

Em sistemas multiprocessados, considerando T_1 como o tempo de execução da aplicação sequencial utilizando um processador, em uma situação ideal o tempo de execução da aplicação paralela utilizando N processadores seria T_1/N. Dessa forma o *speedup* ideal utilizando N processadores seria igual a N.

Eficiência é definida como a utilização média dos N processadores de um sistema multiprocessado. A relação entre *speedup* e eficiência é dada pela Equação 5 (EAGER; ZAHORJAN; LAZOWSKA, 1989):

$$E = \frac{S}{N}. \tag{5}$$

A grande maioria dos programas possui regiões que podem ser paralelizadas, entretanto outras regiões não podem. Portanto se aumentarmos o número de processadores o tempo gasto pela parcela paralelizável do programa será reduzido, mas o tempo gasto pela parte sequencial permanecerá o mesmo (AMDAHL, 1967). A diferença entre o *speedup* ideal e o *speedup* alcançado é, portanto, determinado pela parcela sequencial da aplicação Outros obstáculos para a obtenção do *speedup* ideal são a disputa por recursos compartilhados e o tempo necessário para comunicação entre processadores (EAGER; ZAHORJAN; LAZOWSKA, 1989).

3.5 Resultados de simulação

Para validar a arquitetura proposta foi implementado o algoritmo PSO, visando minimizar as funções objetivo apresentadas na Figura 47. A rede foi configurada com 1, 2, 4, 8, 16 e 32 processadores para cada simulação, considerando as topologias de comunicação anel, vizinhança e *broadcast*. Foram obtidos os valores de tempo consumido em todas as simulações com o intuito de calcularmos os valores de *speedup* e eficiência da rede. Os resultados obtidos foram organizados por topologia de comunicação e em seguida

estes resultados são apresentados, discutidos e comparados. A construção do modelo de arquitetura proposto, e as simulações apresentadas neste capítulo foram realizadas utilizando o software ModelSim (MODEL, 2012).

As três funções objetivo escolhidas para serem utilizadas no PSO são *Spherical*, *Rosenbrock* e *Rastringin*, definidas em (MOLGA; SMUTNICKI, 2005). A primeira função, denominada *Spherical*, é definida pela Equação 6:

$$f(x, y) = x^2 + y^2. \tag{6}$$

A curva correspondente é mostrada na Figura 47(a), com intervalo de interesse [-100, 100] e um ponto de mínimo global *f(0,0)=0.0*. A segunda função é denominada *Rosenbrock* definida pela Equação 7:

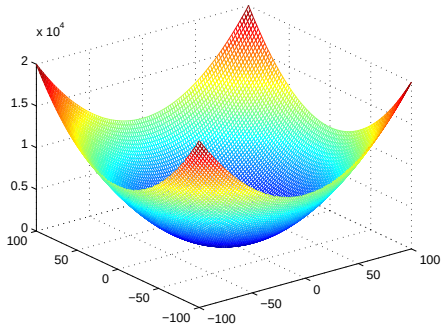
$$f(x, y) = 100(y - (x^2))^2 + (1 - x)^2. \tag{7}$$

A curva correspondente é mostrada na Figura 47(b) possui intervalo de interesse [-2.048, 2.048] e um ponto de mínimo global *f(1,1)=0.0*. A terceira função é denominada *Rastrigin*. Definida pela Equação 8:

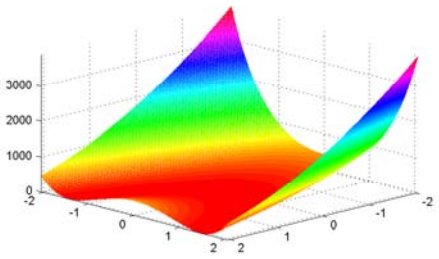
$$f(x, y) = 20 + x^2 + y^2 - 10cos(2\pi x) - 10cos(2\pi y), \tag{8}$$

para duas dimensões. A curva correspondente é mostrada na Figura 47(c) possui intervalo de interesse [-5.12, 5.12] e um ponto de mínimo global *f(0,0)=0.0*, com vários pontos de mínimo local no intervalo.

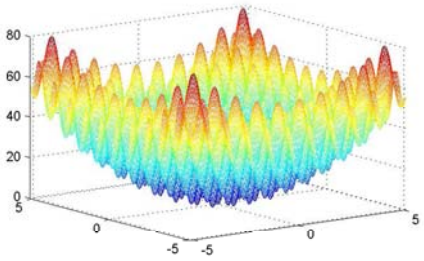
Para a função *Spherical* utilizamos 32 partículas distribuídas entre os processadores da rede, de acordo com as Tabelas 4, 5 e 6. O algoritmo PSO foi implementando com 16 iterações em todos os casos. Na Tabela 4 são apresentados os resultados com relação a tempo de execução, *speedup* e eficiência para a comunicação em anel. Na Tabela 5 são apresentados os resultados para comunicação vizinhança. Na Tabela 6 são apresentados os resultados para comunicação *broadcast*. As Tabelas 4 e 5, demonstram que o tempo de execução diminui com o acréscimo de processadores na rede. Entretanto a Tabela 6 mostra que o tempo de execução para o caso da rede configurada com 32 processadores foi maior em relação ao tempo de execução da rede configurada com 16 processadores. Estas tabelas geraram o gráfico de *speedup* da Figura 48 e o gráfico de eficiência da Figura 49.



(a) Curva da função *Spherical*



(b) Curva da função *Rosenbrock*



(c) Curva da função *Rastrigin*

Figura 47: Curvas das funções utilizadas no processo de otimização

Tabela 4: Resultados da função *Spherical* para a comunicação em anel

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	32	1079592	1	1
2	16	544755	1,98179365	0,990896825
4	8	275971	3,911976258	0,977994065
8	4	141763	7,615470892	0,951933861
16	2	75826	14,23775486	0,889859679
32	1	44914	24,03687046	0,751152202

Tabela 5: Resultados da função *Spherical* para a comunicação vizinhança

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	32	1079592	1	1
2	16	544755	1,98179365	0,990896825
4	8	277706	3,887535739	0,971883935
8	4	143707	7,512452421	0,939056553
16	2	76981	14,02413583	0,876508489
32	1	45446	23,75549003	0,742359064

Tabela 6: Resultados da função *Spherical* para a comunicação *broadcast*

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	32	1079592	1	1
2	16	544755	1,98179365	0,990896825
4	8	283921	3,802438002	0,950609501
8	4	160212	6,738521459	0,842315182
16	2	109562	9,853708403	0,615856775
32	1	115140	9,376341845	0,293010683

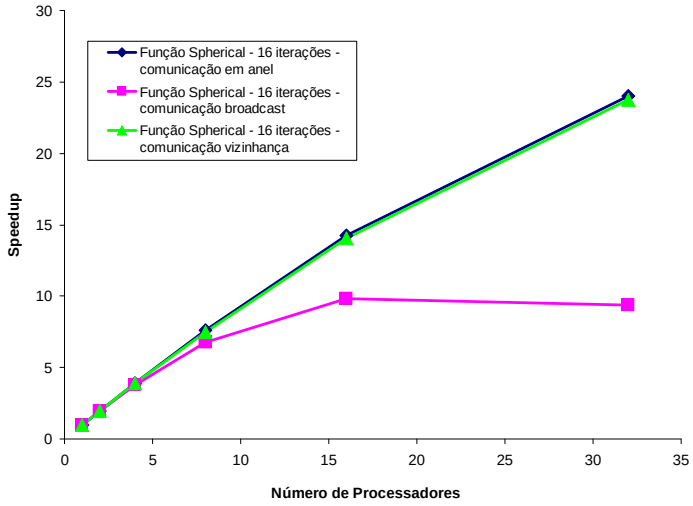


Figura 48: *Speedup* da rede para as três topologias de migração utilizando a função *Spherical*

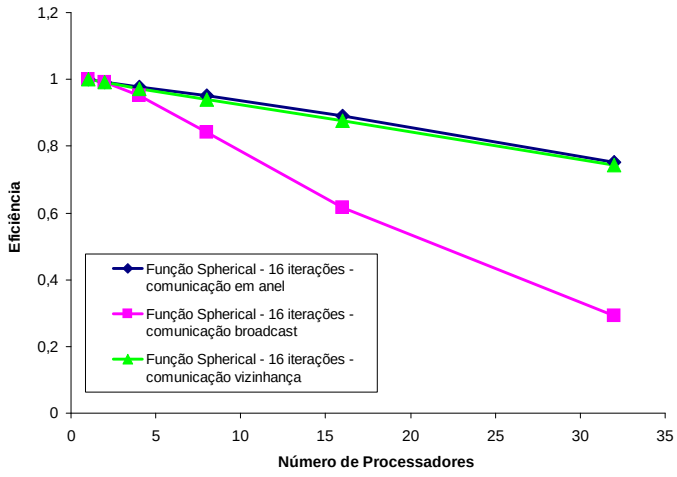


Figura 49: Eficiência da rede para as três topologias de migração utilizando a função *Spherical*

Para a função *Rosenbrock* utilizamos 64 partículas distribuídas entre os processadores da rede, de acordo com as Tabelas 7, 8, 9, implementando 32 iterações em todos os casos. Na Tabela 7 são apresentados os resultados da comunicação em anel. Na Tabela 8 são apresentados os resultados da comunicação vizinhança. Na Tabela 9 são apresentados os resultados da comunicação *broadcast*. Nas três tabelas observa-se que o tempo de execução das aplicações diminui com o aumento do número de processadores da rede. Estas tabelas geraram o gráfico de *speedup* da Figura 50 e o gráfico de eficiência da Figura 51.

Tabela 7: Resultados da função *Rosenbrock* para a comunicação em anel

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	10232089	1	1
2	32	5151219	1,986343233	0,993171616
4	16	2582747	3,961707825	0,990426956
8	8	1304211	7,845424552	0,980678069
16	4	663325	15,42545359	0,964090849
32	2	343537	29,78453267	0,930766646

Tabela 8: Resultados da função *Rosenbrock* para a comunicação vizinhança

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	10232089	1	1
2	32	5151219	1,986343233	0,993171616
4	16	2595968	3,941531252	0,985382813
8	8	1314996	7,781079942	0,972634993
16	4	671576	15,23593607	0,952246004
32	2	349769	29,25384754	0,914182736

Tabela 9: Resultados da função *Rosenbrock* para a comunicação *broadcast*

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	10232089	1	1
2	32	5151219	1,986343233	0,993171616
4	16	2602982	3,930910394	0,982727599
8	8	1367434	7,482693132	0,935336641
16	4	801790	12,76155727	0,797597329
32	2	617331	16,57472085	0,517960027

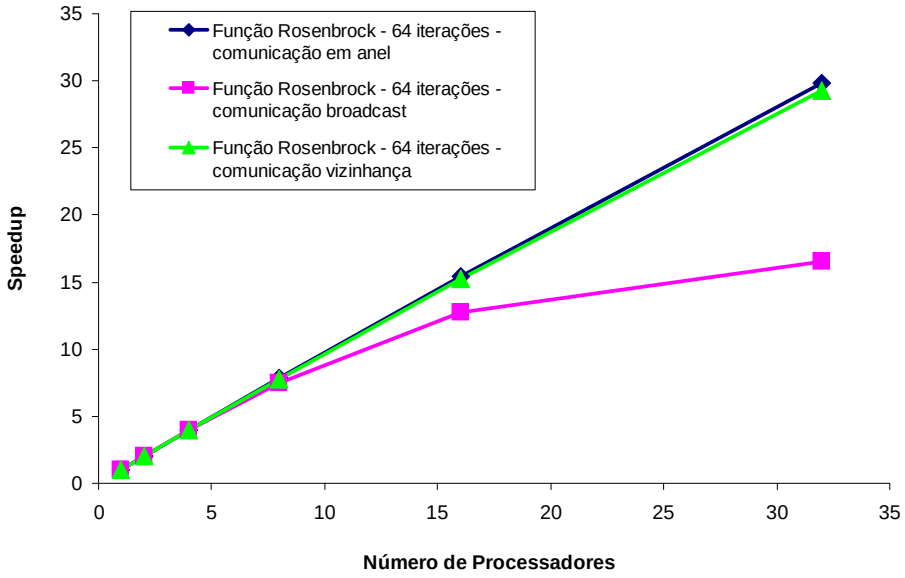


Figura 50: *Speedup* da rede para as três topologias de migração utilizando a função *Rosenbrock*

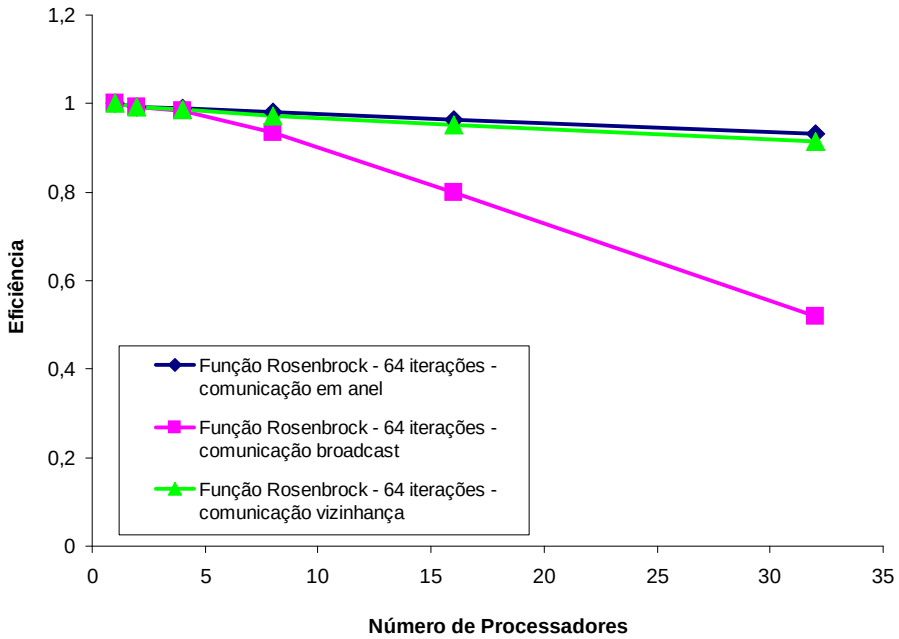


Figura 51: Eficiência da rede para as três topologias de migração utilizando a função *Rosenbrock*

Para a função *Rastrigin* utilizamos 64 partículas distribuídas entre os processadores da rede, conforme Tabelas 10, 11, 12, implementando 32 iterações, para o algoritmo PSO, em todos os casos. Na Tabela 10 são apresentados os resultados da comunicação em anel. Na Tabela 11 são apresentados os resultados da comunicação vizinhança. Na Tabela 12

são apresentados os resultados de simulação para a comunicação *broadcast*. Nestas tabelas, de maneira semelhante a que ocorreu com a função *Rosenbrock*, vemos que o tempo de execução da aplicação diminui com o acréscimo de processadores na rede, para as três topologias de comunicação. O algoritmo PSO para esta função que possui o maior tempo de execução em todos os casos, isto se deve ao tempo necessário para o cálculo de *fitness* que é superior ao das outras funções. Estas tabelas geraram o gráfico de *speedup* da Figura 52 e o gráfico de eficiência da Figura 53.

Tabela 10: Resultados da função *Rastrigin* para a comunicação em anel

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	46166574	1	1
2	32	23088801	1,999522366	0,999761183
4	16	11575210	3,988400556	0,997100139
8	8	5799370	7,960618826	0,995077353
16	4	2942026	15,69210265	0,980756416
32	2	1485505	31,0780334	0,971188544

Tabela 11: Resultados da função *Rastrigin* para a comunicação vizinhança

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	46166574	1	1
2	32	23088801	1,999522366	0,999761183
4	16	11575246	3,988388152	0,997097038
8	8	5810538	7,945318316	0,99316479
16	4	2952458	15,63665732	0,977291083
32	2	1498289	30,81286321	0,962901975

Tabela 12: Resultados da função *Rastrigin* para a comunicação *broadcast*

Número de processadores	Partículas por processador	Tempo em ciclos de *clock*	*Speedup*	Eficiência
1	64	46166574	1	1
2	32	23088801	1,999522366	0,999761183
4	16	11600572	3,97968083	0,994920207
8	8	5853974	7,886364716	0,985795589
16	4	3107786	14,85513288	0,928445805
32	2	1747628	26,41670539	0,825522043

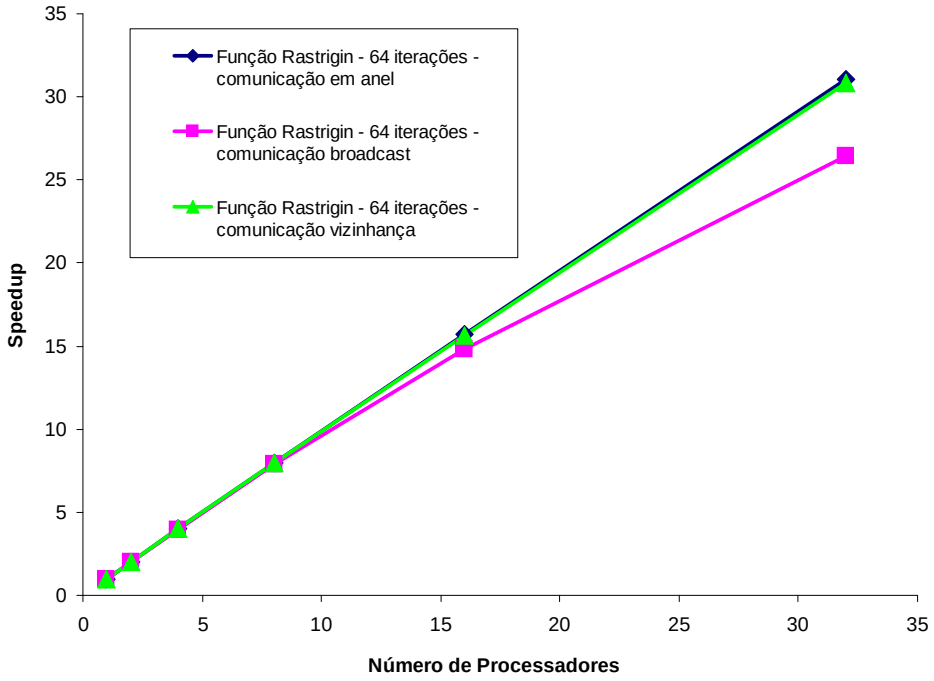


Figura 52: *Speedup* da rede para as três topologias de migração utilizando a função *Rastrigin*

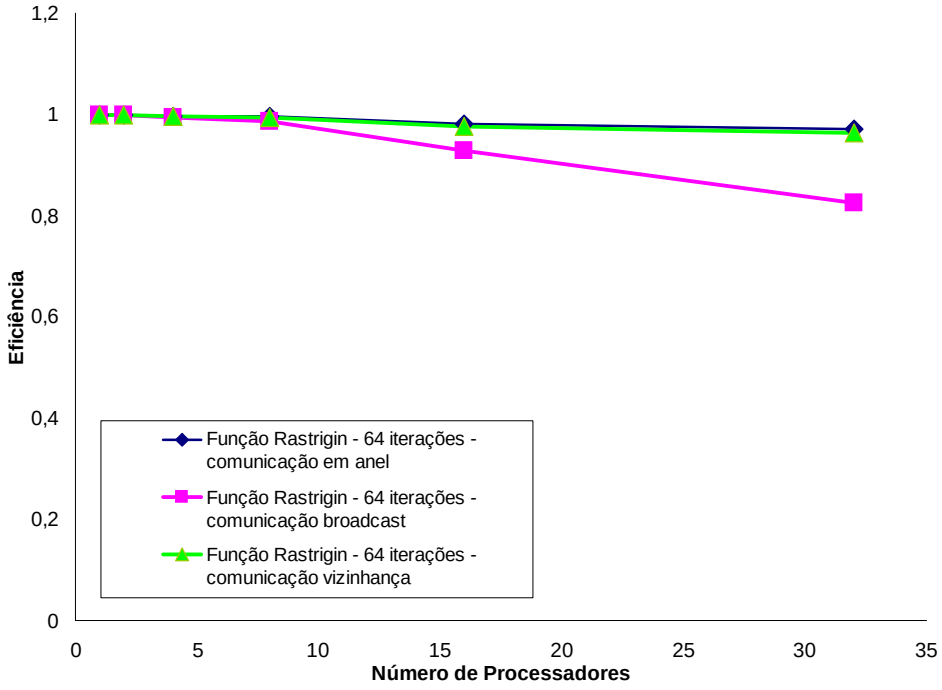


Figura 53: Eficiência da rede para as três topologias de migração utilizando a função *Rastrigin*

Com a análise dos resultados obtidos, verificamos que o desempenho da rede está relacionado ao padrão de comunicação. Na topologia anel foram verificados os maiores

valores de *speedup* e eficiência, neste caso, a cada iteração do algoritmo, um processador faz acesso ao seu módulo de memória e ao módulo pertencente ao vizinho da direita, com isso temos dois processadores competindo pelo uso do barramento. Na topologia vizinhança cada processador acessa o seu módulo de memória, o módulo de memória do vizinho da esquerda e o módulo de memória do vizinho da direita. Em virtude disto, temos três processadores competindo pelo uso do barramento. Verificou-se um discreta diminuição no *speedup* e da eficiência, no caso da topologia vizinhança em relação a topologia em anel, isto se deve ao aumento do número de comunicações e ao aumento da competição pelo uso do módulo de memória.

Na topologia de comunicação *broadcast* os valores de *speedup* e eficiência diminuíram de maneira mais acentuada, nos três casos analisados. Nesta topologia o número de comunicações, para cada barramento, em cada iteração do algoritmo, cresce em um fator de N, onde N é o número de processadores da rede, o que aumenta o tempo necessário para a comunicação entre os processadores diminuindo o desempenho. Nesse caso, também há uma maior competição pelo uso do barramento em virtude de todos os processadores utilizarem todos os barramentos, para acessar dados nos módulos de memória, o que gera maior gargalo na comunicação, afetando o desempenho de maneira mais acentuada que no caso da topologia em vizinhança.

A função objetivo *Rastrigin* foi a que teve o melhor desempenho em termos de *speedup* e eficiência, conforme mostrado nas Figuras 52 e 53, em todos os casos analisados. Nesta função temos uma maior necessidade de ciclos para o cálculo de *fitness*; como este cálculo é feito concorrentemente, isto aumentou o desempenho da paralelização. O caso da função *Spherical* teve o pior desempenho, com o aumento da rede, conforme mostrado na Figura 48 e 49. Na rede com 32 processadores o *speedup* foi menor que o encontrado na rede com 16 processadores, isto se deve ao custo computacional da função *Spherical* que é o menor das três funções analisadas, o que diminuiu o parcela de computação paralela, diminuindo o desempenho da paralelização.

Concluímos que as topologias anel e vizinhança, por envolverem um número menor de comunicações, possuem um rendimento melhor se comparadas à topologia *broadcast*. Esta topologia acarreta um tempo maior de comunicação, pois todos os processadores da rede se comunicam a cada iteração do algoritmo. Quanto às funções objetivo concluímos que quanto maior o custo computacional delas, maior o desempenho da paralelização do

algoritmo.

Temos também o atraso relacionado à parte sequencial do programa nela, ao final das iterações, o PROCESSADOR(0) executa a leitura e comparação do *Gbest* de todos os módulos de memória para a obtenção da melhor partícula de todos os enxames($Best$). Esta parte sequencial cresce à medida que o número de processadores aumenta, afetando consequentemente, o desempenho da arquitetura com o aumento de processadores com isso, todos os padrões de comunicação apresentaram queda no desempenho com o aumento da rede.

3.6 Considerações finais do capítulo

Neste capítulo foram apresentados os resultados de simulação para a arquitetura proposta a qual se mostrou capaz de lidar com várias topologias de comunicação. Os resultados encontrados foram satisfatórios e compatíveis com o esperado. Verificamos nas simulações que o desempenho da rede, nesta aplicação, está relacionado ao padrão de comunicação, tendo a comunicação *broadcast* apresentado o pior desempenho em todas as simulações. Entretanto o custo computacional da função objetivo também influiu no resultado das simulações, quanto maior o custo há um aumento no desempenho do paralelismo. Com isso a função *Rastrigin* obteve o melhor desempenho em todas as simulações realizadas. Concluímos que o desempenho da rede proposta, nesta aplicação, está relacionado ao padrão de comunicação e ao tempo de computação paralela, que é influenciado pelo custo computacional da função objetivo. No próximo capítulo serão apresentadas as conclusões obtidas nesta dissertação e as perspectivas para trabalhos futuros incluindo melhorias na arquitetura apresentada.

Capítulo 4

CONCLUSÕES E TRABALHOS FUTUROS

Nesta dissertação, propomos uma arquitetura de rede de interconexão, baseada na topologia *crossbar*, com memória compartilhada. O modelo da rede foi especificado em VHDL e validado funcionalmente através de simulação. O desempenho da arquitetura proposta foi analisado, com base na execução do método PSO para a otimização de algumas funções objetivo. O esforço computacional envolvido torna esta aplicação uma forte candidata para execução num ambiente multiprocessado. Neste capítulo, tecemos algumas conclusões sobre o trabalho desenvolvido e apresentamos sugestões para trabalhos futuros.

4.1 Conclusões

A demanda por dispositivos cada vez mais rápidos, em virtude de novas aplicações, principalmente aplicações multimídia, pressiona a busca por soluções inovadoras, que ofereçam o desempenho esperado. Sistemas embutidos multiprocessados (MPSoCs) procuram atender a essa expectativa explorando paralelismo espacial. Tais sistemas devem ser capazes de executar várias aplicações concorrentemente ou executar uma mesma aplicação que possa ser paralelizável. Neste caso, a aplicação é dividida em parcelas, onde cada uma é alocada a um processador da rede, para serem executadas concorrentemente.

O paralelismo explorado acarreta uma maior comunicação entre os vários processos. Neste sentido, o conceito de barramento como meio de comunicação não pode mais ser empregado, por conta da alta contenção que este impõe. A solução adotada é introduzir o conceito de rede de interconexão. Neste trabalho, desenvolvemos a arquitetura de uma rede de interconexão baseada na topologia *crossbar*, utilizando memória compartilhada para a troca de informação entre os processos. Este sistema é composto por quatro

módulos principais: rede *crossbar*, processadores, memória compartilhada dividida em N módulos e um controlador de barramento. A rede *crossbar* é composta por N barramentos, onde cada processador faz acesso a um módulo de memória via barramento. A conexão da memória a estes barramentos é feita através de uma chave, implementada por um conjunto de portas *tristate*. Cada barramento utiliza N chaves, cada qual sendo responsável por conectar um dos N processadores a um módulo de memória. O controlador do barramento determina qual processador será o mestre do barramento e fornece os sinais necessários para a conexão deste com o módulo de memória requerido. Dessa forma, temos N controladores de barramento, sendo um para cada barramento. Este controlador é o componente responsável por gerir os conflitos de conexão, isto é, quando dois ou mais processadores solicitam o uso de um mesmo módulo de memória, no mesmo intervalo de tempo. Com isso cada controlador de barramento é constituído de um árbitro e N máquinas de estados, cada qual responsável por emitir os sinais de controle necessários para o respectivo processador e a chave de interconexão correspondente, a fim de estabelecer a comunicação processador-memória requisitada.

A arquitetura da rede proposta foi modelada em VHDL e validada funcionalmente através de simulação, utilizando o simulador MODELSIM (MODEL, 2012). O modelo desenvolvido é parametrizável, permitindo a especificação para N processadores e N módulos de memória.

A avaliação de desempenho foi feita com base na execução paralela do método de Otimização por Enxame de Partículas (*Particle Swarm Optimization* - PSO) aplicado em algumas funções objetivo, em comparação com a respectiva execução sequencial. Neste caso, um enxame de partículas foi dividido igualmente entre os processadores da rede, cada enxame foi alocado em um processador. A partir dos resultados experimentais obtidos, concluímos que o *speedup* e a eficiência do sistema, estão diretamente ligados aos seguintes fatores fatores: topologia de comunicação, tempo de computação paralela da aplicação e ao tamanho da rede.

As topologias anel e vizinhança, por envolverem um número menor de comunicações, possuem um rendimento melhor se comparadas à topologia *broadcast*. Esta topologia acarreta um tempo maior de comunicação, pois todos os processadores da rede se comunicam a cada iteração do algoritmo. O algoritmo PSO aplicado às funções objetivo *Rastrigin* e *Rosenbrock*, atingiu um desempenho maior se comparada à função *Spherical*

em termos de *speedup* e eficiência. Aquelas funções são mais complexas, como os cálculo destas funções é realizado em paralelo, o tempo de computação paralela aumenta, o que aumenta o desempenho da paralelização. O tempo de execução da parte sequencial da aplicação cresce à medida que o número de processadores aumenta. Esta parte da aplicação é responsável pela comparação dos melhores indivíduos encontrados por todos os processadores, ao final da execução, para a obtenção do melhor indivíduo e quanto mais processadores na rede maior será esta parcela.

A contribuição desta dissertação reside no fato de termos uma plataforma parametrizável baseada em rede *crossbar*, com memória compartilhada, em funcionamento. A memória compartilhada junto com a rede *crossbar* possibilita a comunicação entre os processadores sem bloqueio. Este modelo funcional permitirá a verificação do desempenho desta arquitetura frente às topologias propostas em outros trabalhos como por exemplo: malha 2D, toroide, hipercubo, entre outras. De acordo com a avaliação de desempenho realizada, em todos os casos, obtivemos melhora na paralelização do algoritmo em relação ao caso serial, isto pode ser verificado pelos valores de *speedup* encontrados maiores que 1 o que indica que esta arquitetura apresentou bons requisitos de desempenho na aplicação analisada. Entretanto, para uma melhor análise, se faz necessário avaliar o desempenho da plataforma utilizando outras aplicações com outros padrões de comunicação entre os processos.

4.2 Trabalhos futuros

A seguir apresentaremos algumas melhorias a serem realizadas na arquitetura proposta, com a finalidade de incluir outras funcionalidades e aumentar o seu desempenho.

A primeira proposta é a elaboração de uma memória *cache* em cada processador. Com isso, após o dado ser transferido do módulo de memória para a *cache*, os acessos serão feitos na memória cache e não mais no módulo de memória. De forma geral, estes módulos ficarão mais disponíveis para outros processadores, reduzindo o tempo de espera por liberação, já que o processador acessará o seu módulo de memória apenas em caso de falha na cache. Com a cache adicionada ao sistema, deverão ser também adicionados protocolos de coerência de cache, para evitar que dados desatualizados sejam utilizados pelos processadores da rede.

O desenvolvimento de uma unidade de ponto flutuante e sua inclusão no MLite_CPU

reduziria o tempo consumido pelas aplicações. Esta modificação é necessária, pois o PSO e outras aplicações, principalmente processamento de imagens, usam cálculo em ponto flutuante de maneira intensa. No caso de usarmos outros processadores, a escolha será feita por aqueles que já tenham esta funcionalidade.

A prototipagem da rede proposta em um dispositivo FPGA permitirá a avaliação de área ocupada e a frequência de *clock* máxima que o sistema poderá operar. Dessa forma teremos a possibilidade de avaliar a relação custo x desempenho da arquitetura.

A elaboração de um sistema operacional para a arquitetura, dará melhor suporte à comunicação dos processadores com os módulos de memória incluindo escalonamento de tarefas, trocas de contexto e gerenciamento de memória. Entretanto, este sistema operacional deverá ser reduzido, implementando apenas as funcionalidades necessárias às aplicações executadas.

Outras aplicações distribuídas deverão ser analisadas, tais como: processamento de imagens, algoritmos genéticos, sistemas criptográficos. A execução destas tarefas deverá levar em consideração a característica de multiprocessamento da arquitetura, com a finalidade de avaliarmos o respectivo desempenho.

Outros mecanismos de arbitragem poderão ser explorados, a fim de determinar tanto o custo quanto o desempenho oferecidos. Alguns exemplos são: NA-MOO (LEE; LEE; YOO, 2003), *daisy chaining* (TANENBAUM, 1998) e algoritmos de arbitragem baseados em loteria (LAHIRI; RAGHUNATHAN; LAKSHMINARAYANA, 2006).

REFERÊNCIAS

AMDAHL, G. Validity of the single processor approach to achieving large scale computing capabilities. In: ACM. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. Atlantic City, N.J, USA, 1967. p. 483–485.

BAINES, R. The DSP bottleneck. *Communications Magazine*, IEEE, v. 33, n. 5, p. 46–54, 1995.

BENINI, L.; BERTOZZI, D. Network-on-chip architectures and design methods. *Computers and Digital Techniques, IEE Proceedings*, v. 152, n. 2, p. 261–272, 2005. ISSN 1350-2387.

BENINI, L.; MICHELI, G. D. Networks on chips: A new SoC paradigm. *COMPUTER,*, IEEE Computer Society, p. 70–78, 2002.

BJERREGAARD, T.; MAHADEVAN, S. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 38, n. 1, jun 2006. ISSN 0360-0300.

CULLER, D.; SINGH, J.; GUPTA, A. *Parallel computer architecture: a hardware/software approach*. San Francisco, CA, USA: Morgan Kaufmann, 1999. ISBN 9781558603431.

CYBENKO, G. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, Elsevier, v. 7, n. 2, p. 279–301, 1989.

DALLY, W.; TOWLES, B. Route packets, not wires: on-chip interconnection networks. In: *IEEE Design Automation Conference, 2001. Proceedings*. Las Vegas, NV, USA: IEEE, 2001. p. 684–689. ISSN 0738-100X.

DUATO, J.; YALAMANCHILI, S.; LIONEL, N. *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1558608524.

DUNCAN, R. A survey of parallel computer architectures. *Computer*, v. 23, n. 2, p. 5 –16, feb. 1990. ISSN 0018-9162.

EAGER, D.; ZAHORJAN, J.; LAZOWSKA, E. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, IEEE, v. 38, n. 3, p. 408–423, 1989.

ENGELBRECHT, A. P. *Fundamentals of Computational Swarm Intelligence*. Chichester, UK.: John Wiley & Sons, 2006. ISBN 0470091916.

FENG, T. Y. A survey of interconnection networks. *Computer*, v. 14, n. 12, p. 12 –27, dec. 1981. ISSN 0018-9162.

FRANKLIN, M. VLSI performance comparison of banyan and crossbar communications networks. *Computers, IEEE Transactions on*, C-30, n. 4, p. 283 –291, april 1981. ISSN 0018-9340.

FREITAS, H. *Arquitetura de NoC programável baseada em múltiplos clusters de cores para suporte a padrões de comunicação coletiva*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação., 2009.

GALLINI, N.; SCOTCHMER, S. Intellectual property: when is it the best incentive system? In: *Innovation Policy and the Economy, Volume 2*. [S.l.]: MIT Press, 2002. p. 51–78.

GLASS, C. J.; NI, L. M. The turn model for adaptive routing. In: *Proceedings of the 19th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1992. (ISCA '92), p. 278–287. ISBN 0-89791-509-7.

GNU. GCC. *Disponível em:< http://www.gnu.org/software/gcc>. Acesso em 02/08/2012*, 2009.

GUERRIER, P.; GREINER, A. A generic architecture for on-chip packet-switched interconnections. In: IEEE. *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*. Paris, France, 2000. p. 250–256.

GUPTA, P.; MCKEOWN, N. Designing and implementing a fast crossbar scheduler, ieee. *Micro*, IEEE, v. 19, n. 1, p. 20–28, 1999.

GUPTA, R.; ZORIAN, Y. Introducing core-based system design. *Design Test of Computers, IEEE*, v. 14, n. 4, p. 15 –25, oct-dec 1997. ISSN 0740-7475.

HSIAO, M.; JIANG, S. A hand-shake protocol for spread spectrum multiple access networks. In: *Global Telecommunications Conference, 1989, and Exhibition. Communications Technology for the 1990s and Beyond. GLOBECOM '89.* [S.l.]: IEEE, 1989. p. 288 –292 vol.1.

HU, J.; MARCULESCU, R. Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures. In: *Design, Automation and Test in Europe Conference and Exhibition, 2003.* Munich, Germany: IEEE, 2003. p. 688 – 693. ISSN 1530-1591.

IVANOV, A.; MICHELI, G. D. Guest editors' introduction: The network-on-chip paradigm in practice and research. *Design & Test of Computers*, IEEE, v. 22, n. 5, p. 399–403, 2005.

KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: *Neural Networks, 1995. Proceedings., IEEE International Conference on.* Washington, DC, USA: ., 1995. v. 4, p. 1942 –1948 vol.4.

KIM, D.; LEE, K.; LEE, S.; YOO, H. A reconfigurable crossbar switch with adaptive bandwidth control for networks-on-chip. In: IEEE. *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on.* Kobe, Japan, 2005. p. 2369–2372.

KRUSKAL, C.; SNIR, M. The performance of multistage interconnection networks for multiprocessors. *Computers, IEEE Transactions on*, C-32, n. 12, p. 1091 –1098, dec. 1983. ISSN 0018-9340.

LACH, J.; MANGIONE-SMITH, W.; POTKONJAK, M. FPGA fingerprinting techniques for protecting intellectual property. In: IEEE. *Custom Integrated Circuits Conference, 1998. Proceedings of the IEEE 1998.* Santa Clara, CA, USA, 1998. p. 299–302.

LAHIRI, K.; RAGHUNATHAN, A.; LAKSHMINARAYANA, G. The lotterybus on-chip communication architecture. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 14, n. 6, p. 596–608, 2006. ISSN 1063-8210.

LEE, K.; LEE, S.; YOO, H. A distributed crossbar switch scheduler for on-chip networks. In: IEEE. *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*. San Jose, CA, USA, 2003. p. 671–674.

LI, Y.; HARMS, J.; HOLTE, R. Impact of lossy links on performance of multihop wireless networks. In: *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*. San Diego, CA, USA: IEEE, 2005. p. 303 – 308. ISSN 1095-2055.

MADISETTI, V.; SHEN, L. Interface design for core-based systems. *Design & Test of Computers, IEEE*, IEEE, v. 14, n. 4, p. 42–51, 1997.

MHAMDI, L.; GOOSSENS, K.; SENIN, I. Buffered crossbar fabrics based on networks on chip. In: IEEE. *Communication Networks and Services Research Conference (CNSR), 2010 Eighth Annual*. Montreal, Canada, 2010. p. 74–79.

MODEL. Modelsim pe student edition 10.1b. *Disponível em:< http://model.com. Acesso em 02/08/2012*, Mentor Graphics Corporation, 2012.

MOLGA, M.; SMUTNICKI, C. Test functions for optimization needs. *Test functions for optimization needs*, 2005.

MORAES, F.; CALAZANS, N.; MELLO, A.; MOLLER, L.; OST, L. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, Elsevier, v. 38, n. 1, p. 69–93, 2004.

MORAES, F.; MELLO, A.; MÖLLER, L.; OST, L.; CALAZANS, N. A low area overhead packet-switched network on chip: architecture and prototyping. *IFIP VLSI-SOC 2003*, p. 318–323, 2003.

NAVABI, Z. *VHDL: Analysis and Modeling of Digital Systems*. 2nd. ed. New York, NY, USA: McGraw-Hill, Inc., 1997. ISBN 0070464790.

NI, L.; MCKINLEY, P. A survey of wormhole routing techniques in direct networks. *Computer*, IEEE, v. 26, n. 2, p. 62–76, 1993.

OPENCORES. Opencores. *Disponível em:< http://opencores.org. Acesso em 02/08/2012*, 1999.

PALMA, J.; MELLO, A. de; MOLLER, L.; MORAES, F.; CALAZANS, N. Core communication interface for FPGAs. In: *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*. Porto Alegre, Brazil: IEEE, 2002. p. 183 – 188.

PASRICHA, S.; DUTT, N. *On-Chip Communication Architectures: System on Chip Interconnect*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 012373892X, 9780123738929.

PATTERSON, D.; HENNESSY, J. *Computer organization and design: the hardware/software interface*. San Francisco, California, USA: Morgan Kaufmann, 2009. ISBN 9780123744937.

RHOADS, S. Plasma processor. *Open Cores, Disponível em:< http://www. opencores. org/>. Acesso em 02/06/2012*, v. 21, 2007.

SCHALLER, R. Moore's law: past, present and future. *Spectrum, IEEE*, IEEE, v. 34, n. 6, p. 52–59, 1997.

SCHLETT, M. Trends in embedded-microprocessor design. *Computer*, IEEE, v. 31, n. 8, p. 44–49, 1998.

TAMIR, Y.; CHI, H.-C. Symmetric crossbar arbiters for VLSI communication switches. *Parallel and Distributed Systems, IEEE Transactions on*, v. 4, n. 1, p. 13 –27, jan 1993. ISSN 1045-9219.

TANENBAUM, A. S. *Structured Computer Organization*. 4th. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998. ISBN 0130959901.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

TAYAN, O. Networks-on-chip: Challenges, trends and mechanisms for enhancements. In: *Information and Communication Technologies, 2009. ICICT '09. International Conference on*. Karachi, Pakistan: IEEE, 2009. p. 57 –62.

WEBER, M. Arbiters: design ideas and coding styles. *Synopsys Users Group (SNUG), Boston, Mass, USA*, Citeseer, 2001.

WOLF, W. The future of Multiprocessor Systems-on-Chips. In: ACM. *Proceedings of the 41st annual Design Automation Conference*. New York, NY, USA, 2004. p. 681–685.

WOLF, W.; JERRAYA, A.; MARTIN, G. Multiprocessor System-on-Chip (MPSoC) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, IEEE, v. 27, n. 10, p. 1701–1713, 2008.

WOSZEZENKI, C. Alocação de tarefas e comunicação entre tarefas em MPSoCs. *M. Sc., Faculdade de Informática, PUCRS, Porto Alegre, RS, Brazil (June 2007)[in Portuguese]*, 2007.

YABARRENA, J. Tecnologias system on chip e CAN em sistemas de controle distribuído. Biblioteca Digital de Teses e Dissertações da USP, 2011.

ZEFERINO, C. *Redes-em-Chip: Arquiteturas e modelos para avaliação de área e desempenho. 2003. 242 f.* Tese (Doutorado) — Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, 2003.

ZEFERINO, C. A.; SUSIN, A. A. Socin: A parametric and scalable network-on-chip. In: *Proceedings of the 16th symposium on Integrated circuits and systems design*. Washington, DC, USA: IEEE Computer Society, 2003. (SBCCI '03), p. 169–. ISBN 0-7695-2009-X.

ZHANG, Y.; MORRIS, R.; KODI, A. Design of a performance enhanced and power reduced dual-crossbar Network-on-Chip (NoC) architecture. *Microprocessors and Microsystems*, Elsevier, v. 35, n. 2, p. 110–118, 2011.

APÊNDICE A – Descrição em VHDL da arquitetura

A.1 Contador de programa de MLite_CPU modificado

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity pc_next is
  generic (log: natural:=2; i: natural:=4);
   port(clk          : in std_logic;
        reset_in     : in std_logic;
        pc_new       : in std_logic_vector(31 downto 2);
        take_branch  : in std_logic;
        pause_in     : in std_logic;
        opcode25_0   : in std_logic_vector(25 downto 0);
        pc_source    : in pc_source_type;
        pc_future    : out std_logic_vector(31 downto 2);
        pc_current   : out std_logic_vector(31 downto 2);
        pc_plus4     : out std_logic_vector(31 downto 2));
end; --pc_next

architecture logic of pc_next is
   signal pc_reg : std_logic_vector(31 downto 2);
begin

pc_select: process(clk, reset_in, pc_new, take_branch, pause_in,
                   opcode25_0, pc_source, pc_reg)
   variable pc_inc      : std_logic_vector(31 downto 2);
   variable pc_next : std_logic_vector(31 downto 2);
begin
   pc_inc := bv_increment(pc_reg);  --pc_reg+1

   case pc_source is
   when FROM_INC4 =>
      pc_next := pc_inc;
   when FROM_OPCODE25_0 =>
      pc_next := pc_reg(31 downto 28) & opcode25_0;
   when FROM_BRANCH | FROM_LBRANCH =>
      if take_branch = '1' then
         pc_next := pc_new;
      else
```

```
            pc_next := pc_inc;
         end if;
      when others =>
         pc_next := pc_inc;
      end case;

      if pause_in = '1' then
         pc_next := pc_reg;
      end if;

      if reset_in = '1' then
         pc_reg <= ZERO(31 downto 2);
         pc_next := pc_reg;
      elsif rising_edge(clk) then
         pc_reg <= pc_next;
      end if;
      pc_next(31 downto (31-(log-1))):=conv_std_logic_vector (i, log);
      pc_reg(31 downto (31-(log-1)))<=conv_std_logic_vector (i, log);
      pc_inc(31 downto (31-(log-1))):=conv_std_logic_vector (i, log);

      pc_future <= pc_next;
      pc_current <= pc_reg;
      pc_plus4 <= pc_inc;
end process;

end; --logic
```

A.2 Contador para implementar o verificados de requisições pendentes

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

use work.mlite_pack.all;

entity count_mod32 is
  port (clk, rst, inib_count: in std_logic;
    out_c: out std_logic);
 end count_mod32;

 architecture comport of count_mod32 is
 signal int_count: std_logic_vector(11 downto 0):="111111111111";
 begin
   process(clk, rst)
     begin
       if (clk'event and clk = '1') then
   int_count<= int_count + "000000000001";
   if (int_count = "111111111111") and inib_count='0' then
     out_c <='1';
   else
   out_c <='0';
   end if;
   if (int_count = "111111111111") then
```

```
    int_count<="000000000000";
  end if;
 end if;

 if rst='1' or ack='1' then
   int_count<="111111111111";
   out_c<='0';
 end if;
 end process;
 end architecture;
```

A.3 Rede Crossbar

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_unsigned.all;

entity crossbar_e_mem is
  generic(N: natural :=N);
  port(clk: in std_logic;
       col_enable: in M_bits_vector (N-1 downto 0);
       data_write: in param_data (N-1 downto 0);
       write_byte: in param_write (N-1 downto 0);
       addr: in param_addr (N-1 downto 0);
       data_read: out param_data (N-1 downto 0));
end crossbar_e_mem;

architecture estructural of crossbar_e_mem is

component bus_1
generic(N: natural:=N);
  port (clk: in std_logic;
        enable: in std_logic_vector(N-1 downto 0);
        data_write: in param_data (N-1 downto 0);
        write_byte: in param_write (N-1 downto 0);
        addr: in param_addr(N-1 downto 0);
        data_read: out param_data(N-1 downto 0);
        mem_addr: out std_logic_vector (31 downto 2);
        mem_data_read: in std_logic_vector (31 downto 0);
        mem_data_write: out std_logic_vector (31 downto 0);
        mem_write_byte: out std_logic_vector (3 downto 0));
end component;

signal        mem_addr: param_addr(N-1 downto 0);
signal        mem_data_read: param_data (N-1 downto 0);
signal        mem_data_write:  param_data (N-1 downto 0);
signal        mem_write_byte: param_write (N-1 downto 0);
--signal        col_enable_t: M_bits_vector (N-1 downto 0);

constant memory_type : string := "TRI_PORT_X";

  begin

--A:for K in 0 to N-1 generate
```

```
--B:for L in 0 to N-1 generate
-- col_enable_t(K)(L)<=col_enable(L)(K);
--end generate;
--end generate;

C:    FOR i IN 0 TO N-1 generate
bus_N: bus_1 GENERIC MAP (N)
port map (clk, col_enable(i), data_write, write_byte, addr,
data_read, mem_addr(i), mem_data_read(i),
mem_data_write(i), mem_write_byte(i));
u9_ram: ram
generic map (memory_type, "code_" & integer'image(i) & ".txt")
port map (clk, '1', mem_write_byte(i), mem_addr(i),
mem_data_write(i), mem_data_read(i));
end generate;

end architecture;
```

A.4 Barramento da rede crossbar

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_unsigned.all;

entity bus_1 is
  generic(N: natural:=N);
  port (clk: in std_logic;
        enable: in std_logic_vector(N-1 downto 0);
        data_write: in param_data (N-1 downto 0);
        write_byte: in param_write (N-1 downto 0);
        addr: in param_addr(N-1 downto 0);
        data_read: out param_data(N-1 downto 0);
        mem_addr: out std_logic_vector (31 downto 2);
        mem_data_read: in std_logic_vector (31 downto 0);
        mem_data_write: out std_logic_vector (31 downto 0);
        mem_write_byte: out std_logic_vector (3 downto 0));
end bus_1;

architecture extructural of bus_1 is

--constant memory_type : string := "TRI_PORT_X";

begin


A:    FOR i IN 0 TO N-1 generate
   trist_DATA: tris_n_bits
   generic map (32)
   port map (mem_data_read, enable(i), data_read(i));


   trist_ADDR: tris_n_bits
   generic map (30)
   port map (addr(i)(31 downto 2), enable(i), mem_addr(31 downto 2));
```

```
    trist_DATA_WRITE: tris_n_bits
    generic map (32)
    port map (data_write(i), enable(i), mem_data_write);


    trist_WRITE_BYTE: tris_n_bits
    generic map (4)
    port map (write_byte(i), enable(i), mem_write_byte);
    end generate;

end architecture;
```

A.5 Máquinas de Estados

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_signed.all;

library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_signed.all;

entity sm_pri is
 -- generic(I: natural:=1; J: natural:=0);
  port(clk, rst: in std_logic;
  bus_req, arbit_out: in std_logic;
  en_tris, pause_cont, clr: out std_logic;
  ack_out: out std_logic);
end sm_pri;

architecture FSM of sm_pri is
  type estado is (reset, con_1, cont, wait1,
ack, disc, con_2, clear, pause, pause_disc);

    SIGNAL atual: estado:= reset;
    signal int_count: std_logic_vector(2 downto 0):="000";
    signal sinc: std_logic;

    begin
   process(clk, rst)
     begin
       if (clk'event and clk = '1') then
   int_count<= int_count + "001";
   if (int_count = "111") then
     sinc <='1';
   else
   sinc <='0';
   end if;
   if (int_count = "111") then
    int_count<="000";
```

```
  end if;
 end if;

 if rst='1' or ack='1' then
   int_count<="000";
   sinc <='0';
 end if;
 end process;

--fsm_master: if I=J generate  -- gera máquina de estados master
saida: process (atual)
begin
   if atual = reset then
   en_tris <='0';
   pause_cont <='1';
   clr <='0';
   ack_out <='0';

   elsif atual = con_1 then
   en_tris <='1';
   pause_cont <='1';
   clr <='0';
   ack_out <='0';

   elsif atual = cont then
   en_tris <='1';
   pause_cont <='0';
   clr <='0';
   ack_out <='0';

 elsif atual = wait1 then
 en_tris <='1';
 pause_cont <='0';
 clr <='0';
 ack_out <='0';

   elsif atual = ack then
   en_tris <='1';
   pause_cont <='0';
   clr <='0';
   ack_out <='1';

   elsif atual = disc then
   en_tris <='0';
   pause_cont <='0';
   clr <='0';
   ack_out <='0';


   elsif atual = con_2 then
   en_tris <='1';
   pause_cont <='0';
   clr <='0';
   ack_out <='0';


   elsif atual = clear then
   en_tris <='1';
```

```
  pause_cont<='0';
  clr<='1';
  ack_out<='0';

  elsif atual = pause then
  en_tris<='1';
  pause_cont<='1';
  clr<='0';
  ack_out<='0';


  elsif atual = pause_disc then
  en_tris<='0';
  pause_cont<='1';
  clr<='0';
  ack_out<='0';


end if;
end process saida;

trans: process (clk, rst)
begin
if rising_edge(clk) then
  case atual is
  when reset   =>  if rst='1' then
                   atual<=reset;
                   else
                   atual<= con_1;
                   end if;

  when con_1 => if sinc='1' then
                   atual<= cont;
                   else
                   atual<= con_1;
                   end if;

  when cont => if arbit_out='1' and bus_req='1' then
                   atual<=cont;
                   elsif bus_req='0' then
                   atual<=wait1;
                   elsif arbit_out='0' then
                   atual<=pause;
                   end if;

when wait1 => if arbit_out ='1' then
              atual<=ack;
              else
              atual<=disc;
              end if;

 when ack => atual <= disc;

 when disc => if arbit_out='1' then
                   atual<=con_2;
                   else
                   atual<=disc;
                   end if;
```

```
 when con_2 =>  if sinc='1' then
                   atual<=clear;
                   else
                   atual<= con_2;
                   end if;

 when clear => atual <= cont;

 when pause => atual <= pause_disc;

 when pause_disc => if arbit_out ='1' then
                    atual<=con_1;
                    else
                    atual<=pause_disc;
                    end if;

 end case;
 end if;

   if rst='1' then
     atual<=reset;
   end if;
   end process trans;
-- end generate;

 end FSM;



library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;
use ieee.std_logic_signed.all;

entity sm_sec is
  port(clk, rst: in std_logic;
  bus_req, arbit_out: in std_logic;
  en_tris, dis_en: out std_logic;
  ack_out, inib_count: out std_logic);
end sm_sec;

architecture FSM of sm_sec is
  type estado1 is (reset, wait_1, wait_2, con, dis,
  en, disc, ack);
    SIGNAL atual1: estado1:= reset;
    signal int_count: std_logic_vector(2 downto 0):="000";
    signal sinc: std_logic;

    begin

   process(clk, rst)
     begin
       if (clk'event and clk = '1') then
   int_count<= int_count + "001";
   if (int_count = "111") then
     sinc<='1';
   else
```

```
    sinc <='0';
    end if;
    if (int_count = "111") then
     int_count<="000";
  end if;
 end if;

 if rst='1' or ack='1' then
   int_count<="000";
   sinc <='0';
 end if;
 end process;

saida1: process (atual1)
begin


  if atual1 = reset then
  en_tris <='0';
  dis_en <='0';
  ack_out <='0';
  inib_count <='0';

  elsif atual1 = wait_1 then
  en_tris <='0';
  dis_en <='0';
  ack_out <='0';
  inib_count <='1';

  elsif atual1 = wait_2 then
  en_tris <='0';
  dis_en <='0';
  ack_out <='0';
  inib_count <='1';

  elsif atual1 = con then
  en_tris <='1';
  dis_en <='0';
  ack_out <='0';
  inib_count <='1';

  elsif atual1 = dis then
  en_tris <='1';
  dis_en <='1';
  ack_out <='0';
  inib_count <='1';

  elsif atual1 = en then
  en_tris <='1';
  dis_en <='0';
  ack_out <='0';
  inib_count <='1';

  elsif atual1 = disc then
  en_tris <='0';
  dis_en <='0';
  ack_out <='0';
  inib_count <='1';
```

```
   elsif atual1 = ack then
   en_tris <='0';
   dis_en <='0';
   ack_out <='1';
   inib_count <='1';

end if;
end process saida1;

trans1: process (clk, rst)
begin
if rising_edge(clk) then
  case atual1 is
  when reset   => if arbit_out='1' then
                  atual1 <=wait_1;
                  else
                  atual1 <= reset;
                  end if;

  when wait_1 => atual1<=wait_2;

  when wait_2 => atual1<=con;

  when con => if sinc='1' then
                atual1<=dis;
                else
                atual1<=con;
                end if;

  when dis=> if bus_req='0' then
                atual1<=en;
                else
                atual1<=dis;
                end if;


 when en=>   atual1<=disc;


 when disc=>   atual1<=ack;


 when ack=>   atual1<=reset;


 end case;
 end if;

   if rst='1' then
     atual1<=reset;
   end if;
   end process trans1;

 end FSM;
```

A.6 Controlador de barramento

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.mlite_pack.all;
use ieee.std_logic_unsigned.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.mlite_pack.all;
use ieee.std_logic_unsigned.all;

entity crossbar_ctrl is
  generic(N: natural:=N;
          log: natural:=log);
  port(clk_x8, clk, reset, reset_2: in std_logic;
       addr: in log_bits_vector (N-1 downto 0);
       col: out M_bits_vector (N-1 downto 0);
       grant: out std_logic_vector(N-1 downto 0));
end crossbar_ctrl;

architecture estructural of crossbar_ctrl is

component dmux_2_4_1bits
port (S1: in std_logic_vector(log-1 downto 0);
entrada: in std_logic;
EB: out std_logic_vector(N-1 downto 0));
end component;

component comp_addr
 generic (log: natural:=2; N: natural:=2);
  port (clr, reset: in std_logic;
    addr_in: in std_logic_vector(31 downto (31-(log-1)));
    bus_req_out: out std_logic;
    addr_mux: out std_logic_vector(31 downto (31-(log-1))));
end component;

component and_n is
generic(N: natural:=N);
port (ent: in std_logic_vector (N-1 downto 0);
  saida: out std_logic);
end component;

component count is
   port (clk, rst, inib_count: in std_logic;
    out_c: out std_logic);
end component;

component or_n is
generic(N: natural:=N);
port (ent: in std_logic_vector (N-1 downto 0);
  saida: out std_logic);
end component;

type req_2 is array (N-1 downto 0) of std_logic_vector (N-1 downto 0);
```

```
signal req_1: req_2;
signal gnt: req_2;
signal req_t: req_2;
signal act_me: req_2;
signal inib_count:req_2;
signal inib_count2: std_logic_vector(N-1 downto 0);
SIGNAL pause_cont: std_logic_vector(N-1 downto 0);
signal dis_en: req_2;
signal dis_en_t: req_2;
SIGNAL dis_en_proc: std_logic_vector(N-1 downto 0);
signal set_addr_latch: std_logic_vector(N-1 downto 0);
signal clear_addr: std_logic_vector(N-1 downto 0);
signal act: std_logic_vector(N-1 downto 0);
signal act_1: std_logic_vector(N-1 downto 0);
signal act_count: std_logic_vector(N-1 downto 0);
signal addr_reg: log_bits_vector(N-1 downto 0);
signal bus_req_reg: std_logic_vector(N-1 downto 0);

begin


-- GERAÇÃO DO DMUX, MUX E REGISTRADOR
dmux_MUX: for I in 0 to N-1 generate


dmux1: dmux_2_4_1bits
port map(addr_reg(I), '1', req_1(I));

comp:comp_addr
generic map(log, I)
port map(clear_addr(I), reset_2, addr(I), bus_req_reg(I), addr_reg(I));
end generate;

-- TRASNPOSTA PARA PAUSAR OS PROCESSADORES
A:for K in 0 to N-1 generate
B:for L in 0 to N-1 generate
  req_t(K)(L)<=req_1(L)(K);
  dis_en_t(K)(L)<=dis_en(L)(K);
end generate;
end generate;

-- GERAÇÃO ARBITRO
arbiter:for j in 0 to N-1 generate
arbiter1: rrarbiter
generic map(N)
port map(clk_x8, reset, req_t(j), act(j), gnt(j));
cnt: count
port map(clk_x8, reset, inib_count2(j), act_count(j));


--or_1: or_n generic map(N)-- or clear addr
--port map(clear_addr(j), clear_addr_latch(j));
--or_3: or_n generic map(N)-- or PAUSE_CONT
--port map(pause_cont(j), pause_cont_proc(j));
or_4: or_n generic map(N)-- or DIS_EN
port map(dis_en_t(j), dis_en_proc(j));
or_6:  or_n generic map(N)-- inibe o contador
port map(inib_count(j), inib_count2(j));
```

```
grant(j)<=((not(dis_en_proc(j)) and bus_req_reg(j)) or pause_cont(j));

-- OR ACK
or_5: or_n generic map(N)
port map(act_me(j), act_1(j));
act(j)<= act_1(j) or act_count(j);
end generate;

-- GERAÇÃO DE MICRO_SM
msm11: for m in 0 to N-1 generate
msm21: for o in 0 to N-1 generate
A1:if (m=o) generate
sm:sm_pri
port map(clk_x8, reset, req_t(m)(o), gnt(m)(o), col(m)(o),
pause_cont(o), clear_addr(o), act_me(m)(o));
 dis_en(m)(o)<='0';
 inib_count(m)(o)<='0';
end generate;
A2:if m /= o generate
sm:sm_sec
port map(clk_x8, reset, req_t(m)(o), gnt(m)(o), col(m)(o),
dis_en(m)(o), act_me(m)(o), inib_count(m)(o));

end generate;
end generate;
end generate;

end architecture;
```

A.7 Arquitetura Completa

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;

entity complete_arch is
end entity;

architecture estructural of complete_arch is

component crossbar_e_mem
    generic(N: natural :=N);
  port(clk: in std_logic;
       col_enable: in M_bits_vector (N-1 downto 0);
       data_write: in param_data (N-1 downto 0);
       write_byte: in param_write (N-1 downto 0);
       addr: in param_addr (N-1 downto 0);
       data_read: out param_data (N-1 downto 0));
       end component;

component crossbar_ctrl
   generic(N: natural:=N;
           log: natural:=log);
```

```
  port(clk_x8, clk, reset, reset_2: in std_logic;
       addr: in log_bits_vector (N-1 downto 0);
       col: out M_bits_vector (N-1 downto 0);
       grant: out std_logic_vector(N-1 downto 0));
end component;

component processors
  generic (log: natural:=log);
 port(clk, reset: in std_logic;
       grant: in std_logic_vector(N-1 downto 0);
       data_read_cross: in param_data(N-1 downto 0);
       data_write: out param_data(N-1 downto 0);
       byte_we_next: out param_write (N-1 downto 0);
       addr: out param_addr (N-1 downto 0);
       addr_cross: out log_bits_vector (N-1 downto 0));
       end component;

  signal clk: std_logic:='1';
  --signal clk_x2: std_logic:='1';
  signal clk_ctrl: std_logic:='1';
  signal reset: std_logic:='1';
  signal reset_2: std_logic:='1';
  signal write_byte: param_write(N-1 downto 0);
  signal data_write: param_data(N-1 downto 0);
  signal data_read: param_data(N-1 downto 0);
  signal addr: param_addr (N-1 downto 0);
  signal addr_mem: param_addr (N-1 downto 0);
  signal col: M_bits_vector (N-1 downto 0);
  signal grant: std_logic_vector(N-1 downto 0);
  signal addr_cross: log_bits_vector(N-1 downto 0);

  begin

  clk   <= not clk after 200 ns;
  --clk_x2   <= not clk_x2 after 100 ns;
  clk_ctrl<=not clk_ctrl after 25 ns;
  reset <= '0' after 400 ns;
  reset_2<= '0' after 7200 ns;


  A: for i in 0 to N-1 generate
  addr_mem(i)<=zero(31 downto (31-(log-1))) & addr(i)((30-(log-1)) downto 2);
  end generate;


  comp1: processors
  generic map(log)
  port map (clk, reset, grant, data_read, data_write,
  write_byte, addr, addr_cross);

  comp2: crossbar_e_mem
  generic map(N)
  port map (clk, col, data_write, write_byte,
  addr_mem, data_read);

  comp3: crossbar_ctrl
  generic map (N, log)
  port map (clk_ctrl, clk, reset, reset_2,
```

```
  addr_cross, col, grant);

end architecture;


\section*{A.8 Árbitro}
\lstset{language=VHDL,
            basicstyle=\footnotesize,
            numbers=none,
            numberstyle=\footnotesize,
            frame=none,
            rulesepcolor=\color{white}}
\begin{lstlisting}


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
 entity rrarbiter is
 generic (CNT : natural:=4);
 port (
 clk             : in    std_logic;
 rst_n           : in    std_logic;
 req             : in    std_logic_vector(CNT-1 downto 0);
 ack             : in    std_logic;
 grant           : out   std_logic_vector(CNT-1 downto 0)
 );
 end;
 architecture rrarbiter of rrarbiter is
 signal grant_q  : std_logic_vector(CNT-1 downto 0);
 signal pre_req  : std_logic_vector(CNT-1 downto 0);
 signal sel_gnt  : std_logic_vector(CNT-1 downto 0);
 signal isol_lsb : std_logic_vector(CNT-1 downto 0);
 signal mask_pre : std_logic_vector(CNT-1 downto 0);
 signal win      : std_logic_vector(CNT-1 downto 0);
 begin


 grant    <= grant_q;
 mask_pre <=      req and not
(std_logic_vector(unsigned(pre_req) - 1) or pre_req);
 sel_gnt  <= mask_pre and
std_logic_vector(unsigned(not(mask_pre)) + 1);
 isol_lsb <=      req and
std_logic_vector(unsigned(not(req)) + 1);
 win      <= sel_gnt when
mask_pre /= (CNT-1 downto 0 => '0') else isol_lsb;
 process (clk,rst_n)
 begin
 if rising_edge(clk) then
 if rst_n = '1' then
  pre_req <= (others => '0');
  grant_q <= (others => '0');
 else
  if ack = '1' or grant_q = (CNT-1 downto 0 => '0') then
   pre_req <= win;
  end if;
   if grant_q = (CNT-1 downto 0 => '0') or ack = '1' then
```

```
    grant_q <= win;
   end if;
  end if;
  end if;
  end process;
end rrarbiter;
```

A.9 Processadores

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;

entity processors is
  generic (log: natural:=2);
  port(clk, reset: in std_logic;
       grant: in std_logic_vector(N-1 downto 0);
       data_read_cross: in param_data(N-1 downto 0);
       data_write: out param_data(N-1 downto 0);
       byte_we_next: out param_write (N-1 downto 0);
       addr: out param_addr (N-1 downto 0);
       --bus_req: out std_logic_vector (N-1 downto 0);
       addr_cross: out log_bits_vector (N-1 downto 0));
  end; --entity tbench
architecture logic of processors is

   constant memory_type : string := "TRI_PORT_X";
   --signal pause_m          : std_logic_vector (N-1 downto 0);
   signal interrupt        : std_logic;
   signal address          : param_addr (N-1 downto 0);
   signal tmp_addr         : param_addr (N-1 downto 0);
   signal mem_write        : param_write (N-1 downto 0);

   --component req_gnt
  --port (clk, rst, gnt: in std_logic;
    --grantee: out std_logic);
  --end component;
begin --architecture
   ------------------------------PROCESSADORES ------------------------------
   ---------------------------------------------------------------------------
 CPU: for j in 0 to N-1 generate
      u1_plasma: mlite_cpu
      generic map (memory_type, "defaut", "default",
      "default", 3, log, j)
      port map (clk, reset, interrupt, tmp_addr(j),
      mem_write(j), address(j), byte_we_next(j),
      data_write(j), data_read_cross(j), grant(j));

      --u2_reg_gnt: req_gnt
      --port map (clk, reset, grant(i), pause_m(i));
   ------------------------------END PROCESSADORES------------------------------
   ---------------------------------------------------------------------------
   --bus_req(i)<=address(i)(29);
   addr_cross(j)<=address(j)(31 downto (31-(log-1)));
   addr(j)<=address(j);
```

```
end generate;
end; --architecture logic
```

A.10 Pacote Mlite_pack Modificado

```
-----------------------------------------------------------------------
-- TITLE: Plasma Misc. Package
-- AUTHOR: Steve Rhoads (rhoadss@yahoo.com)
-- DATE CREATED: 2/15/01
-- FILENAME: mlite_pack.vhd
-- PROJECT: Plasma CPU core
-- COPYRIGHT: Software placed into the public domain by the author.
--    Software 'as is' without warranty.  Author liable for nothing.
-- DESCRIPTION:
--    Data types, constants, and add functions needed for the Plasma CPU.
-----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package mlite_pack is

-------------------------------------
--||||||||||||||||||||||||||||| - -
-------------------------------------
----  MUDANÇA DE PARAMETROS  ----
-------------------------------------
   constant N: NATURAL :=4;
   constant log: NATURAL:=2;
-------------------------------------
--||||||||||||||||||||||||||||| - -
-------------------------------------

  constant ZERO_1: std_logic_vector(127 downto 0) :=(OTHERS => '0');

   constant ZERO           : std_logic_vector(31 downto 0) :=
      "00000000000000000000000000000000";
   constant ONES           : std_logic_vector(31 downto 0) :=
      "11111111111111111111111111111111";
   --make HIGH_Z equal to ZERO if compiler complains
   constant HIGH_Z         : std_logic_vector(31 downto 0) :=
      "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";

   subtype alu_function_type is std_logic_vector(3 downto 0);
   constant ALU_NOTHING    : alu_function_type := "0000";
   constant ALU_ADD        : alu_function_type := "0001";
   constant ALU_SUBTRACT   : alu_function_type := "0010";
   constant ALU_LESS_THAN  : alu_function_type := "0011";
   constant ALU_LESS_THAN_SIGNED : alu_function_type := "0100";
   constant ALU_OR         : alu_function_type := "0101";
   constant ALU_AND        : alu_function_type := "0110";
   constant ALU_XOR        : alu_function_type := "0111";
   constant ALU_NOR        : alu_function_type := "1000";
```

```
subtype shift_function_type is std_logic_vector(1 downto 0);
constant SHIFT_NOTHING         : shift_function_type := "00";
constant SHIFT_LEFT_UNSIGNED   : shift_function_type := "01";
constant SHIFT_RIGHT_SIGNED    : shift_function_type := "11";
constant SHIFT_RIGHT_UNSIGNED  : shift_function_type := "10";

subtype mult_function_type is std_logic_vector(3 downto 0);
constant MULT_NOTHING          : mult_function_type := "0000";
constant MULT_READ_LO          : mult_function_type := "0001";
constant MULT_READ_HI          : mult_function_type := "0010";
constant MULT_WRITE_LO         : mult_function_type := "0011";
constant MULT_WRITE_HI         : mult_function_type := "0100";
constant MULT_MULT             : mult_function_type := "0101";
constant MULT_SIGNED_MULT      : mult_function_type := "0110";
constant MULT_DIVIDE           : mult_function_type := "0111";
constant MULT_SIGNED_DIVIDE    : mult_function_type := "1000";

subtype a_source_type is std_logic_vector(1 downto 0);
constant A_FROM_REG_SOURCE : a_source_type := "00";
constant A_FROM_IMM10_6    : a_source_type := "01";
constant A_FROM_PC         : a_source_type := "10";

subtype b_source_type is std_logic_vector(1 downto 0);
constant B_FROM_REG_TARGET : b_source_type := "00";
constant B_FROM_IMM        : b_source_type := "01";
constant B_FROM_SIGNED_IMM : b_source_type := "10";
constant B_FROM_IMMX4      : b_source_type := "11";

subtype c_source_type is std_logic_vector(2 downto 0);
constant C_FROM_NULL        : c_source_type := "000";
constant C_FROM_ALU         : c_source_type := "001";
constant C_FROM_SHIFT       : c_source_type := "001"; --same as alu
constant C_FROM_MULT        : c_source_type := "001"; --same as alu
constant C_FROM_MEMORY      : c_source_type := "010";
constant C_FROM_PC          : c_source_type := "011";
constant C_FROM_PC_PLUS4    : c_source_type := "100";
constant C_FROM_IMM_SHIFT16: c_source_type := "101";
constant C_FROM_REG_SOURCEN: c_source_type := "110";

subtype pc_source_type is std_logic_vector(1 downto 0);
constant FROM_INC4        : pc_source_type := "00";
constant FROM_OPCODE25_0 : pc_source_type := "01";
constant FROM_BRANCH      : pc_source_type := "10";
constant FROM_LBRANCH     : pc_source_type := "11";

subtype branch_function_type is std_logic_vector(2 downto 0);
constant BRANCH_LTZ : branch_function_type := "000";
constant BRANCH_LEZ : branch_function_type := "001";
constant BRANCH_EQ  : branch_function_type := "010";
constant BRANCH_NE  : branch_function_type := "011";
constant BRANCH_GEZ : branch_function_type := "100";
constant BRANCH_GTZ : branch_function_type := "101";
constant BRANCH_YES : branch_function_type := "110";
constant BRANCH_NO  : branch_function_type := "111";

-- mode(32=1,16=2,8=3), signed, write
subtype mem_source_type is std_logic_vector(3 downto 0);
constant MEM_FETCH    : mem_source_type := "0000";
```

```
constant MEM_READ32  : mem_source_type := "0100";
constant MEM_WRITE32 : mem_source_type := "0101";
constant MEM_READ16  : mem_source_type := "1000";
constant MEM_READ16S : mem_source_type := "1010";
constant MEM_WRITE16 : mem_source_type := "1001";
constant MEM_READ8   : mem_source_type := "1100";
constant MEM_READ8S  : mem_source_type := "1110";
constant MEM_WRITE8  : mem_source_type := "1101";
```

ARRAYS DICIONADOS AO PROJETO

```
type param_addr is array(NATURAL range <>)
of std_logic_vector (31 downto 2);
type param_data is array(NATURAL range <>)
of std_logic_vector (31 downto 0);
type param_write is array(NATURAL range<>)
of std_logic_vector (3 downto 0);
type M_bits_vector is array(NATURAL range <>)
of std_logic_vector(N-1 downto 0);
type log_bits_vector is array(NATURAL range <>)
of std_logic_vector(log-1 downto 0);


function bv_adder(a      : in std_logic_vector;
                  b      : in std_logic_vector;
                  do_add: in std_logic) return std_logic_vector;
function bv_negate(a : in std_logic_vector) return std_logic_vector;
function bv_increment(a : in std_logic_vector(31 downto 2)
                      ) return std_logic_vector;
function bv_inc(a : in std_logic_vector
                ) return std_logic_vector;

-- For Altera
COMPONENT lpm_ram_dp
   generic (
      LPM_WIDTH : natural;     -- MUST be greater than 0
      LPM_WIDTHAD : natural;     -- MUST be greater than 0
      LPM_NUMWORDS : natural := 0;
      LPM_INDATA : string := "REGISTERED";
      LPM_OUTDATA : string := "REGISTERED";
      LPM_RDADDRESS_CONTROL : string := "REGISTERED";
      LPM_WRADDRESS_CONTROL : string := "REGISTERED";
      LPM_FILE : string := "UNUSED";
      LPM_TYPE : string := "LPM_RAM_DP";
      USE_EAB  : string := "OFF";
      INTENDED_DEVICE_FAMILY  : string := "UNUSED";
      RDEN_USED  : string := "TRUE";
      LPM_HINT : string := "UNUSED");
   port (
      RDCLOCK    : in std_logic := '0';
      RDCLKEN    : in std_logic := '1';
      RDADDRESS : in std_logic_vector(LPM_WIDTHAD-1 downto 0);
      RDEN       : in std_logic := '1';
      DATA       : in std_logic_vector(LPM_WIDTH-1 downto 0);
      WRADDRESS : in std_logic_vector(LPM_WIDTHAD-1 downto 0);
      WREN       : in std_logic;
      WRCLOCK    : in std_logic := '0';
```

```
        WRCLKEN    : in std_logic := '1';
        Q          : out std_logic_vector(LPM_WIDTH-1 downto 0));
END COMPONENT;

-- For Altera
component LPM_RAM_DQ
    generic (
        LPM_WIDTH     : natural;     -- MUST be greater than 0
        LPM_WIDTHAD   : natural;     -- MUST be greater than 0
        LPM_NUMWORDS  : natural := 0;
        LPM_INDATA    : string := "REGISTERED";
        LPM_ADDRESS_CONTROL: string := "REGISTERED";
        LPM_OUTDATA   : string := "REGISTERED";
        LPM_FILE      : string := "UNUSED";
        LPM_TYPE      : string := "LPM_RAM_DQ";
        USE_EAB       : string := "OFF";
        INTENDED_DEVICE_FAMILY  : string := "UNUSED";
        LPM_HINT      : string := "UNUSED");
              port (
        DATA      : in std_logic_vector(LPM_WIDTH-1 downto 0);
        ADDRESS   : in std_logic_vector(LPM_WIDTHAD-1 downto 0);
        INCLOCK   : in std_logic := '0';
        OUTCLOCK  : in std_logic := '0';
        WE        : in std_logic;
        Q         : out std_logic_vector(LPM_WIDTH-1 downto 0));
end component;

-- For Xilinx
component RAM16X1D
    -- synthesis translate_off
    generic (INIT : bit_vector := X"16");
    -- synthesis translate_on
    port (DPO    : out STD_ULOGIC;
          SPO    : out STD_ULOGIC;
          A0     : in STD_ULOGIC;
          A1     : in STD_ULOGIC;
          A2     : in STD_ULOGIC;
          A3     : in STD_ULOGIC;
          D      : in STD_ULOGIC;
          DPRA0  : in STD_ULOGIC;
          DPRA1  : in STD_ULOGIC;
          DPRA2  : in STD_ULOGIC;
          DPRA3  : in STD_ULOGIC;
          WCLK   : in STD_ULOGIC;
          WE     : in STD_ULOGIC);
end component;

-- For Xilinx Virtex-5
component RAM32X1D
    -- synthesis translate_off
    generic (INIT : bit_vector := X"32");
    -- synthesis translate_on
    port (DPO    : out STD_ULOGIC;
          SPO    : out STD_ULOGIC;
          A0     : in STD_ULOGIC;
          A1     : in STD_ULOGIC;
          A2     : in STD_ULOGIC;
          A3     : in STD_ULOGIC;
```

```
            A4     : in STD_ULOGIC;
            D      : in STD_ULOGIC;
            DPRA0  : in STD_ULOGIC;
            DPRA1  : in STD_ULOGIC;
            DPRA2  : in STD_ULOGIC;
            DPRA3  : in STD_ULOGIC;
            DPRA4  : in STD_ULOGIC;
            WCLK   : in STD_ULOGIC;
            WE     : in STD_ULOGIC);
end component;

component pc_next
  generic (log: natural; i: natural);
   port(clk          : in std_logic;
        reset_in     : in std_logic;
        pc_new       : in std_logic_vector(31 downto 2);
        take_branch  : in std_logic;
        pause_in     : in std_logic;
        opcode25_0   : in std_logic_vector(25 downto 0);
        pc_source    : in pc_source_type;
        pc_future    : out std_logic_vector(31 downto 2);
        pc_current   : out std_logic_vector(31 downto 2);
        pc_plus4     : out std_logic_vector(31 downto 2));
end component;

component mem_ctrl
   port(clk           : in std_logic;
        reset_in      : in std_logic;
        pause_in      : in std_logic;
        nullify_op    : in std_logic;
        address_pc    : in std_logic_vector(31 downto 2);
        opcode_out    : out std_logic_vector(31 downto 0);

        address_in    : in std_logic_vector(31 downto 0);
        mem_source    : in mem_source_type;
        data_write    : in std_logic_vector(31 downto 0);
        data_read     : out std_logic_vector(31 downto 0);
        pause_out     : out std_logic;

        address_next  : out std_logic_vector(31 downto 2);
        byte_we_next  : out std_logic_vector(3 downto 0);

        address       : out std_logic_vector(31 downto 2);
        byte_we       : out std_logic_vector(3 downto 0);
        data_w        : out std_logic_vector(31 downto 0);
        data_r        : in std_logic_vector(31 downto 0));
end component;

component control
   port(opcode        : in  std_logic_vector(31 downto 0);
     intr_signal  : in  std_logic;
     rs_index     : out std_logic_vector(5 downto 0);
     rt_index     : out std_logic_vector(5 downto 0);
     rd_index     : out std_logic_vector(5 downto 0);
     imm_out      : out std_logic_vector(15 downto 0);
     alu_func     : out alu_function_type;
     shift_func   : out shift_function_type;
     mult_func    : out mult_function_type;
```

```
      branch_func   : out branch_function_type;
      a_source_out  : out a_source_type;
      b_source_out  : out b_source_type;
      c_source_out  : out c_source_type;
      pc_source_out: out pc_source_type;
      mem_source_out:out mem_source_type;
      exception_out: out std_logic);

end component;

component reg_bank
   generic(memory_type : string := "XILINX_16X");
   port(clk             : in  std_logic;
        reset_in        : in  std_logic;
        pause           : in  std_logic;
        rs_index        : in  std_logic_vector(5 downto 0);
        rt_index        : in  std_logic_vector(5 downto 0);
        rd_index        : in  std_logic_vector(5 downto 0);
        reg_source_out  : out std_logic_vector(31 downto 0);
        reg_target_out  : out std_logic_vector(31 downto 0);
        reg_dest_new    : in  std_logic_vector(31 downto 0);
        intr_enable     : out std_logic);
end component;

component bus_mux
   port(imm_in        : in  std_logic_vector(15 downto 0);
        reg_source    : in  std_logic_vector(31 downto 0);
        a_mux         : in  a_source_type;
        a_out         : out std_logic_vector(31 downto 0);

        reg_target    : in  std_logic_vector(31 downto 0);
        b_mux         : in  b_source_type;
        b_out         : out std_logic_vector(31 downto 0);

        c_bus         : in  std_logic_vector(31 downto 0);
        c_memory      : in  std_logic_vector(31 downto 0);
        c_pc          : in  std_logic_vector(31 downto 2);
        c_pc_plus4    : in  std_logic_vector(31 downto 2);
        c_mux         : in  c_source_type;
        reg_dest_out  : out std_logic_vector(31 downto 0);

        branch_func   : in  branch_function_type;
        take_branch   : out std_logic);
end component;

component alu
   generic(alu_type   : string := "DEFAULT");
   port(a_in          : in  std_logic_vector(31 downto 0);
        b_in          : in  std_logic_vector(31 downto 0);
        alu_function  : in  alu_function_type;
        c_alu         : out std_logic_vector(31 downto 0));
end component;

component shifter
   generic(shifter_type : string := "DEFAULT" );
   port(value         : in  std_logic_vector(31 downto 0);
        shift_amount  : in  std_logic_vector(4 downto 0);
        shift_func    : in  shift_function_type;
```

```
        c_shift      : out std_logic_vector(31 downto 0));
end component;

component mult
   generic(mult_type  : string := "DEFAULT");
   port(clk        : in  std_logic;
        reset_in   : in  std_logic;
        a, b       : in  std_logic_vector(31 downto 0);
        mult_func  : in  mult_function_type;
        c_mult     : out std_logic_vector(31 downto 0);
        pause_out  : out std_logic);
end component;

component pipeline
   port(clk             : in  std_logic;
        reset           : in  std_logic;
        a_bus           : in  std_logic_vector(31 downto 0);
        a_busD          : out std_logic_vector(31 downto 0);
        b_bus           : in  std_logic_vector(31 downto 0);
        b_busD          : out std_logic_vector(31 downto 0);
        alu_func        : in  alu_function_type;
        alu_funcD       : out alu_function_type;
        shift_func      : in  shift_function_type;
        shift_funcD     : out shift_function_type;
        mult_func       : in  mult_function_type;
        mult_funcD      : out mult_function_type;
        reg_dest        : in  std_logic_vector(31 downto 0);
        reg_destD       : out std_logic_vector(31 downto 0);
        rd_index        : in  std_logic_vector(5 downto 0);
        rd_indexD       : out std_logic_vector(5 downto 0);

        rs_index        : in  std_logic_vector(5 downto 0);
        rt_index        : in  std_logic_vector(5 downto 0);
        pc_source       : in  pc_source_type;
        mem_source      : in  mem_source_type;
        a_source        : in  a_source_type;
        b_source        : in  b_source_type;
        c_source        : in  c_source_type;
        c_bus           : in  std_logic_vector(31 downto 0);
        pause_any       : in  std_logic;
        pause_pipeline  : out std_logic);
end component;

component mlite_cpu
   generic(memory_type      : string  := "XILINX_16X";
        mult_type       : string  := "DEFAULT"; --AREA_OPTIMIZED
        shifter_type    : string  := "DEFAULT"; --AREA_OPTIMIZED
        alu_type        : string  := "DEFAULT"; --AREA_OPTIMIZED
        pipeline_stages : natural := 3;
        log: natural; i: natural); --2 or 3
port(clk          : in std_logic;
     reset_in     : in std_logic;
     intr_in      : in std_logic;

     address_next : out std_logic_vector(31 downto 2); --for synch ram
     byte_we_next : out std_logic_vector(3 downto 0);

     address      : out std_logic_vector(31 downto 2);
```

```
        byte_we       : out std_logic_vector(3 downto 0);
        data_w        : out std_logic_vector(31 downto 0);
        data_r        : in std_logic_vector(31 downto 0);
        mem_pause     : in std_logic);
  end component;

 component sm_pri
 --generic(I: natural:=1; J: natural:=0);
 port(clk, clk_m, rst: in std_logic;
 bus_req, arbit_out: in std_logic;
 en_tris, pause_cont, dis_en, clr_addr, clr: out std_logic;
 ack_out, inib_count: out std_logic);
 end component;

 component sm_sec is
 --generic(I: natural:=1; J: natural:=0);
 port(clk, clk_m, rst: in std_logic;
 bus_req, arbit_out: in std_logic;
 en_tris, pause_cont, dis_en, clr_addr, clr: out std_logic;
 ack_out, inib_count: out std_logic);
 end component;

  component rrarbiter is
generic (CNT : natural:=4);
port (
       clk             : in    std_logic;
       rst_n           : in    std_logic;
       req             : in    std_logic_vector(CNT-1 downto 0);
       ack             : in    std_logic;
       grant           : out   std_logic_vector(CNT-1 downto 0));
  end component;

  component tris_n_bits is
     generic (N: natural:=32);
     port ( Ent: in std_logic_vector (N-1 downto 0);
            C: in std_logic;
            S: out std_logic_vector (N-1 downto 0));
  end component;

  component cache
     generic(memory_type : string := "DEFAULT");
     port(clk            : in std_logic;
          reset          : in std_logic;
          address_next   : in std_logic_vector(31 downto 2);
          byte_we_next   : in std_logic_vector(3 downto 0);
          cpu_address    : in std_logic_vector(31 downto 2);
          mem_busy       : in std_logic;

          cache_access   : out std_logic;   --access 4KB cache
          cache_checking : out std_logic;   --checking if cache hit
          cache_miss     : out std_logic);  --cache miss
  end component; --cache

  component mux_2_1_Nbits
     generic (log: natural);
     port (EB1, EB2: in std_logic_vector (log-1 downto 0);
     S1: in std_logic;
     SAIDA1: out std_logic_vector (log-1 downto 0));
```

```
  end component;

  component ram
     generic(memory_type : string := "DEFAULT";
             file_read   : string := "code.txt");
     port(clk               : in std_logic;
          enable            : in std_logic;
          write_byte_enable : in std_logic_vector(3 downto 0);
          address           : in std_logic_vector(31 downto 2);
          data_write        : in std_logic_vector(31 downto 0);
          data_read         : out std_logic_vector(31 downto 0));
  end component; --ram

  component int_ram
     generic(memory_type : string := "DEFAULT");
     port(clk               : in std_logic;
          enable            : in std_logic;
          write_byte_enable : in std_logic_vector(3 downto 0);
          address           : in std_logic_vector(31 downto 2);
          data_write        : in std_logic_vector(31 downto 0);
          data_read         : out std_logic_vector(31 downto 0));
end component; --entity ram

  component uart
     generic(log_file : string := "UNUSED");
     port(clk          : in std_logic;
          reset        : in std_logic;
          enable_read  : in std_logic;
          enable_write : in std_logic;
          data_in      : in std_logic_vector(7 downto 0);
          data_out     : out std_logic_vector(7 downto 0);
          uart_read    : in std_logic;
          uart_write   : out std_logic;
          busy_write   : out std_logic;
          data_avail   : out std_logic);
  end component; --uart

  component eth_dma
     port(clk          : in std_logic;                          --25 MHz
          reset        : in std_logic;
          enable_eth   : in std_logic;
          select_eth   : in std_logic;
          rec_isr      : out std_logic;
          send_isr     : out std_logic;

          address      : out std_logic_vector(31 downto 2); --to DDR
          byte_we      : out std_logic_vector(3 downto 0);
          data_write   : out std_logic_vector(31 downto 0);
          data_read    : in std_logic_vector(31 downto 0);
          pause_in     : in std_logic;

          mem_address : in std_logic_vector(31 downto 2);  --from CPU
          mem_byte_we : in std_logic_vector(3 downto 0);
          data_w       : in std_logic_vector(31 downto 0);
          pause_out    : out std_logic;

          E_RX_CLK     : in std_logic;                          --2.5 MHz receive
          E_RX_DV      : in std_logic;                          --data valid
```

```
        E_RXD       : in std_logic_vector(3 downto 0);   --receive nibble
        E_TX_CLK    : in std_logic;                      --2.5 MHz transmit
        E_TX_EN     : out std_logic;                     --transmit enable
        E_TXD       : out std_logic_vector(3 downto 0)); --transmit nibble
end component; --eth_dma

component plasma
   generic(memory_type : string := "XILINX_X16";
           log_file    : string := "UNUSED";
           ethernet    : std_logic := '0';
           use_cache   : std_logic := '0');
   port(clk           : in std_logic;
        reset         : in std_logic;
        uart_write    : out std_logic;
        uart_read     : in std_logic;

        address       : out std_logic_vector(31 downto 2);
        byte_we       : out std_logic_vector(3 downto 0);
        data_write    : out std_logic_vector(31 downto 0);
        data_read     : in std_logic_vector(31 downto 0);
        mem_pause_in  : in std_logic;
        no_ddr_start  : out std_logic;
        no_ddr_stop   : out std_logic;

        gpio0_out     : out std_logic_vector(31 downto 0);
        gpioA_in      : in std_logic_vector(31 downto 0));
end component; --plasma

component ddr_ctrl
   port(clk       : in std_logic;
        clk_2x    : in std_logic;
        reset_in  : in std_logic;

        address   : in std_logic_vector(25 downto 2);
        byte_we   : in std_logic_vector(3 downto 0);
        data_w    : in std_logic_vector(31 downto 0);
        data_r    : out std_logic_vector(31 downto 0);
        active    : in std_logic;
        no_start  : in std_logic;
        no_stop   : in std_logic;
        pause     : out std_logic;

        SD_CK_P   : out std_logic;     --clock_positive
        SD_CK_N   : out std_logic;     --clock_negative
        SD_CKE    : out std_logic;     --clock_enable

        SD_BA     : out std_logic_vector(1 downto 0);  --bank_address
        SD_A      : out std_logic_vector(12 downto 0); --address(row or col)
        SD_CS     : out std_logic;     --chip_select
        SD_RAS    : out std_logic;     --row_address_strobe
        SD_CAS    : out std_logic;     --column_address_strobe
        SD_WE     : out std_logic;     --write_enable

        SD_DQ     : inout std_logic_vector(15 downto 0); --data
        SD_UDM    : out std_logic;     --upper_byte_enable
        SD_UDQS   : inout std_logic;   --upper_data_strobe
        SD_LDM    : out std_logic;     --low_byte_enable
        SD_LDQS   : inout std_logic);  --low_data_strobe
```

```
   end component; --ddr

end; --package mlite_pack


package body mlite_pack is

function bv_adder(a      : in std_logic_vector;
                  b      : in std_logic_vector;
                  do_add: in std_logic) return std_logic_vector is
   variable carry_in : std_logic;
   variable bb       : std_logic_vector(a'length-1 downto 0);
   variable result   : std_logic_vector(a'length downto 0);
begin
   if do_add = '1' then
      bb := b;
      carry_in := '0';
   else
      bb := not b;
      carry_in := '1';
   end if;
   for index in 0 to a'length-1 loop
      result(index) := a(index) xor bb(index) xor carry_in;
      carry_in := (carry_in and (a(index) or bb(index))) or
                  (a(index) and bb(index));
   end loop;
   result(a'length) := carry_in xnor do_add;
   return result;
end; --function


function bv_negate(a : in std_logic_vector) return std_logic_vector is
   variable carry_in : std_logic;
   variable not_a    : std_logic_vector(a'length-1 downto 0);
   variable result   : std_logic_vector(a'length-1 downto 0);
begin
   not_a := not a;
   carry_in := '1';
   for index in a'reverse_range loop
      result(index) := not_a(index) xor carry_in;
      carry_in := carry_in and not_a(index);
   end loop;
   return result;
end; --function


function bv_increment(a : in std_logic_vector(31 downto 2)
                     ) return std_logic_vector is
   variable carry_in : std_logic;
   variable result   : std_logic_vector(31 downto 2);
begin
   carry_in := '1';
   for index in 2 to 31 loop
      result(index) := a(index) xor carry_in;
      carry_in := a(index) and carry_in;
   end loop;
   return result;
end; --function
```

```
function bv_inc(a : in std_logic_vector
                ) return std_logic_vector is
   variable carry_in : std_logic;
   variable result   : std_logic_vector(a'length-1 downto 0);
begin
   carry_in := '1';
   for index in 0 to a'length-1 loop
      result(index) := a(index) xor carry_in;
      carry_in := a(index) and carry_in;
   end loop;
   return result;
end; --function

end; --package body
```

A.11 Comparador de Endereços

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use work.mlite_pack.all;

entity comp_addr is
  generic (log: natural:=2; N: natural:=3);
  port (clr, reset, set_addr, clk: in std_logic;
    addr_in: in std_logic_vector(31 downto (31-(log-1)));
    bus_req_out: out std_logic;
    addr_mux: out std_logic_vector(31 downto (31-(log-1))));
  end comp_addr;

architecture comport of comp_addr is
 begin
  process(addr_in, clr, clk, reset, set_addr)
  begin
  --if clk'event and clk='1' then
  if (addr_in /= conv_std_logic_vector (N, log) and reset='0') then
  addr_mux<=addr_in(31 downto (31-(log-1)));
  bus_req_out<='1';
  else
  addr_mux<=conv_std_logic_vector (N, log);
  --end if;
end if;
if clr='1' or reset='1' then
  bus_req_out<='0';
  addr_mux<=conv_std_logic_vector (N, log);
end if;
--if set_addr'event and set_addr='1' then
--  addr_mux<=conv_std_logic_vector (N, log);
--end if;
end process;
end architecture;
```

A.12 Decodificador de Endereços

```
library ieee;
use ieee.std_logic_1164.all;
--use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.mlite_pack.all;

entity dmux_2_4_1bits is
  generic (N: natural:=N;
  log: natural:=log);
port (S1: in std_logic_vector(log-1 downto 0);
entrada: in std_logic;
EB: out std_logic_vector(N-1 downto 0));
end dmux_2_4_1bits;

architecture algoritmic of dmux_2_4_1bits is

 --signal num: integer:=0;
begin
process (S1, entrada)
  begin
    --num<=(conv_integer(s1));
    eb<=zero_1(N-1 downto 0);
    eb(conv_integer(s1))<=entrada;
    end process;
end algoritmic;
```

A.13 Memória RAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_textio.all;
use std.textio.all;
use work.mlite_pack.all;

entity ram is
   generic(memory_type : string := "DEFAULT";
               file_read    : string:="CODE.TXT");
   port(clk                : in std_logic;
        enable             : in std_logic;
        write_byte_enable  : in std_logic_vector(3 downto 0);
        address            : in std_logic_vector(31 downto 2);
        data_write         : in std_logic_vector(31 downto 0);
        data_read          : out std_logic_vector(31 downto 0));
end; --entity ram

architecture logic of ram is
   constant ADDRESS_WIDTH   : natural := 20;
```

```
begin

   generic_ram:
   if memory_type /= "ALTERA_LPM" generate
   begin
   --Simulate a synchronous RAM
   ram_proc: process(clk, enable, write_byte_enable,
         address, data_write) --mem_write, mem_sel
      variable mem_size : natural := 2 ** ADDRESS_WIDTH;
      variable data : std_logic_vector(31 downto 0);
      subtype word is std_logic_vector(data_write'length-1 downto 0);
      type storage_array is
         array(natural range 0 to mem_size/4 - 1) of word;
      variable storage : storage_array;
      variable index : natural := 0;
      file load_file : text open read_mode is file_read;
      variable hex_file_line : line;
   begin

           --Load in the ram executable image
      if index = 0 then
         while not endfile(load_file) loop
--The following two lines had to be commented out for synthesis
            readline(load_file, hex_file_line);
            hread(hex_file_line, data);
            storage(index) := data;
            index := index + 1;
         end loop;
      end if;


         index := conv_integer(address(ADDRESS_WIDTH-1 downto 2));
         data := storage(index);

       if clk'event and clk='1' then
         if enable = '1' then
            if write_byte_enable(0) = '1' then
               data(7 downto 0) := data_write(7 downto 0);
            end if;
            if write_byte_enable(1) = '1' then
               data(15 downto 8) := data_write(15 downto 8);
            end if;
            if write_byte_enable(2) = '1' then
               data(23 downto 16) := data_write(23 downto 16);
            end if;
            if write_byte_enable(3) = '1' then
               data(31 downto 24) := data_write(31 downto 24);
            end if;
         end if;
         end if;

         if write_byte_enable /= "0000" then
            storage(index) := data;
         end if;


      data_read <= data;
   end process;
```

```
    end generate; --generic_ram

end; --architecture logic
```

A.13 Porta or N bits

```
library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;

entity or_n is
generic(N: natural:=N);
port (ent: in std_logic_vector (N-1 downto 0);
  saida: out std_logic);
end or_n;

architecture comport of or_n is
  begin
    a: process (ent)
    variable result: std_logic;
    begin
      result:='0';
    b:  for i in 0 to N-1 loop
        result:=result or ent(i);
      end loop;
      saida<=result;
    end process;
  end architecture;
```

APÊNDICE B – Funções em C e Linguagem de montagem utilizadas na avaliação de desempenho

B.1 Inicialização da rede e funções para acesso a memória compartilhada

```
 #Reserve 512 bytes for stack
    .comm InitStack, 512

    .text
    .align 2
    .global entry
    .ent entry
entry:
    .set noreorder

    #These four instructions should be the first instructions.
    #convert.exe previously initialized $gp, .sbss_start, .bss_end, $sp
    addiu $gp, $gp, 0x7ff8      #initialize global pointer
    la    $5, __bss_start       #$5 = .sbss_start
    la    $4, _end              #$2 = .bss_end
    addiu $sp, $sp, 0x7ff8    #initialize stack pointer

$BSS_CLEAR:
    sw    $0, 0($5)
    slt   $3, $5, $4
    bnez  $3, $BSS_CLEAR
    addiu $5, $5, 4

    jal   main
    nop
$L1:
    j $L1
    .end entry

    .global   OS_AsmMult
    .ent      OS_AsmMult
OS_AsmMult:
    .set noreorder
    multu $4, $5
    mflo  $2
    mfhi  $4
```

```
    jr      $31
    sw      $4, 0($6)
    .set reorder
    .end OS_AsmMult


  .global move_rc
    .ent move_rc
move_rc:
     .set noreorder
    lw       $2, 0($4) # v0 recebe conteúdo de gb
    lw       $3, 0($5) # v1 recebe conteúdo de c
    sw       $2, 0xFFFc($3) # mov. gb para a partição compartilhada da memória
    addiu    $3, $3, 4 # incrementa c
    jr       $31
    sw       $3, 0($5)# armazena c incrementado
     .set reorder
    .end move_rc


 .global rd
    .ent rd
   # rd(&gbp, n, &c)
rd:
    .set noreorder
    lw       $3, 0($6)  # conteudo de c -> reg v1
    addu    $8, $5, $3  # reg t0 recebe n+c
    lw       $2, 0xFFFc($8) # reg v0 recebe mem(ffff + t0)
    sw       $2, 0($4) # gbp recebe o conteúdo de v0
    addiu    $3, $3, 4 # incremento de c
    jr       $31
    sw       $3, 0($6)# armazena na mesma posição de c o c incrementado
    .set reorder
    .end rd
```

B.2 Função soma em ponto flutuante

```
unsigned long FP_Add(unsigned long a, unsigned long b)
{
    unsigned long c;
    unsigned long as, bs, cs;
    long ae, af, be, bf, ce, cf;
    as = a >> 31;//v
    ae = (a >> 23) & 0xff;
    af = 0x00800000 | (a & 0x007fffff);
    bs = b >> 31;
    be = (b >> 23) & 0xff;
    bf = 0x00800000 | (b & 0x007fffff);
    if(ae > be)
    {
        if(ae - be < 30)
            bf >>= ae - be;
        else
            bf = 0;
        ce = ae;
```

```
    }
    else
    {
        if(be - ae < 30)
            af >>= be - ae;
        else
            af = 0;
        ce = be;
    }
    cf = (as ? -af : af) + (bs ? -bf : bf);
    cs = cf < 0;// se cf < 0 cs = 1
    cf = cf>=0 ? cf : -cf;//v

    if(cf == 0)
        return cf;
    while(cf & 0xff000000)
    {
        ++ce;
        cf >>= 1;
    }
    while((cf & 0xff800000) == 0)
    {
        --ce;
        cf <<= 1;
    }
    c = (cs << 31) | (ce << 23) | (cf & 0x007fffff);
    if(ce < 1)
        c = 0;
    return c;
}
```

B.3 Função multiplicação em ponto flutuante

```
unsigned long FP_Mult(unsigned long a, unsigned long b)
{
    unsigned long c;
    unsigned long as, af, bs, bf, cs, cf;
    long ae, be, ce;
    unsigned long hi, lo;
    as = a >> 31;
    ae = (a >> 23) & 0xff;
    af = 0x00800000 | (a & 0x007fffff);
    bs = b >> 31;
    be = (b >> 23) & 0xff;
    bf = 0x00800000 | (b & 0x007fffff);
    cs = as ^ bs;// ou exclusivo entre os bits de sinal
    // protótipo da função para multiplicação rápida
    unsigned int OS_AsmMult(unsigned long a, unsigned long b, unsigned long *c);
    lo = OS_AsmMult(af, bf, &hi);
    cf = (hi << 9) | (lo >> 23);
    ce = ae + be - 0x80 + 1;//tirando o excesso de 127 da soma dos expoentes
    if(cf == 0)
        return cf;
    while((cf & 0xff000000)!= 0)
    {
```

```
        ++ce;
        cf >>= 1;
    }
    c = (cs << 31) | (ce << 23) | (cf & 0x007fffff);
    if(ce < 1)
        c = 0;
    return c;
}
```

B.4 Função comparação em ponto flutuante

```
int FP_Cmp(unsigned long a, unsigned long b)
{
    unsigned long as, ae, af, bs, be, bf;
    int gt;
    if(a == b)
        return 0;
    as = a >> 31;
    bs = b >> 31;
    if(as > bs)
        return -1;
    if(as < bs)
        return 1;
    gt = as ? -1 : 1;
    ae = (a >> 23) & 0xff;
    be = (b >> 23) & 0xff;
    if(ae > be)
        return gt;
    if(ae < be)
        return -gt;
    af = 0x00800000 | (a & 0x007fffff);
    bf = 0x00800000 | (b & 0x007fffff);
    if(af > bf)
        return gt;
    return -gt;
}
```

B.5 Função potência natural em ponto flutuante

```
unsigned long FP_Pot(unsigned long base, int exp)
{
 int i;
 unsigned long b=base;
 for (i=1; i<exp; i++)
 {
     b=FP_Mult(b,base);
 }
 return b;
}
```

B.6 Função para geração de números aleatórios

```
unsigned long rand_1(unsigned long seed_1)
{
 unsigned long a;
 unsigned long b;
 unsigned long c;
 unsigned long d;
 unsigned long seed_c;
     // operação mantissa
 seed_c = seed_1;
 a= (0x000000FF & (seed_c >> 15));
 b= a & 0x00000001;
 c= (a >> 2) & 0x00000001;
 c= b ^ c;
 b= (a >> 3) & 0x00000001;
 c= c ^ b;
 b= (a >> 4) & 0x00000001;
 c= c ^ b;
 a= (a << 1) & 0xff;
 a=a | c;// não desprezar

 ///operação expoente
 b= ((seed_c >> 23) & 0x00000007);
 c= b & 0x000000001;
 d= b >> 1;
 d= d & 0x000000001;
 d= d ^ c;
 b= (b << 1)& 0x7;
 b= b | d;

 b = b << 23;
 a = a << 15;
 c = seed_c & 0xfc000000;
 d = seed_c & 0x00007fff;
 seed_c= c | b | a | d;
return seed_c;
}
```

B.7 Função divisão em ponto flutuante

```
unsigned long FP_Div(unsigned long a, unsigned long b)
{
   unsigned long c;
   unsigned long as, af, bs, bf, cs, cf;
   unsigned long int hi, lo;
   long ae, be, ce;
   as = a >> 31;
   ae = (a >> 23) & 0xff;
   af = 0x00800000 | (a & 0x007fffff);
   bs = b >> 31;
   be = (b >> 23) & 0xff;
   bf = 0x00800000 | (b & 0x007fffff);
   cs = as ^ bs;
   ce = ae - (be - 0x80) + 6 - 8;
```

```
    lo = af / bf;
    hi = af % bf;
    lo = lo << 24;
    unsigned long int d = 16777216;
  if (hi != 0)
  {
  while (d > 1)
  {
      while (hi < bf)
            {
             hi = hi << 1;
             d = d >> 1;
            }
  lo=lo+(d);
  hi=hi % bf;
  if (hi == 0)
     {
      break;
     }
  }
  }
    cf = lo;
    if(cf == 0)
       return cf;
    while((cf & 0xff000000)!=0)
    {
       ++ce;
       cf >>= 1;
    }
    if(ce < 0)
       ce = 0;
    c = (cs << 31) | (ce << 23) | (cf & 0x007fffff);
    if(ce < 1)
       c = 0;
    return c;
}
```

B.8 Função seno

```
unsigned long FP_Sen(unsigned long arc)
{
unsigned long pi=0x40490FDB;
unsigned long pi_dois=0x3FC90FDB;
unsigned long dois_pi=0x40C90FDB;
unsigned long um_cinco_pi=0x4096CBE4;
unsigned long fact_9=0x48B13000;
unsigned long fact_11=0x4C184540;
unsigned long fact_7=0x459D8000;
unsigned long fact_5=0x42F00000;
unsigned long fact_3=0x40C00000;
unsigned long neg= 0x80000000;
unsigned long a=arc;
unsigned long p1;
unsigned long p2;
unsigned long p3;
```

```
unsigned long p4;
unsigned long p5;
unsigned long p6;

if (FP_Cmp(a, 0x00000000)==-1)
{
      a=a ^ neg;
      a=FP_Add(a, pi);
      }

while (FP_Cmp(a, dois_pi)>0)
{
      a=FP_Add(a, (dois_pi ^ neg));
      }

if ((FP_Cmp(a, pi_dois)==1) && (FP_Cmp(a, um_cinco_pi)<0))
{
   a = FP_Add(pi, (a ^ neg));
}
if ((FP_Cmp(a, um_cinco_pi)==1) && (FP_Cmp(a, dois_pi)<0))
{
a= FP_Add(a, (dois_pi ^ neg));
}

p2=FP_Div((FP_Pot(a, 3)), fact_3);

p3=FP_Div((FP_Pot(a, 5)), fact_5);

p4=FP_Div((FP_Pot(a, 7)), fact_7);

p5=FP_Div((FP_Pot(a, 9)), fact_9);

p6=FP_Div((FP_Pot(a, 11)), fact_11);

p1= FP_Add(a, (p2 ^neg));

p1= FP_Add(p1, p3);

p1= FP_Add(p1, (p4 ^neg));

p1= FP_Add(p1, p5);

p1= FP_Add(p1, (p6 ^neg));


return p1;
}
```

B.9 Funções de avaliação

```
//Rastrigin
unsigned long Rast(unsigned long x, unsigned long y)
{
   unsigned int a;
   unsigned int a_2;
```

```
    unsigned int a_3;
    a=FP_Add((FP_Mult(x,x)),(FP_Mult(y,y)));
    a=FP_Add(a, 0x41A00000);
    a_2=FP_Cos(FP_Mult(0x40C90FDB, x));
    a_3=FP_Cos(FP_Mult(0x40C90FDB, y));
    a_2=FP_Mult(a_2, 0x41200000);
    a_3=FP_Mult(a_3, 0x41200000);
    a=FP_Add(a, (a_2 ^ 0x80000000));
    a=FP_Add(a, (a_3 ^ 0x80000000));
    return a;
}
//Rosemblock
unsigned long Rose(unsigned long x, unsigned long y)
{
     unsigned int a;
     unsigned int a_2;
     a=0x42C80000;
     a_2=FP_Add(FP_Mult(x,x), y ^ 0x80000000);
     a_2=FP_Mult(a_2,a_2);
     a=FP_Mult(a,a_2);
     a_2= FP_Add(x, 0xBF800000);
     a_2=FP_Mult(a_2,a_2);
     a=FP_Add(a, a_2);
     return a;
}
//Sphere
unsigned long Sphe(unsigned long x, unsigned long y)
{
 unsigned int a;
 a=FP_Add((FP_Mult(x,x)),(FP_Mult(y,y)));
 return a;
}
```

B.9 Função principal do PSO

```
//função principal PSO
//inicialização dos parametros
int main (void)
{
     int iteracoes = 16;
     int particulas = 8;
     unsigned int x[particulas];
     unsigned int y[particulas];//posição inicial das particulas
     unsigned int p_b[particulas];
     int i =0;
     int j= 0;
     int k= 0;
     unsigned int p_bx[particulas];
     unsigned int p_by[particulas];
     unsigned int g_bx=0x00000000;
     unsigned int g_by=0x00000000;
     unsigned int g_b=0x00000000;
     unsigned int tmp_pb;
     unsigned int v_x[particulas];
     unsigned int v_y[particulas];
```

```
    unsigned int rnd_aux = 0x3F7FAEBC;
    unsigned int aux_w=0xBd800000;
    unsigned int w=0x3F800000;
    unsigned int r1;
    unsigned int r2;
    unsigned int vtemp_x;
    unsigned int vtemp_y;
    unsigned int x_temp;
    unsigned int y_temp;
    unsigned long c1=0x3FBEB852;
    unsigned long neg= 0x80000000;
    unsigned long v_max= 0x40000000;
    unsigned long dez=0x41200000;
    unsigned long n_cin=0xC0A00000;
    unsigned int gb_p;
    unsigned int gbx_p;
    unsigned int gby_p;
    unsigned int n_1=1;
    unsigned int n_0=0;
    unsigned int n=0x38000000;//aponta para o 4º processador
    unsigned int data_ready=0;
    unsigned int c=4;
    unsigned int n_proc=4;
    //unsigned int data_ready=0;
    //wr_m2(&data_ready, &b);

// inicialização do pbest e gbest
for (i=0; i<(particulas); i++)
{
   rnd_aux=rand_1(rnd_aux);
   x[i]=FP_Add((FP_Mult(rnd_aux, dez)),n_cin);
   rnd_aux=rand_1(rnd_aux);
   y[i]=FP_Add((FP_Mult(rnd_aux, dez)),n_cin);
   p_b[i]=Sphe(x[i],y[i]);
   rnd_aux=rand_1(rnd_aux);
   v_x[i]=rnd_aux;
   rnd_aux=rand_1(rnd_aux);
   v_y[i]=rnd_aux;
   p_bx[i]=x[i];
   p_by[i]=y[i];
   if (i==0)
   {
     g_b = p_b[0];
     g_bx = p_bx[0];
     g_by = p_by[0];
   }
}
// início do algorítmo
for (i=0; i<iteracoes; i++)// total de iterações
{
  for (j=0; j<particulas; j++)// total de particulas
   {
    tmp_pb=Sphe(x[j],y[j]);

    if (FP_Cmp(tmp_pb,p_b[j])==-1)
    {
       p_b[j]=tmp_pb;
       p_bx[j]=x[j];
```

```
            p_by[j]=y[j];
        }
        if (FP_Cmp(p_b[j],g_b)==-1)
        {
            g_b=p_b[j];
            g_bx=p_bx[j];
            g_by=p_by[j];
        }

    }
// atualização da velocidade da partícula
    for (j=0; j<particulas; j++)
    {
            rnd_aux=rand_1(rnd_aux);
            r1=rnd_aux;
            rnd_aux=rand_1(rnd_aux);
            r2=rnd_aux;
            vtemp_x=v_x[j];
            vtemp_y=v_y[j];
            x_temp=x[j];
            y_temp=y[j];
            v_x[j]=FP_Add(FP_Mult(w, vtemp_x),
            FP_Add(FP_Mult(FP_Mult(c1, r1),
            FP_Add(p_bx[j], (x_temp ^ neg))),
            FP_Mult(FP_Mult(c1, r2), FP_Add(g_bx, (x_temp ^ neg)))));
            v_y[j]=FP_Add(FP_Mult(w, vtemp_y),
            FP_Add(FP_Mult(FP_Mult(c1, r1),
            FP_Add(p_by[j], (y_temp ^ neg))),
            FP_Mult(FP_Mult(c1, r2), FP_Add(g_by, (y_temp ^ neg)))));

           if (FP_Cmp(v_x[j],v_max)==1)
           {
              v_x[j]=v_max;
           }

           if (FP_Cmp(v_y[j],v_max)==1)
           {
              v_y[j]=v_max;
           }

           if (FP_Cmp(v_x[j],(v_max ^ neg))==-1)
           {
              v_x[j]=v_max ^ neg;
           }
           if (FP_Cmp(v_y[j],(v_max ^ neg))==-1)
           {
             v_y[j]=v_max ^ neg;
           }
           x[j] = FP_Add(x_temp, v_x[j]);
           y[j] = FP_Add(y_temp, v_y[j]);
}
w = FP_Add(w, aux_w);

// comunicação
move_rc (&n_0, &c);
move_rc (&g_b, &c);
move_rc (&g_bx, &c);
move_rc (&g_by, &c);
```

```
c=4;
move_rc (&n_1, &c);
c=4;
n=0x40000000;//n aponta para p2

for (k=1; k<n_proc; k++)
{
   while (data_ready!=1)
   {
     rd(&data_ready, n, &c);
     c=4;
   }
c=8;
rd (&gb_p, n, &c);
rd (&gbx_p, n, &c);
rd (&gby_p, n, &c);
c=4;
data_ready=0;
     if (n==0xc0000000)
     {
       n=0x00000000;
     }
     else
     {
       n=n+0x40000000;
     }
      if ((FP_Cmp(g_b,gb_p)==1))
      {
        g_b= gb_p;
        g_bx= gbx_p;
        g_by= gby_p;
      }
}
}
n=0x40000000;//aponta para 2º processador
for (k=1; k<n_proc; k++)
{
while (data_ready!=1)
{
rd(&data_ready, n, &c);
c=4;
}
c=8;
rd (&gb_p, n, &c);
rd (&gbx_p, n, &c);
rd (&gby_p, n, &c);
c=4;
data_ready=0;
if ((FP_Cmp(g_b,gb_p)==1))
{
g_b= gb_p;
g_bx= gbx_p;
g_by= gby_p;
}
c=4;
n=n+0x40000000;
}
c=24;
```

```
move_rc (&g_b, &c);
move_rc (&g_bx, &c);
move_rc (&g_by, &c);
}
```