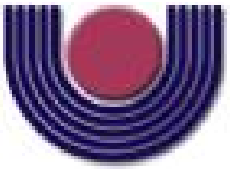

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Implementação de barramento para arquiteturas multiprocessadas em SystemC

Willian Dias Tamagi

CASCAVEL
2011

WILLIAN DIAS TAMAGI

IMPLEMENTAÇÃO DE BARRAMENTO PARA ARQUITETURAS MULTIPROCESSADAS EM SYSTEMC

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Marcio Oyamada

CASCAVEL
2011

WILLIAN DIAS TAMAGI

IMPLEMENTAÇÃO DE BARRAMENTO PARA ARQUITETURAS MULTIPROCESSADAS EM SYSTEMC

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Oyamada (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Anibal Mantovani Diniz
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Edmar André Bellorini
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 18 de novembro de 2011

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

HDL	Hardware Description Languages
MIPS	Microprocessor without Interlocked Pipeline Stages
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuits
OSI	Open SystemC Initiative
SPARC	Scalable Processor Architecture
ARM	Acorn RISC Machine
SISD	Single Instruction, Single Data
SIMD	Single Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
MIMD	Multiple Instruction, Multiple Data
UMA	Uniform Memory Access
NUMA	Non-uniform Memory Access
CC-NUMA	Cache Coherent Non-uniform Memory Access
SCU	Snoop Control Unit
MMCC	Multiprocessador Minimalista com Caches Coerentes

Sumário

Resumo

As arquiteturas de múltiplos processadores possuem uma grande vantagem custo-benefício em relação a se criar novos processadores mais potentes, pois o custo de associar processadores já existentes é muito mais barato. Assim como os processadores foram evoluindo, sua técnica de projeto e planejamento para o desenvolvimento também teve diversas transformações. Até a década de 80 os projetos de lógica programável eram desenvolvidos a partir de esquemáticos, então a partir da década de 90, linguagens de programação específicas, chamadas de HDLs (Hardware Description Languages), começaram a ser utilizadas. As ferramentas de descrição de hardware podem ser construídas em qualquer linguagem de propósito geral, algumas destas linguagens e outras bibliotecas oferecem total suporte para a modelagem de hardware e software, como portas, sinais e outros. Nesse trabalho foi utilizada a biblioteca de modelagem e simulação de hardware SystemC, com o objetivo de desenvolver uma arquitetura multiprocessada com memória de dados compartilhada, utilizando o modelo SardMIPS baseado na arquitetura de processadores MIPS. Um estudo de caso foi realizado de uma arquitetura com dois processadores, onde cada um possui sua própria memória de instruções mas compartilham uma única memória de dados através de um barramento.

Palavras-chave: MMCC, SystemC, Coerência de Cache, Multiprocessador.

Capítulo 1

Introdução

Graças à criação dos transistores pela *Bell Telephone Laboratories* em 1947, foi possível o desenvolvimento dos microprocessadores, cada vez mais complexos, menores e mais velozes. Segundo a lei de Moore [4], a densidade dos circuitos tem dobrado a cada dois anos e sua velocidade de operação a cada três. Este fato se deve a diminuição do tamanho dos transistores, com isso mais transistores caberão em uma área, que consequentemente terá maior velocidade. Por outro lado vemos a limitação física, pois mesmo aumentando o número de transistores, ainda trabalhamos em áreas similares, assim limitando a complexidade dos núcleos. Outro fato importante é que a frequência já não é dobrada a cada 3 anos [5].

Por esses e outros motivos a indústria foi obrigada a buscar meios para ganho em desempenho. Seguindo o caminho lógico para aumentar o desempenho de uma arquitetura com um único processador, as arquiteturas de múltiplos processadores, este tipo de arquitetura começou a ser estudada para ser empregada em supercomputadores nas décadas de 60 e 70 [6] [7]. Constituídos de diversos módulos de processamento e utilizando-se de componentes discretos, os módulos destes sistemas eram interconectados através de barramentos compartilhados [8], redes de processadores com conexões ponto a ponto [9] [10] ou combinações dessas técnicas [11]. Este tipo de iniciativa não se encaixa somente em computadores de uso comum, também está sendo aplicada amplamente em projetos de sistemas embarcados [12].

Assim como os processadores foram evoluindo, sua técnica de projeto e planejamento para o desenvolvimento também teve diversas transformações. Até a década de 80 os projetos de lógica programável eram desenvolvidos a partir de esquemáticos, então a partir da década de 90, linguagens de programação específicas, chamadas de HDLs (*Hardware Description Languages*), começaram a ser utilizadas. Atualmente, quase a totalidade destes projetos emprega

alguma HDL [13]. As duas principais HDLs são o Verilog e o VHDL [14], porém as ferramentas de simulação podem ser construídas em qualquer linguagem de propósito geral. Algumas destas linguagens e outras bibliotecas dão total suporte para a modelagem de hardware e software, como portas, sinais e outros. Um exemplo é o SystemC [15], que teve sua primeira versão disponibilizada em 1999 pela OSI (*Open SystemC Initiative*). O SystemC atualmente não é uma linguagem de programação, mas sim uma biblioteca de classes e macros para C++, sendo seu principal objetivo a descrição de hardware, onde é possível conectar linguagem de descrição de sistema (C++) com HDL. Assim podendo implementar arquiteturas executáveis. A principal característica do SystemC é sua alta velocidade de simulação em níveis mais altos de abstração.

1.1 Motivação

A principal motivação do desenvolvimento desde trabalho foi buscar um maior desempenho em simulações. Jerry Krasner [16] em pesquisa realizada em 2003 com empresas de desenvolvimento de sistemas embarcado obteve os seguintes dados:

- Aproximadamente 54% dos projetos são completados depois do tempo proposto;
- O tempo médio de atraso é de 3,9 meses;
- 40% dos projetos não chegam a 30% do proposto na especificação em relação ao desempenho e funcionalidade. Este valor sobe para 57% se considerados os requisitos de custos e cronograma;

Uma das principais causas apontadas, é a utilização de ferramentas não adequadas de projeto e desenvolvimento, que acabam limitando a implementação, testes e simulações. Isso acaba ofuscando uma previsão segura de um projeto. Atualmente, a indústria tem priorizado o uso de protótipos virtuais para diminuir os erros de projeto, possibilitando que a validação de *hardware* e *software* possa ser realizada nos estágios iniciais.

1.2 Objetivo geral

Este trabalho tem como objetivo a criação de um protótipo virtual em SystemC de uma arquitetura multiprocessada com a memória de dados compartilhada entre os processadores.

Para isto será utilizado o simulador SardMIPS [17] que é baseado na arquitetura MIPS. Os processadores serão conectados através de um barramento que controla a comunicação entre os mesmos e o acesso a memória compartilhada.

1.3 Objetivos específicos

- Estudo do SystemC
- Implementação de um barramento em SystemC;
- Criação de uma arquitetura multiprocessada;

1.4 Organização do trabalho

O trabalho tem por sequência, no Capitulo 2, a classificação das arquiteturas multiprocessadas, suas principais características e exemplos das mesmas. No Capitulo 3, é discutido sobre a coerência de *cache*, seus principais protocolos com foco no protocolo MESI. No Capitulo 4, é descrita plataforma virtual desenvolvida, mostrando suas principais características, descrito também o simulador SardMIPS e o modelo do MMCC em qual nosso projeto foi baseado. No Capitulo 5, são mostradas as conclusões e trabalhos futuros.

Capítulo 2

Multiprocessador

A forma mais comum de se classificar sistemas de processadores é de acordo com o fluxo de instruções e dados, esta classificação é introduzida por Flynn [18], onde são propostas as seguintes categorias de sistemas:

- SISD(*single instruction, single data*)- Único fluxo de instrução para um único fluxo de dados, uniprocessadores;
- SIMD(*single instruction, multiple data*)- Único fluxo de instrução para um múltiplo fluxo de dados, processadores vetoriais;
- MISD(*multiple instruction, single data*)- Múltiplo fluxo de instrução para um único fluxo de dados, não existem computadores comerciais desta categoria;
- MIMD(*multiple instruction, multiple data*)- Múltiplo fluxo de instrução para um múltiplo fluxo de dados, multiprocessadores;

A partir da década de 80 as arquiteturas MIMD tem sido a preferência na construção de computadores de uso geral. Dois fatores são os principais causadores deste crescimento [19]:

- Devido a flexibilidade que a arquitetura MIMD proporciona, com as configurações corretas de *software* e *hardware* pode alcançar um grande desempenho e também executar varias tarefas ao mesmo tempo;
- As arquiteturas MIMD oferecem um grande benefício em relação ao custo e performance, pois a maioria dos computadores multiprocessados possuem processadores já utilizados em computadores com apenas um processador;

As arquiteturas MIMD podem ser classificadas em dois tipos, de acordo com a comunicação entre os processadores: as arquiteturas de memória compartilhada (Multiprocessadores) e arquiteturas de memória distribuída (Multicomputadores). A principal diferença entre estes dois tipos é a presença ou ausência do compartilhamento de memória. Esta diferença está diretamente ligada ao modo como são projetados, construídos e programados bem como em sua escala e preço.

2.1 Multicomputadores

Neste tipo de arquitetura cada processador possui sua própria memória local e a comunicação é realizada através da troca de mensagens por uma rede de interconexão, basicamente transferem dados de uma memória a outra através de mensagens, isso pode ser implementado criando uma cópia de uma porção do espaço de endereçamento de dados e envia-la para outro processador. A maior dificuldade da comunicação de mensagens é que ela é desalinhada e de comprimento arbitrário, já em um sistema de memória compartilhada normalmente é orientada para a transferência de blocos de dados alinhados, organizados como blocos de cache. Além disso a execução de aplicações depende muito de como os dados estão alocados no sistema. As principais características desdas arquiteturas são [20]:

- A configuração de *hardware* pode ser simplificada, especialmente se comparada as arquiteturas multiprocessadas escaláveis de memória compartilhada com controle de coerência de *cache*;
- A comunicação é explicita, desta forma obrigando aos programadores e compiladores desta arquitetura trabalhem sobre a comunicação entra os processadores, tornando o desenvolvimento de aplicações para este tipo de arquitetura muito complexo;

2.2 Multiprocessadores

Em arquiteturas multiprocessadas de compartilhamento de memória a comunicação pode ser feita pelo simples fato de um processador executar uma operação de escrita e outro processador efetuar uma leitura sobre o mesmo bloco. É um modelo de fácil entendimento aos

programadores e é aplicável a uma grande área de problemas. Estes são os fatores responsáveis pela sua popularidade. As principais características desta arquitetura são [19]:

- Compatibilidade com mecanismos de compreendimento sobre todos com processadores que se comunicam com a memória compartilhada;
- Facilidade na programação de aplicações e nas compilações;
- Baixo fluxo de comunicação e melhor aproveitamento da banda de comunicação quando utilizado dados pequenos. Isso se dá ao fato de comunicação implícita e do uso do mapeamento de memória para garantir a segurança em *hardware* ao invés de deixar a cargo do sistema operacional;
- A habilidade de diminuir o acesso remoto através do uso de *caches* em *hardware*;

Podem ser classificadas em dois tipos, de acordo com seu acesso a memória, podem prover acessos uniformes UMA (*Uniform Memory Access*), ou não uniformes NUMA (*Non-Uniform Memory Access*).

2.2.1 NUMA (*Non-uniform Memory Access*) - Acesso Não Uniforme à Memória

Este tipo de arquitetura é comumente utilizada em GPUs, existe um módulo para cada processador e a memória é compartilhada entre esses módulos. Existe um controlador em cada módulo, e o acesso a dados pode ser tanto local (dentro do próprio módulo) ou remoto (em outro módulo) e a coerência de *cache* pode ou não ser implementada, quando há controle sobre a coerência de *cache* alguns autores classificam a arquitetura como CC-NUMA (*Cache Coherent Non-uniform Memory Access*) [19]. Um exemplo da arquitetura NUMA pode ser visto na Figura 2.1.

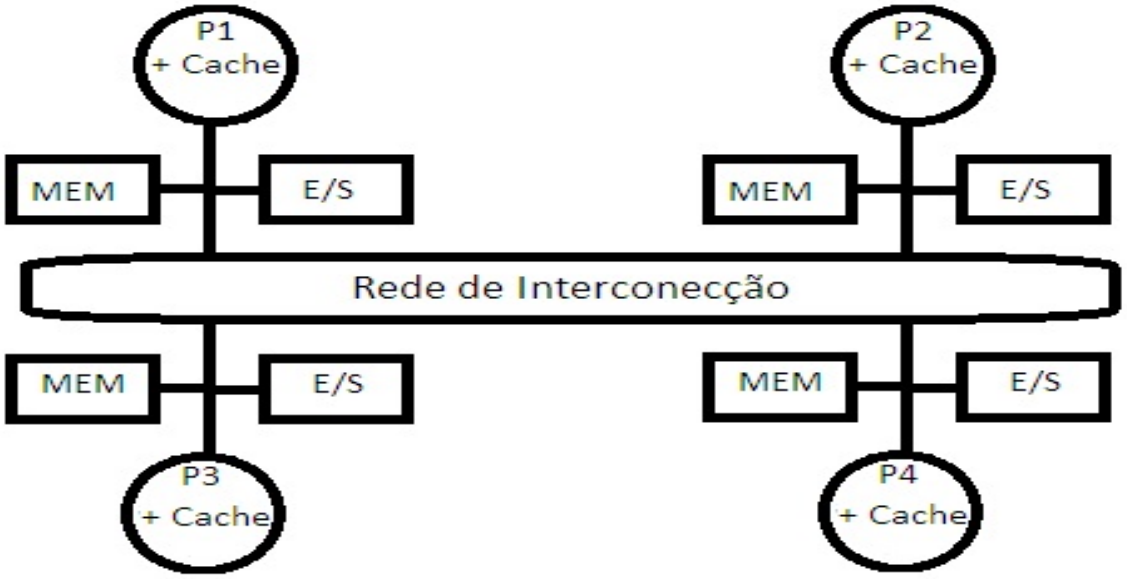

Figura 2.1: Arquitetura NUMA com 4 processadores

Em arquiteturas NUMAs é permitido usar um número maior de processadores, como o *Origin* da *Silicon Graphics*, que suporta até 1024 processadores MIPS R10000 [21] e o sistema que utiliza coerência de *cache* da *Sequent* que é permitido ate 252 processadores *Pentium II* [22], além disso, possuem a vantagens de serem escaláveis.

Cada módulo do sistema pode conter um subsistema UMA, como os sistemas citados anteriormente o *Origin* possui dois processadores MIPS R10000 por módulo, e o sistema da *Sequent* inclui quatro processadores *Pentium II*. Nesses casos a coerência de *cache* é tratada pelo protocolo de diretório.

2.2.2 UMA (*Uniform Memory Access*) - Acesso Uniforme à Memória

A organização UMA que será empregada em nosso trabalho é o tipo mais popular de organização. Neste tipo de arquitetura todos os processadores compartilham a mesma memória e os processadores são interconectados através de um barramento compartilhado. Também tem uma grande vantagem em relação ao custo-benefício, pois o desenvolvimento da arquitetura é mais simples, e o tráfego do barramento pode ser reduzido aumentando o tamanho das *caches*. Pode suportar tanto *caches* compartilhadas quanto as *caches* não compartilhadas. Quando os processadores não compartilham dados o funcionamento é o mesmo de uma arquitetura unipro-

cessada [19].

O compartilhamento centralizado possui uma limitação do número de processadores. Quanto mais processadores, maior o tráfego do barramento, que além de ser usado na troca de dados, também é usado para tratar a coerência de *cache*. Assim é formado um gargalo no barramento comprometendo muito o desempenho do sistema.

Um exemplo da arquitetura UMA é o sistema chamado de *Power Challenge* da *Silicon Graphics* que é limitado a 64 processadores R10000 em um único sistema, pois além desse valor o desempenho é degradado substancialmente [23].

Este tipo de arquitetura possui o problema da incoerência de *cache*, como utiliza mais de um processador e cada processador possui sua própria memória *cache*, suas *caches* podem conter cópias do mesmo bloco da memória, o que pode causar inconsistência dos dados. Podemos analisar da seguinte forma: Supondo uma arquitetura com dois processadores, P1 e P2, ambos com suas *caches*, porém compartilham a memória principal através de um barramento (Figura 2.2).

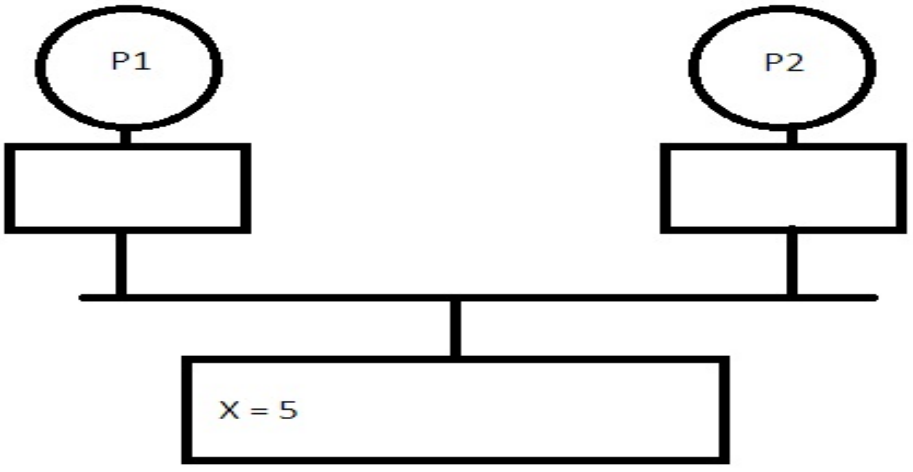

Figura 2.2: Exemplo da inconsistência entre as *caches*

O processador P1 requisita um dado na memória principal (Figura 2.3), este dado é armazenado em sua *cache* para que caso o processador necessite novamente do dado ele possa buscá-lo diretamente em sua *cache*, sem que tenha que buscá-lo na memória principal (mais custoso).

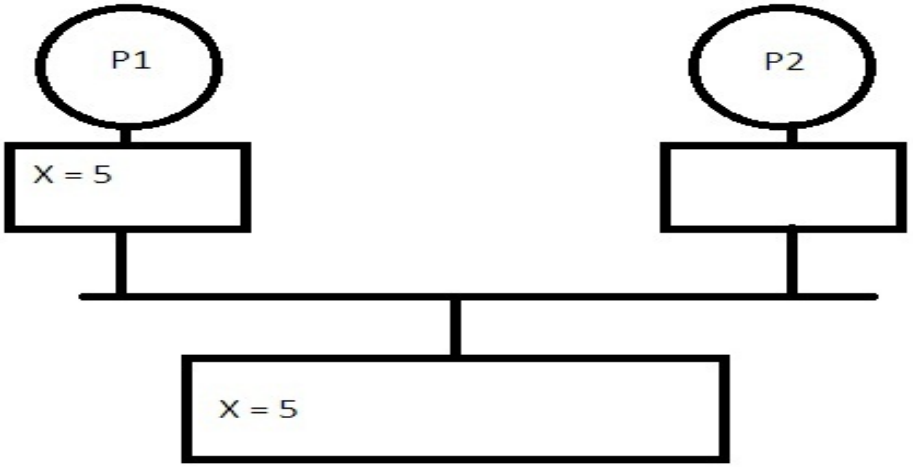

Figura 2.3: Exemplo da inconsistência entre as *caches*

Em um segundo acesso o processador P1 modifica este determinado dado, esta cópia se encontra somente na *cache* deste processador que a modificou (Figura 2.4).

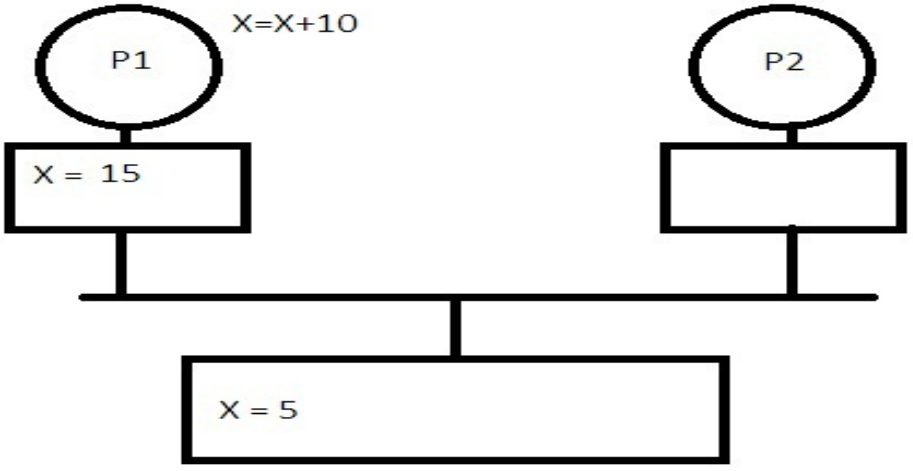

Figura 2.4: Exemplo da inconsistência entre as *caches*

Quando P2 requisita este mesmo dado como não há nada para controlar este acesso, o processador P2 busca o dado na memória principal que está desatualizado e inválido (Figura 2.5).

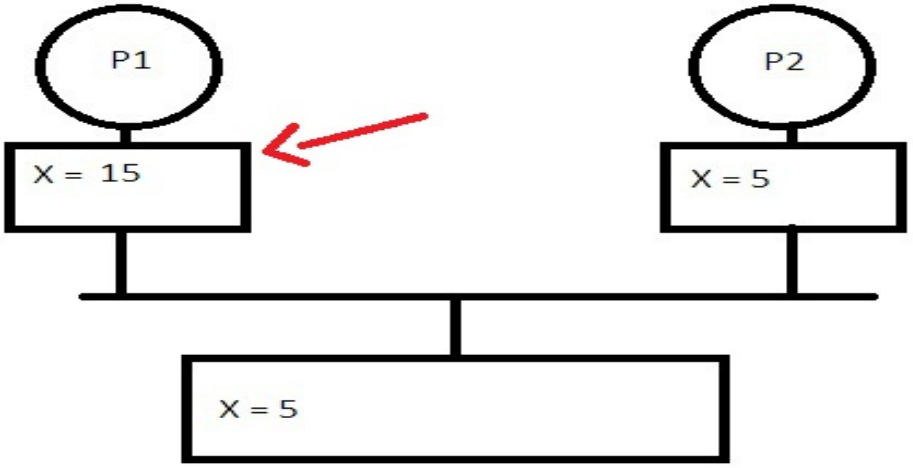

Figura 2.5: Exemplo da inconsistência entre as *caches*

Para este tipo de problema existem diversos protocolos de controle da coerência de *cache* que serão descritos no Capitulo 3.

2.2.3 Exemplos

Como exemplo de multiprocessadores podemos citar os processadores *ARM11* e *Cortex-A9* [1] ambos da empresa ARM, que utilizam para tratamento da coerência de *cache* o protocolo de espionagem MESI que será descrito no capitulo 3. O protocolo é implementado e gerenciado pelo chamado *Snoop Control Unit* (SCU) que age como um barramento. Este barramento monitora o acesso entre as *caches* L1 do sistema e o próximo nível hierárquico de memória. Embora estes processadores tenham que manter a compatibilidade com o MESI, eles buscam implementar estratégias visando maior desempenho e otimização, como é mostrado abaixo:

- *Direct Data Intervention* (DDI): O SCU mantem uma cópia do índice de RAM de todas as *caches* do sistema. Isso permite uma detecção eficaz se o bloco da *cache* for requisitado por outra *cache* no domínio de coerência, antes de buscar o bloco no nível hierárquico superior de memória;

- *Cache-to-cache* Migration: Se o SCU encontra o bloco da *cache* requisitado por um processador em outro núcleo, ele mesmo copia ou transfere o bloco da *cache* encontrada diretamente até o processador requisitante, sem passar pela memória;

Este tipo de controle pode ser visto na Figura 2.6.

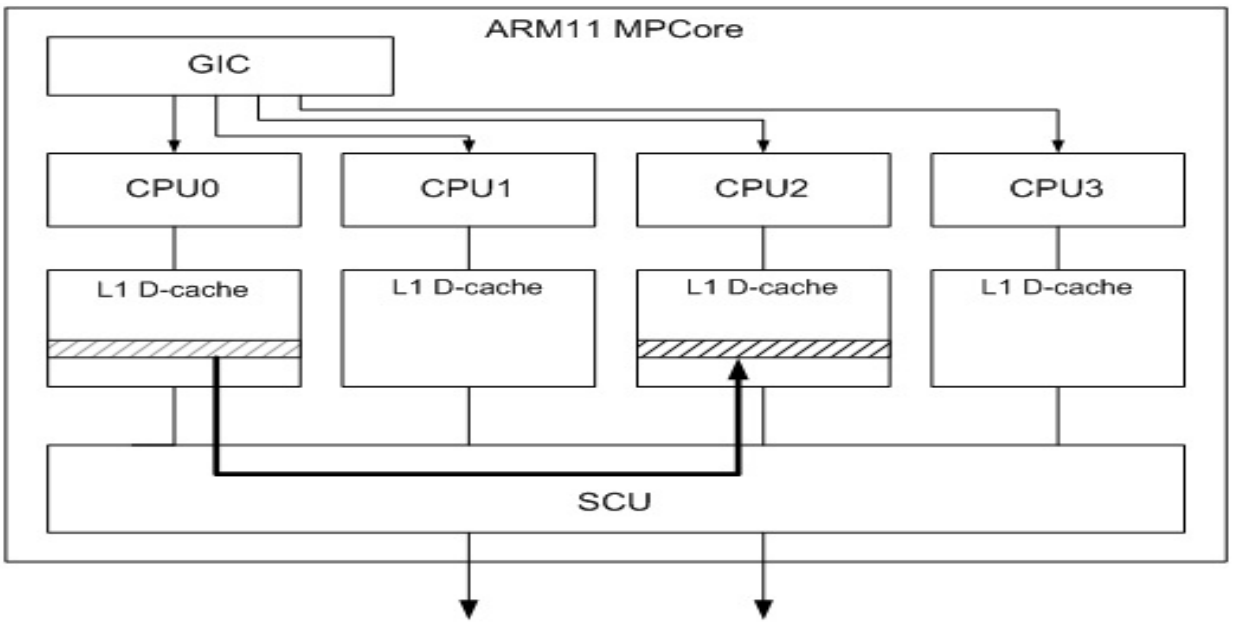

Figura 2.6: *Cache-to-cache* Migration, ARM11 [1]

Outro exemplo é o processador *Intel Nehalem* de 8 núcleos. Este processador é baseado na arquitetura NUMA, utiliza 2 módulos *Nehalem* com 4 núcleos cada (*Quadcore*) como pode ser observado na Figura 2.7.

O processador *Nehalem* é baseado em arquitetura NUMA com coerência de *cache*. Para assegurar a coerência entre todas as *caches*, a *cache* L3 possui algumas *tags* adicionais para rastrear de qual é o núcleo de origem do dado. Se o dado é modificado na *cache* L3, ela sabe se a modificação veio de outro núcleo, assim ela sabe que precisa atualizar a *cache* L1/L2 do núcleo origem do dado antigo.

Este controle da coerência de *cache* é feito através de uma modificação do protocolo por espionagem MESI, chamado de MESIF, devido a inclusão de um novo estado chamado de *forward*. O estado *forward* indica que a *cache* que contenha o bloco nesse estado, fica responsável por atualizar todas as outras cópias do bloco compartilhado [2].

Um exemplo de arquitetura UMA será o MMCC que será descrito no Capitulo 4.

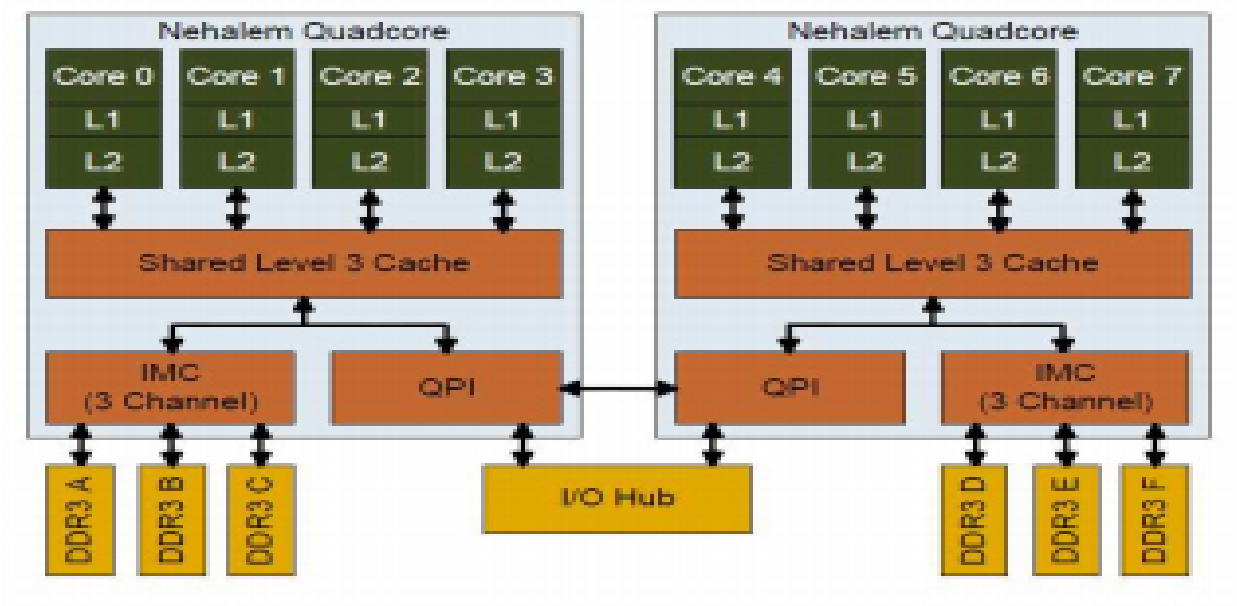

Figura 2.7: Processador Intel Nehalem com 8 núcleos [2]

Capítulo 3

Coerência de *Cache*

Para controlar a coerência de *cache*, existem dois tipos de protocolos: o protocolo por espionagem (*snooping*) que mantém a coerência através do rastreamento do barramento do sistema, onde cada processador fica responsável por monitorar o barramento e a sua cópia em *cache*, e o protocolo por diretório (*directory*) que diferentemente do protocolo por espionagem obtém a coerência de uma forma centralizada, ou seja, armazena as informações das cópias e do compartilhamento dos blocos da *cache* de todos os processadores de uma maneira centralizada, em um local chamado diretório, onde esses diretórios são distribuídos. A seguir são apresentados os protocolos de coerência, mostrando seus respectivos funcionamentos.

3.1 Protocolo por diretório

A idéia deste protocolo é manter um banco de dados contendo a localização de cada bloco da *cache* e em que estado ele está. Quando um bloco é referenciado, o banco de dados é pesquisado para saber onde ele está e se ele está limpo ou sujo (modificado). Este banco de dados é chamado de diretório. Como a cada requisição de bloco da *cache* é varrido o diretório, a resposta do mesmo deve ser rápida para permitir um bom desempenho do protocolo. Normalmente este protocolo é utilizado em arquiteturas NUMA onde em cada nó da arquitetura possui um diretório [20].

A seguir é demonstrado o funcionamento do protocolo utilizando a arquitetura da Figura 3.1.

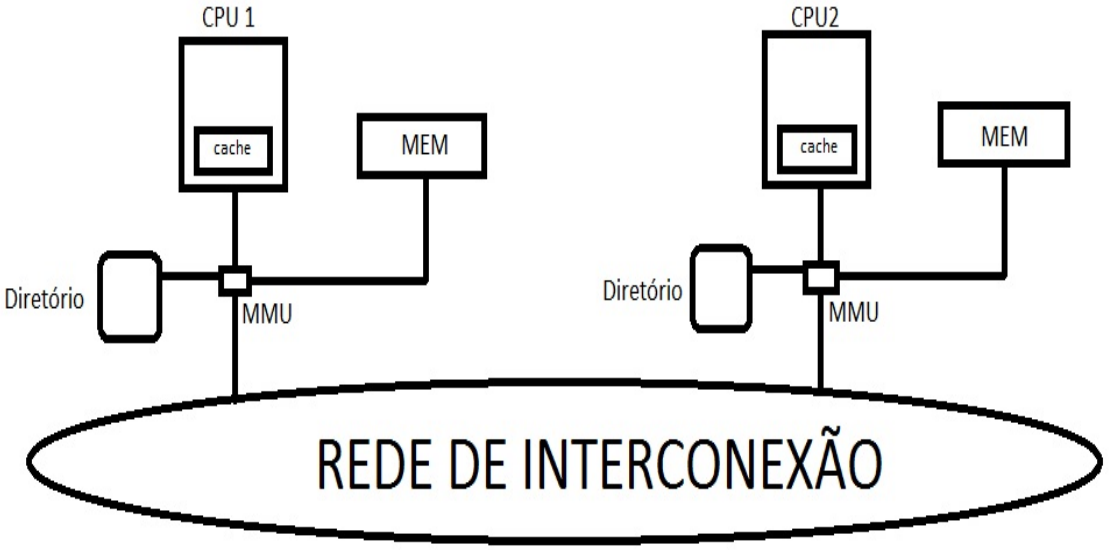

Figura 3.1: Arquitetura NUMA com dois nós

1. CPU1 requisita um dado passando um endereço ao MMU (Unidade de Gerenciamento de Memória), o MMU quebra o endereço e mostra que o endereço requisitado encontra-se no CPU2 no bloco 2, então é enviada uma mensagem ao CPU2 através da rede de interconexão perguntando se o bloco 2 está em cache;

2. Quando a mensagem chega ao nó da CPU2 o endereço passado na mensagem (bloco 2) é roteado para o diretório do nó e acessado o bloco 2 do diretório caso não contenha nenhum endereço, o diretório busca o endereço em sua memória local e envia ao CPU1, então atualiza o bloco 2 do diretório informando que o bloco se encontra no nó da CPU1. Caso houvesse um endereço de outro nó no bloco 2 do diretório da CPU2 este nó seria instruído a passar este dado para a CPU1 e invalidar sua *cache* local.

3.2 Protocolo por Espionagem

Entre os principais protocolos baseados em espionagem temos MSI [24], MESI [25], MOESI [26] e DRAGON [27]. Nesse tipo de protocolo, o estado dos blocos de dados da *cache* podem ser alterados, ou por ações do próprio processador local, ou através de ações de

outros processadores que incidem no barramento compartilhado. Entre os protocolos de espionagem os mais utilizados são o MESI e o MOESI, esses dois protocolos possuem uma grande diferença, o MESI pode ser implementado em circuitos mais simples como de quatro estados, ao invés dos cinco estados requisitados pelo MOESI, consequentemente o MESI utiliza apenas dois bits de controle, contra três do MOESI.

Esta diferença do número de bits de controle reflete-se em menor quantidade de memória para armazenamento e em uma lógica combinacional mais simples e veloz. [13] A seguir detalharemos o protocolo MESI a ser utilizado em nosso trabalho.

3.2.1 MESI

O protocolo MESI tem esse nome pois o bloco de dados pode ter quatro estados, *modified*, *exclusive*, *shared* e *invalid*, é usado amplamente em multiprocessadores comerciais tais como *Pentium* e *PowerPC* [23]. Utilizando este protocolo a memória *cache* inclui dois bits de controle de estado por rótulo, assim cada bloco da *cache* pode estar em um dos quatro estados proposto pelo protocolo, que são descritos a seguir.

O estado *modified* indica que o bloco da *cache* foi modificado, ou seja, está diferente da memória principal, portanto somente esta *cache* possui o bloco atualizado. O estado *exclusive* indica que somente este *cache* possui o bloco e é igual ao bloco da memória principal. O estado *shared* indica que o bloco da *cache* está compartilhado por outras *caches* e está igual à memória principal. O estado *invalid* indica que o bloco da *cache* não é valido. Todos os estados são controlados a partir de sinais enviados ao barramento compartilhado entre os processadores.

A Figura 3.2 representa o protocolo MESI, sendo a máquina de estados da esquerda representa as transições do processador e a máquina de estados à direita o monitoramento do barramento (ações de outros processadores), os círculos representam os estados do protocolo (*modified*, *exclusive*, *shared* e *invalid*), as setas as transições entre os estados, em cada transição é indicado qual operação a transição representa, no diagrama de estados temos as seguintes transições:

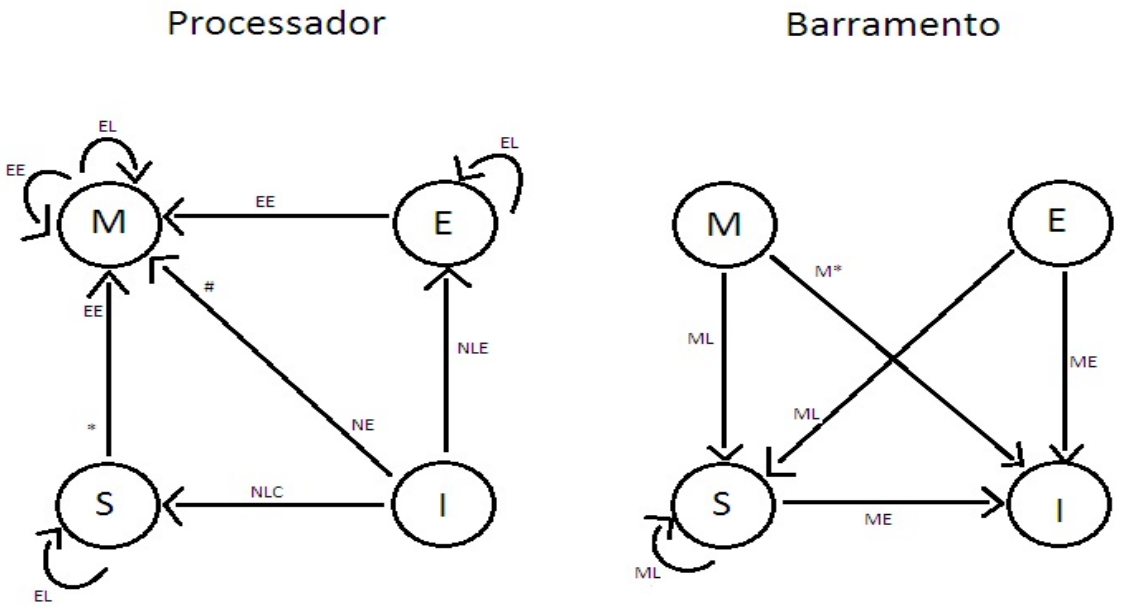

Figura 3.2: Diagrama protocolo MESI

- EL: Indica que o processador está executando uma leitura sobre um bloco encontrado na *cache* local;
- *: É enviado um sinal de invalidação das demais cópias para o barramento;
- EE: Indica que o processador está executando uma escrita sobre o bloco encontrado na *cache* local;
- NE: Indica que o processador está executando uma escrita, porem não encontrou o bloco em sua *cache*;
- #: Indica que o processador está executando uma leitura no barramento com a intenção de modificação;
- NLE: Indica que o processador está executando uma operação de leitura, porem não encontrou o bloco em sua *cache*, e nenhuma outra *cache* o possui;
- NLC: Indica que o processador está executando uma operação de leitura, porem não encontra o dado em sua *cache* mas existe pelo menos uma outra *cache* no sistema que possui uma cópia deste bloco;

- ML: Indica que outro processador está requisitando um leitura sobre o bloco através do monitoramento do barramento;
- ME: Indica que outro processador está requisitando uma escrita sobre o bloco através do monitoramento do barramento;
- M*: Indica que outro processador está fazendo uma leitura com a intenção de modificar o bloco;

Supondo uma arquitetura multiprocessada com dois processadores (P1 e P2), temos os seguintes passos em uma situação do uso do protocolo MESI:

1. P1 e P2 não tem nenhum dado em suas *caches* (Figura 3.3);

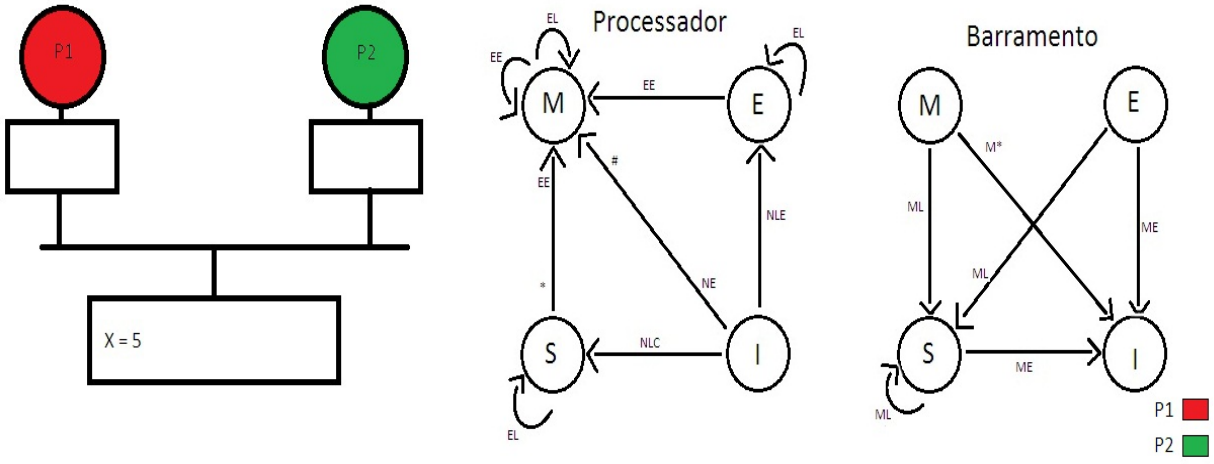

Figura 3.3: Exemplo protocolo MESI (a)

2. P1 requisita uma leitura de um bloco que contenha o dado "X", é buscado o bloco em sua *cache*, como não foi encontrado é efetuada uma leitura sobre o barramento à procura de uma cópia do bloco, primeiro verifica-se se há alguma *cache* com o bloco que esteja com a cópia válida, como não há nenhuma *cache* com o bloco, P1 busca o bloco da memória principal (transição NLE do processador);
3. É armazenado o bloco que contém o dado "X" na *cache* de P1 com o estado *exclusive* (3.4), pois nenhuma outra *cache* do sistema possui uma cópia do mesmo bloco. P1 pode executar várias leituras sobre o bloco (EL), pois a cópia em sua *cache* é válida e ainda continuará no estado *exclusive* (Figura 3.4);

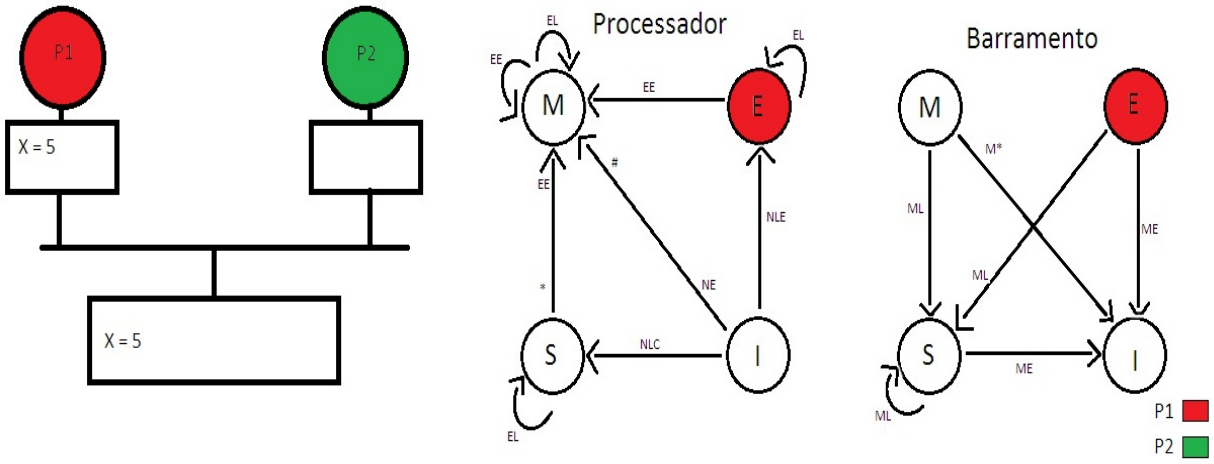

Figura 3.4: Exemplo protocolo MESI (b)

4. P1 requisita uma escrita sobre o bloco que contém o dado "X" como P1 possui uma copia do bloco em sua cache, é alterado o valor do bloco somente em sua cache local, consequentemente o estado do bloco vai para *modified* (Figura 3.5);

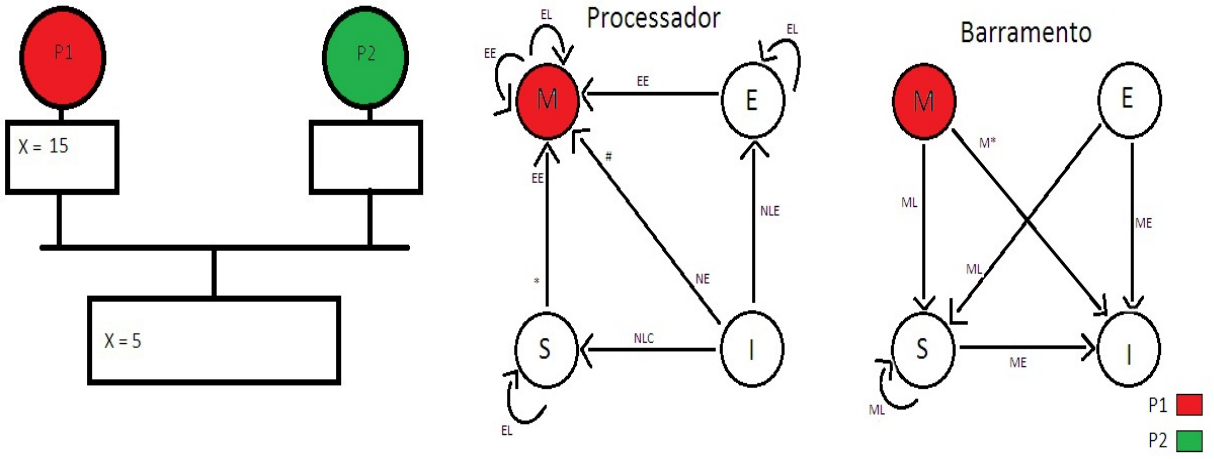

Figura 3.5: Exemplo protocolo MESI (c)

5. P2 requisita uma leitura sobre o bloco que contém o dado "X", então é efetuada uma busca em sua *cache*, como não é encontrado o bloco P2 efetua uma leitura sobre o barramento, é encontrada a cópia na *cache* de P1 (NLC), P1 recebe o sinal de que existe outro processador requisitando uma leitura sobre o bloco em sua *cache* (ML) portando é modificado o estado do bloco em P1 para *shared* e o bloco de P2 também para o estado *shared* (Figura 3.6);

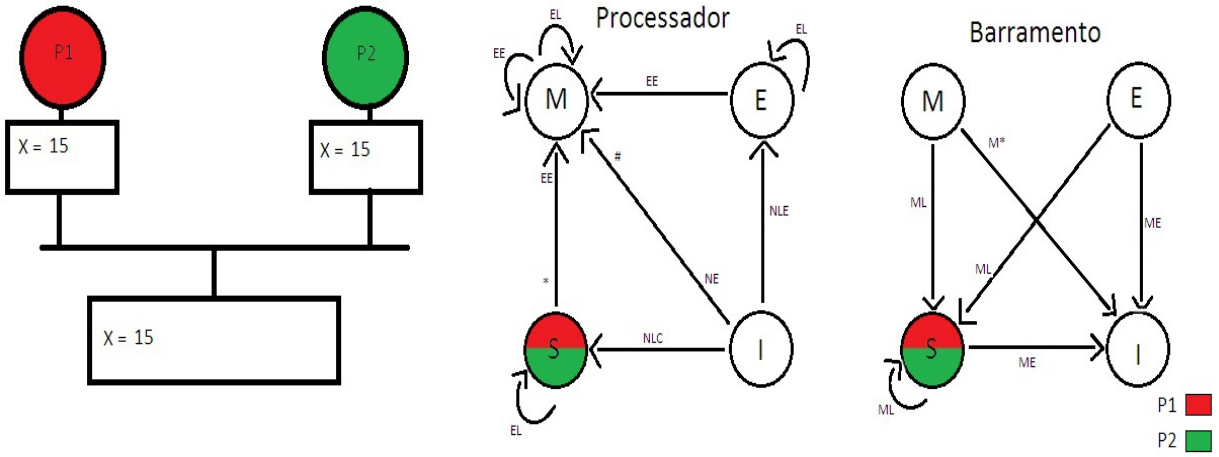

Figura 3.6: Exemplo protocolo MESI (d)

Este protocolo pode ser facilmente implementado através de um barramento que funciona como uma máquina de estados, este barramento fica entre a *cache* do processador e o barramento de controle, cada bloco da *cache* possui dois *bytes* para indicar seu estado (00 - *modified*, 01 - *exclusive*, 10 - *shared*, 11- *invalid*). Quando um processador faz uma requisição ele primeiramente verifica se o dado é encontrado em sua *cache* local, então o barramento do protocolo envia a requisição através do barramento compartilhado de controle aos outros barramentos controladores de *caches* remotas, esta requisição pode ou não mudar o estado do bloco em outras *caches*, então é retornado ao barramento requisitante para qual estado o bloco deve ir.

Capítulo 4

Implementação

Este trabalho tem como objetivo a implementação de um protótipo virtual de arquiteturas multiprocessadas que compartilham a memória de dados através de um barramento de controle. Utilizamos o processador *SardMIPS* que já foi modelado em SystemC. Foi escolhida esta ferramenta pois ela nos permite modelar tanto componentes de *hardware* como de *software* em C++, assim facilitando o processo de criação do nosso protótipo e de aplicações. Podemos criar blocos hierárquicos de *hardware* assim o protótipo pode trabalhar de forma modular, consequentemente implicando em uma grande vantagem, na correção de erros, testes e simulações.

Atualmente a arquitetura MMCC está descrita em VHDL. Porém a simulação do MMCC em VHDL limita a velocidade e as correções de erros. Por este fato, tendo como base o MMCC, foi desenvolvido um protótipo virtual em SystemC visando acelerar a simulação da arquitetura e facilitar a implementação tanto da arquitetura em si como de aplicações para a mesma. Este protótipo será descrito na seção Arquitetura desenvolvida.

4.1 MMCC

A arquitetura MMCC (Multiprocessador Minimalista com *Caches* Coerentes) foi desenvolvida por Jorge Tortato Junior e Roberto Hexsel tendo como objetivo principal o desenvolvimento em VHDL de um *Multiprocessor System-on-Chip* (MPSoC) Minimalista com *Caches* Coerentes (MMCC). Nessa implementação cada núcleo consiste em um *miniMIPS* [28] baseado no processador MIPS R2000, e seu sistema de *cache* e gerenciamento de memória. Também foi utilizado o protocolo de coerência de *cache* por espionagem MESI. O acesso à memória e as *caches* remotas ocorrem através de um barramento compartilhado.

Existem três tipos de interfaces conectadas ao barramento, interface de *cache*, de *snoop* (protocolo MESI) e de controlador de acesso. A interface de *cache* inicia as requisições de leitura/escrita, que são atendidas pela interface de *snoop* ou de controle de acesso. Já a interface de *snoop* e de controle de acesso, diferenciam-se apenas porque na interface de *snoop* é utilizado um sinal indicando o estado do bloco acessado.

A unidade de gerenciamento de memória (MMU) foi adicionada ao *miniMIPS* e tem os objetivos de permitir que o software possa ser realocado na memória e de prover proteções básicas de acesso. Também foram incorporadas caches de instruções e de dados, sendo a *cache* de dados com suporte à coerência, e a *cache* de instruções uma versão simplificada da *cache* de dados sem suporte à coerência [13]. As arquiteturas possuem configurações pré-definidas, com um, dois, quatro e oito núcleos de processamento. A arquitetura MMCC pode ser visualizada na figura 4.1 [13].

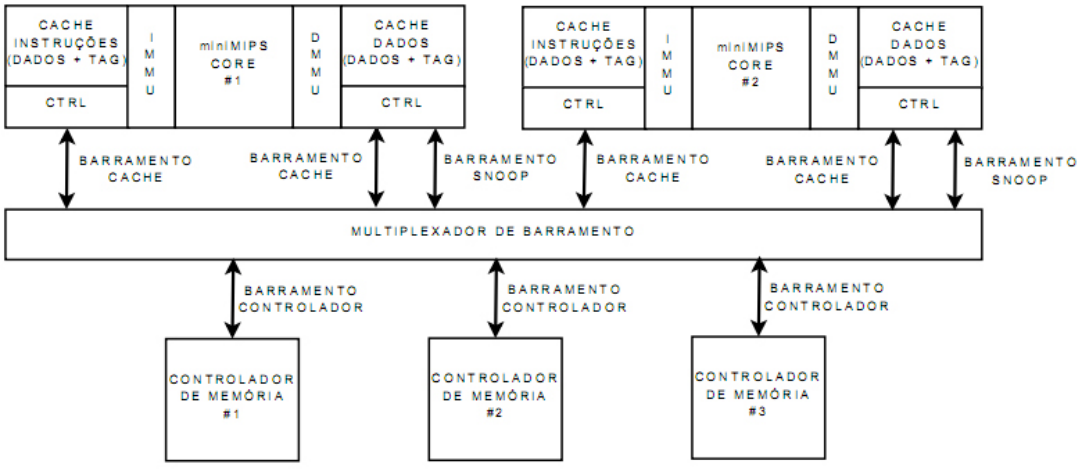

Figura 4.1: MMCC com 2 núcleos MiniMips

O modelo do MMCC proposto por Jorge Tortato Junior e Roberto Hexsel é sintetizável, sendo possível sua implementação em FPGAs [13].

4.2 SardMIPS

Este modelo de simulação, assim como o modelo *miniMIPS*, é baseado no processador MIPS R2000 porem implementado utilizando a biblioteca SystemC ao invés da linguagem VHDL. A arquitetura MIPS foi inicializada pelo professor Hennessy da universidade de Stan-

ford no ano de 1981. A principal característica do processador R2000 é o uso de um *pipeline* de cinco estágios, ou seja, a cada sinal de *clock* a instrução vai para o próximo estágio, desta maneira podemos colocar até cinco instruções no microprocessador em estágios diferentes, aumentando consideravelmente o desempenho. Seus cinco estágios são o estágio de busca, estágio de decodificação de instruções, estágio de execução de instruções, estágio de acesso à memória e o estágio de *Write back*.

A arquitetura MIPS contém 32 registradores, onde cada um possui um tamanho de 32 bits, onde os grupos de 32 bits são nomeados de *word*. Os respectivos registradores podem ser visualizados na Tabela 4.1.

Registrador	**Função**
$zero	Contem apenas o valor constante "0"
$v0-$v1	Utilizado em retornos de procedimentos e funções
$a0-$a3	Utilizados para parâmetros de funções
$t0-$t9	Registradores temporários
$s0-$s7	Registradores salvos
$gp	Utilizado como um ponteiro global
$sp	Utilizado como ponteiro apontando a base da pilha
$fp	Utilizado como ponteiro apontando o topo da pilha
$ra	Utilizado como ponteiro apontando para o endereço onde a função deve retornar

Tabela 4.1: Principais registradores MIPS [3]

As instruções da arquitetura MIPS são divididas em [5]:

- Instruções de operações aritméticas como adição (*add*) subtração (*sub*), essas operações são feitas entre 2 registradores, e armazenado o resultado em um terceiro registrador, por exemplo: add $s0,$t0,$s2 esta operação é equivalente a $s0 = $t0 + $s2. Podemos também executar operações aritméticas entre um registrador e um numero constante utilizando a instrução de adição com imediato (*addi*);

- Instruções de transferência de dados, basicamente composto por duas instruções a instrução *load*, que carrega determinado dado de um endereço da memória, e a instrução *store*, que armazena determinado dado em um endereço da memoria. As instruções só se diferenciam pelo tipo de dado a ser transferido, *word* (32 bits) utilizasse a instrução *lw* (*load word*) e *sw* (*store word*), nessas intruções existem também os tipos de dados *half*

que indicam a metade de um *word* (16 bits) e *byte* (8 bits). Um exemplo da operação load word: lw $t0, $s5 que é equivalente a $t0 = $s5;

- Instruções lógicas, contém as instruções de lógica básica, *and*, *or*, *nor*, *shift left*, *shift right*;

- Instruções condicionais, contém as instruções de condições, quando igual (*beq*), quando diferente (*bne*);

- Instruções de *jump*, *jump* (j) utilizada para efetuar um "salto" até o endereço utilizado na instrução, *jump register* (jr) utilizado geralmente com o registrador $ra para retornar de uma função;

As instruções mais importantes para a coerência de *cache*, são as instruções de *load* e *store*, pois são as únicas instruções que tem acesso a memória, serão essas instruções que trabalharão em conjunto com o protocolo MESI.

O SardMIPS em sua versão atual encontra-se modelado basicamente da seguinte maneira: um processador acessando diretamente a memória de dados e de instruções, como pode ser visto na Figura 4.2.

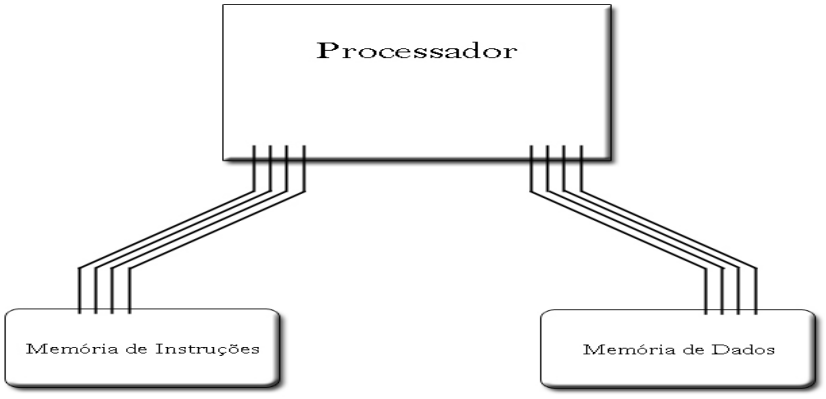

Figura 4.2: Versão atual do modelo SardMIPS

4.3 Arquitetura desenvolvida

O projeto do multiprocessador se encontra da seguinte forma, foi desenvolvido o barramento de controle ao acesso da memória de dados, inserido um novo processador e uma nova memória de instruções, assim criando uma arquitetura com dois processadores onde cada processador possui sua própria memória de instruções porém compartilham a memória de dados através do barramento de controle. Como utilizamos o SystemC para modelar nossa arquitetura, a inserção dos novos módulos de processador e memória de instruções foi muito simples só foi necessário criar novas instancias dos módulos do SardMIPS. O barramento de controle de acesso a memória de dados será descrito na próxima seção. Por fim o nosso protótipo em uma representação básica pode ser visualizado 4.3.

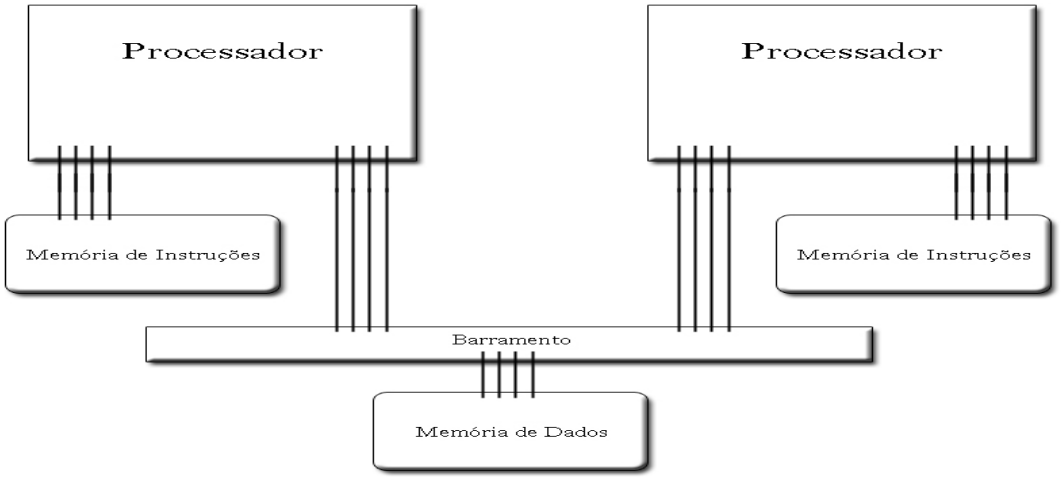

Figura 4.3: Versão atual do modelo multiprocessado

4.3.1 Barramento de controle de acesso

O desenvolvimento do barramento teve início com a definição das portas, como no modelo original do SardMIPS o processador comunicava-se diretamente com a memória de dados, agora o barramento possui essas mesmas conexões pois está intermediando esta comunicação, porém tendo um conjunto de portas para cada processador do sistema. Como o barramento possui n conjuntos de portas ligadas aos processadores (um conjunto para cada processador), utilizamos vetores de portas para facilitar o desenvolvimento e ampliação da arquitetura, o conjunto de portas ligadas ao processador são apresentadas na Tabela 4.2.

Nome	Tipo	Descrição
datareq[n]	sc_in<sc_logic>	É o sinal indicando que o processador *n* está requisitando uma operação para o barramento.
dataaddr[n]	sc_in<sc_uint<32> >	Indica em qual endereço da memória que o processador *n* deseja executar sua operação.
datarw[n]	sc_in<sc_logic>	Indica se a operação requisitada pelo processador *n* ao barramento é de escrita (1) ou de leitura (0).
datawrite[n]	sc_in<sc_lv<32> >	Caso a operação requisitada pelo processador *n* seja de escrita, o processador passa através desta porta qual valor deseja escrever no endereço da memória.
datahold[n]	sc_out<bool>	É o sinal que deixa o processador *n* em espera (quando sinal é levantado) ate que a operação requisitada seja executada.
dataabs[n]	sc_in<sc_lv<2> >	É um seletor do tipo de byte a ser escrito01 para byte e 10 para halfword.
addrl2proc[n]	sc_out<sc_logic>	Retorno do barramento ao processador *n* indicando que o endereço de leitura é desalinhado.
addrs2proc[n]	sc_out<sc_logic>	Retorno do barramento ao processador *n* indicando que o endereço de escrita é desalinhado.
page_fault2proc[n]	sc_out<sc_logic>	Retorno do barramento ao processador *n* indicando que houve falha no endereço.
dataread[n]	sc_out<sc_lv<32> >	Retorno do barramento ao processador *n* contendo o dado lido do endereço.

Tabela 4.2: Portas com conexão aos processadores

Independentemente da quantia de processadores, sempre haverá o mesmo numero de portas do barramento que se comunicam com a memória. Primeiramente o barramento seleciona qual requisição será atendida para depois repassar a requisição para as portas de comunicação com a memória. Essas portas podem ser visualizadas na Tabela 4.3.

Nome	Tipo	Descrição
datareq2mem	sc_out<sc_logic>	É o sinal indicando que o barramento, está requisitando uma operação para a memória.
dataaddr2mem	sc_out<sc_uint<32> >	Indica em qual endereço da memória que o barramento deseja executar sua operação.
datarw2mem	sc_out<sc_logic>	Indica se a operação requisitada pelo barramento a memória é de escrita (1) ou de leitura (0).
datawrite2mem	sc_out<sc_lv<32> >	Caso a operação requisitada pelo barramento seja de escrita, o barramento passa através desta porta qual valor deseja escrever no endereço da memória.
memhold	sc_in<bool>	É o sinal que deixa o barramento em espera (quando sinal é levantado) ate que a operação requisitada seja executada.
databs2mem	sc_out<sc_lv<2> >	É um seletor do tipo de byte a ser escrito01 para byte e 10 para halfword.
addrl	sc_in<sc_logic>	Retorno da memória ao barramento indicando que o endereço de leitura é desalinhado.
addrs	sc_in<sc_logic>	Retorno da memória ao barramento indicando que o endereço de escrita é desalinhado.
page_fault	sc_in<sc_logic>	Retorno da memória ao barramento indicando que houve falha no endereço.
dataread2mem	sc_in<sc_lv<32> >	Retorno da memória ao barramento contendo o dado lido do endereço.

Tabela 4.3: Portas com conexão a memória

O controle do acesso a memória é garantido através de dois métodos internos do barramento:

- get_request() - Este método é acionado quando as entradas datareq são modificadas, ou seja, quando é levantado o sinal datareq no vetor de entradas é disparado este método. Também sensível a mudanças nas entradas dataaddr e datawrite para o auxilio ao controle das requisições. O método possui um vetor de requisições, o tamanho deste vetor é igual ao número de processadores, cada posição deste vetor representa um processador e no momento que é disparado o método, são varridas as entradas datareq para encontrar qual processador fez a requisição e adicionado ao vetor de requisições, e por fim é levantado o

datahold deste processador, forçando assim o processador ficar em espera;

- handle_request() - Este método é responsável por controlar o vetor de requisições, bem como o gerenciamento do acesso a memória, ele é sensível ao sinal do *clock*, ou seja, a cada descida do *clock* é disparado este método. Esta maquina de estados é apresentada na Figura 4.4.

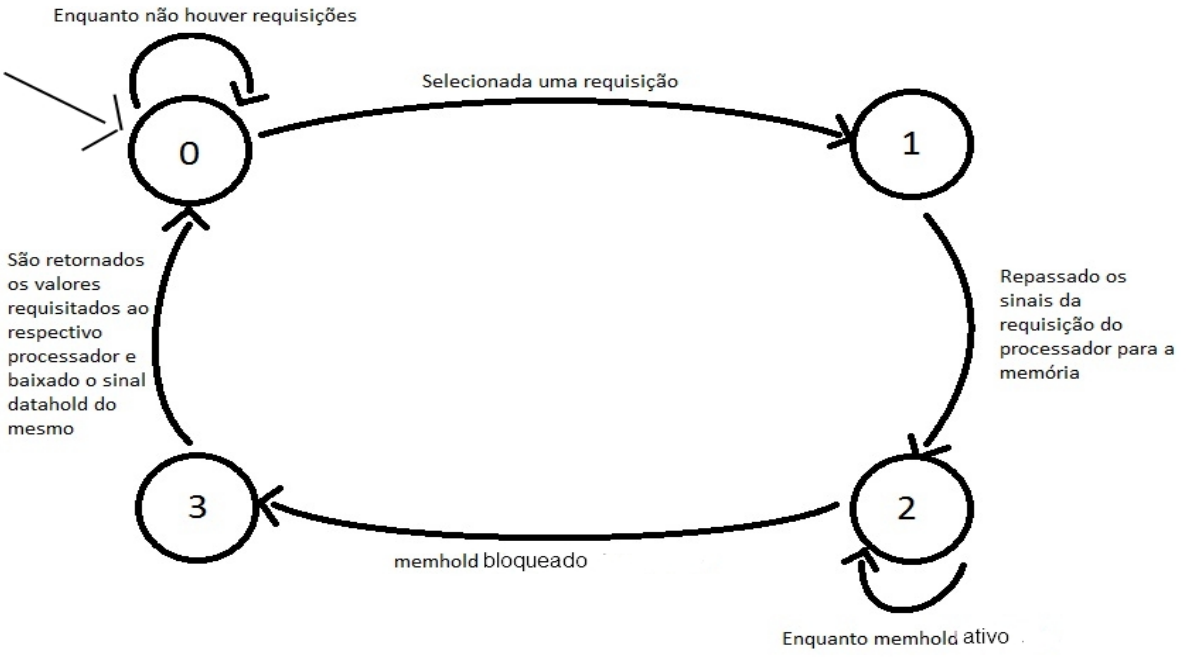

Figura 4.4: Máquina de estados do método handle_request()

  - Estado 0 - É o estado que seleciona a requisição no vetor de requisições;
  - Estado 1 - Neste estado o barramento captura os dados do conjunto de entradas do processador requisitante que foi escolhido no estado anterior e então envia a requisição para a memória, com os respectivos parâmetros;
  - Estado 2 - É o estado responsável por tratar a latência da memória;
  - Estado 3 - Neste estado o barramento recebe o retorno da memória e repassa para o processador (caso seja uma leitura), abaixa o sinal datahold deste processador, retira a requisição concluída do vetor de requisições e retorna ao estado 0;

Graças ao barramento, as requisições dos processadores e o controle do acesso a memória foi bem sucedido. Porém o controle deste acesso é feito dando prioridade aos primeiros processadores, pois como cada processador é representado por uma posição no vetor de requisições, quando o método handle_request() é acionado e está em seu estado 0 (estado em que seleciona a requisição a ser atendida) , a varredura deste vetor é feita através de um *for* e a primeira requisição que ele encontra é escolhida. O código do barramento pode ser visto no Apêndice A.1.

4.3.2 Testes

Os testes foram feitos através de simulações para testar o comportamento do barramento em relação ao sistema. Foi implementado um código de testes simples onde uma variável inteira recebe o valor 10, este mesmo programa é executado nos dois processadores em paralelo, este código de teste pode ser visualizado no Apêndice A.2.

O SardMIPS tem alguns endereços da memória que são reservados para ações específicas, dentre os endereços reservados temos como principais:

- 0x00009000 - Endereço da memória para entrada e saida;
- 0x7FFFFFFC - Endereço de parada, ao ser escrito algum dado neste endereço a simulação é finalizada;

4.3.3 Simulação

Em simulações de protótipos virtuais em SystemC pode-se visualizar os fluxos de sinais através do *wave form*, nele são apresentados todos os sinais que foram previamente declarados, em função do tempo de simulação. A simulação da arquitetura multiprocessada criada, tem como objetivo mostrar o controle feito pelo barramento para garantir a exclusividade do acesso à memória, a simulação será detalhada a seguir, onde a linha rosa da imagens representam o tempo corrente da simulação, a linha em chave vermelha representa o conjunto de sinais do processador zero e a linha em chave verde do processador um. Antes de começar a simulação por padrão o processador faz alguns ajustes de endereços da pilha, por este fato as imagens apresentadas são posteriores a estes ajustes.

1. Como o teste utilizado na simulação é executado paralelamente nos dois processadores, pode ser visto que na linha de execução os dois processadores fazem a requisição ao barramento simultaneamente, através do sinal datareq, com seus respectivos sinais dda requisição (leitura ou escrita, endereço e dado de entrada/saída), que no caso são iguais (Figura 4.5);

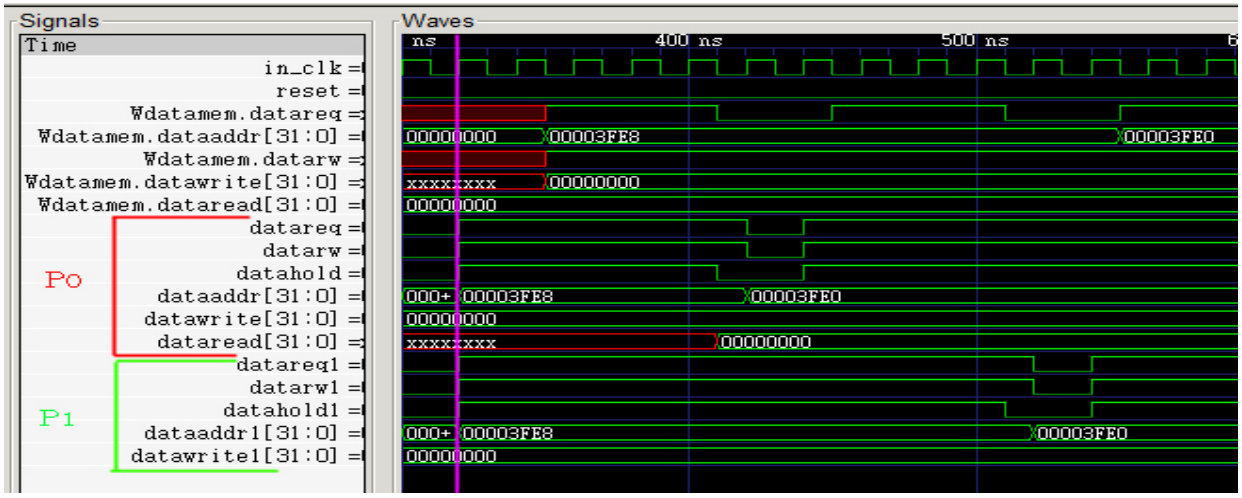

Figura 4.5: Simulação da arquitetura (a)

2. O barramento então aloca na lista de requisições as duas requisições e levanta o sinal datahold indicando que os processadores ficarão em espera aguardando o retorno do barramento. Usando o critério de menor id o barramento seleciona a requisição do processador zero, e faz uma requisição à memória. Os sinais são repassados do processador selecionado para a memória, como apresentado na Figura 4.6;

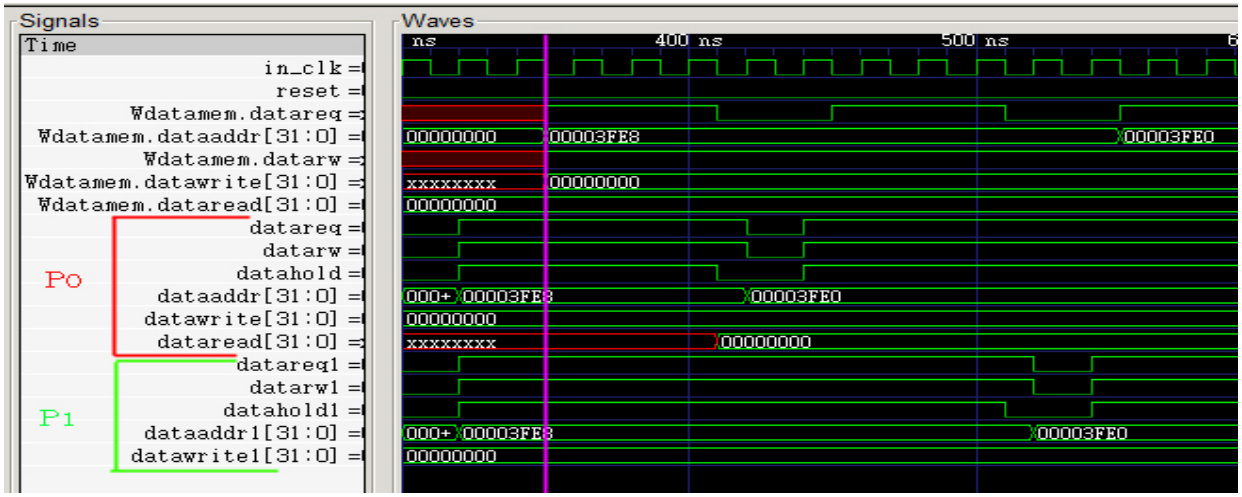

Figura 4.6: Simulação da arquitetura (b)

3. Quando é recebido o retorno da memória, o barramento então desce o sinal de requisição à memória (Wdatamem.datareq) e o sinal datahold do processador (Figura 4.7), e então retorna os valores ao processador requisitante. Adicionalmente, o barramento retira a requisição atendida da lista de requisições e o processador desce o sinal datareq ao barramento (Figura 4.8);

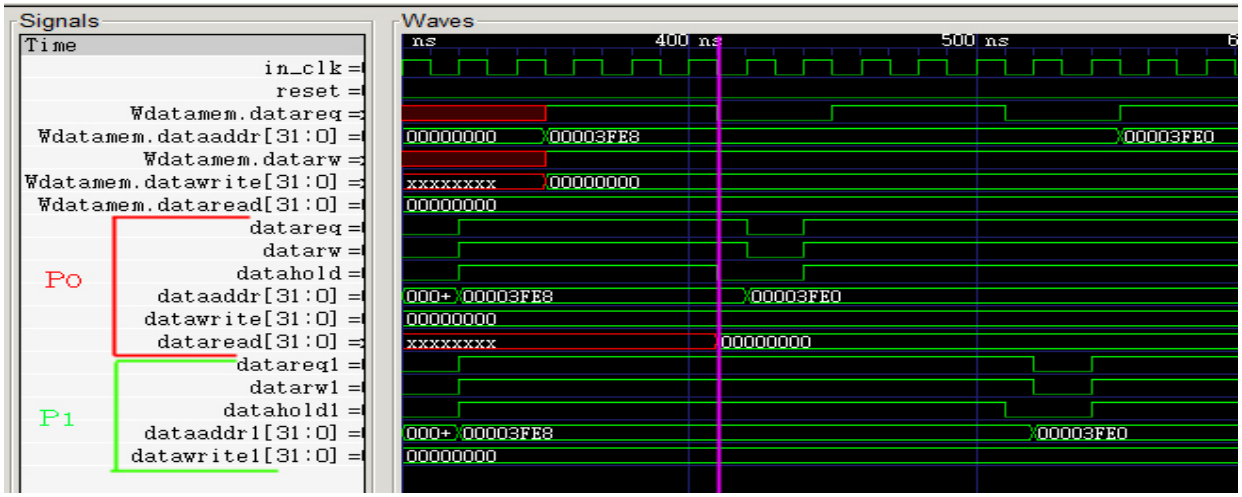

Figura 4.7: Simulação da arquitetura (c)

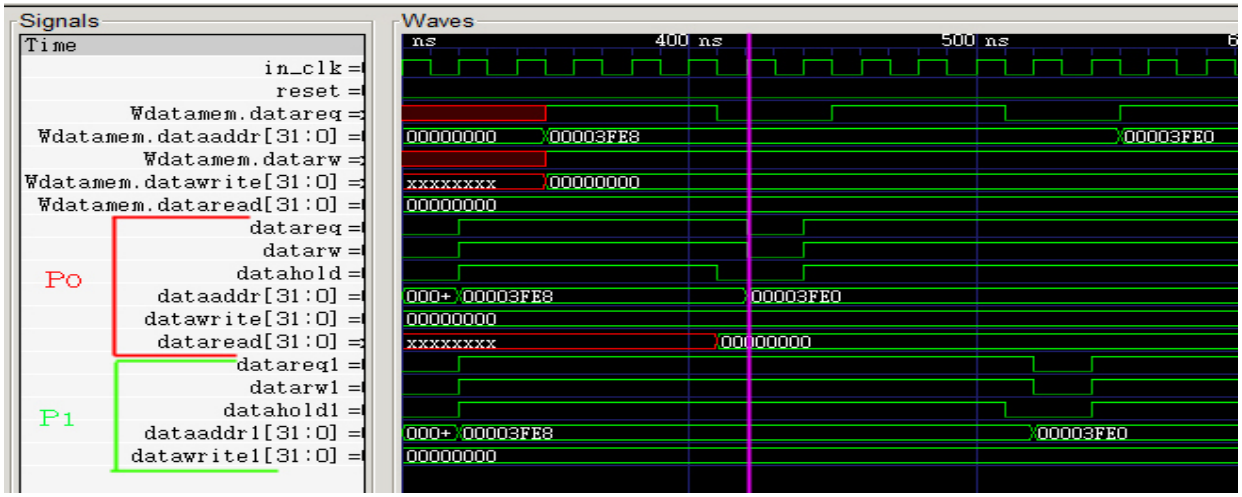

Figura 4.8: Simulação da arquitetura (d)

Todas as requisições ao barramento são tratadas da maneira exemplificada anteriormente, assim garantindo a restrição de acesso e evitando colisões.

Capítulo 5

Conclusões e Trabalhos Futuros

O objetivo deste trabalho é a criação de uma arquitetura multiprocessada com memória de dados compartilhada. Na criação desta arquitetura multiprocessada foi adicionado o barramento de controle de acesso a memória de dados, possibilitando a interconexão dos processadores. Foram tomados cuidados em relação ao desenvolvimento do barramento, criando vetores das entradas, assim facilitando a inserção de um novo conjunto de entradas para outros processadores em trabalhos futuros.

Neste processo do desenvolvimento do multiprocessador foi possível avaliar a dificuldade envolvida no processo de desenvolvimento de modelos multiprocessados, dentre as principais dificuldades encontradas podemos citar:

- Apesar de utilizar o SystemC para facilitar o desenvolvimento do modelo (pois trabalha em um alto nível de abstração), o SardMIPS está em um nível sintetizável, ou seja esta modelado no nível de sinais. Portanto mesmo facilitando a criação do barramento a inclusão correta do mesmo no sistema foi muito complexa, pois foram necessárias várias avaliações sobre o *wave form* do sistema inteiro;

- A dificuldade em modelar *hardware*, pois todos os fluxos de sinais ocorrem em paralelo, além disso, a sincronização correta com os sinais de subida e descida do *clock*;

- Complexidade de modelar arquiteturas multiprocessadas;

Os resultados obtidos mostram que o barramento com dois processadores, teve sucesso, evitando colisões e coordenando o acesso à memória. Alguns módulos e atividades desejáveis

para o sistema não foram desenvolvidas devido à restrições de tempo para a conclusão deste trabalho, podendo ser realizadas em trabalhos futuros. Dentre elas podemos citar:

- Melhorar o barramento de controle ao acesso da memória, principalmente o seu critério de escolha das requisições dos processadores, pois atualmente o barramento prioriza as requisições dos primeiros processadores;
- Desenvolver o mapeamento da memória de dados, com espaços compartilhados e privativos para cada processador;
- Desenvolvimento de um módulo de memória *cache*;
- Desenvolvimento do módulo de controle da coerência das *caches*;
- Criação das arquiteturas com mais processadores;
- Desenvolvimento de testes específicos para plataformas multiprocessadas;
- Comparação dos resultados obtidos com o modelo MMCC;

Apêndice A

Códigos

A.1 Código do barramento

Code A.1: Código da implementação do barramento.

```
1  //
2  // my_bus.cpp BUS CONTROL - WDT -
3  //
4  #include "my_bus.h"
5
6  void my_bus::get_request()
7  {
8          if (reset.read() == false) {
9                  for (int i = 0; i < nproc; i++){
10                         if (datareq[i].read() == 1){
11                                 req[i] = true;
12                                 datahold[i].write(1);
13                         }
14                 }
15         }
16 }
17
18 void my_bus::handle_request()
19 {
20         if (reset.read() == false){
21                 switch(estado) {
22                         case 0:
23                                 for (int i = 0; i < nproc; i++){
24                                         if (req[i] == true) {
25                                                 proc = i;
26                                                 estado = 1;
27                                                 break;
28                                         }
29                                 }
30                                 break;
31                         case 1:
32                                 dataaddr2mem.write(dataaddr[proc].read());
33                                 datarw2mem.write(datarw[proc].read());
34                                 datawrite2mem.write(datawrite[proc].read())
                                       ;
```

```
35                                      databs2mem.write(databs[proc].read());
36                                      datareq2mem.write(SC_LOGIC_1);
37                                      estado = 2;
38                                      break;
39                              case 2:
40                                      estado = 3;
41                                      break;
42
43                              case 3:
44                                      if (memhold.read() == false){
45                                              estado = 4;
46                                      }
47                                      estado = 4;
48                                      break;
49                              case 4:
50                                      dataread[proc].write(dataread2mem.read());
51                                      addrl2proc[proc].write(addrl.read());
52                                      addrs2proc[proc].write(addrs.read());
53                                      page_fault2proc[proc].write(page_fault.read
                                            ());
54                                      req[proc] = false;
55                                      datareq2mem.write(SC_LOGIC_0);
56                                      datahold[proc].write(0);
57                                      estado = 0;
58                                      break;
59
60                      }
61              }
62              else {
63                      estado = 0;
64              }
65      }
```

A.2 Código de teste

Code A.2: Código de teste usado na simulação

```
1  #define memstore(address,save) { \
2  unsigned int *ctrlstore = (unsigned int *) address; \
3  *ctrlstore = save;}
4
5  #define memprint(address,save) { \
6  unsigned int *ctrlstore1 = (unsigned int *) address; \
7  *ctrlstore1 = save;}
8
9
10
11 #define ADDR_IO         0x00009000
12 #define ADDR_STOP       0x7FFFFFFC
13 #define ADDR_QUICK      0x00003F00
14 #define PRINT_IO        0x7FFFFFE0
15 int *print = 0x7FFFFFE0;
16 int a;
17
18 int main(void)
19 {
20
21          int j = 0;
22          asm("nop");
23         a = a + 10;
24          asm("nop");
25          memprint(PRINT_IO,a);
26
27          while (j < 10) j++;
28          asm("nop");
29          memstore(ADDR_STOP,0);
30
31          return 0;
32
33 }
```

Referências Bibliográficas

[1] DOCUMENTATION, A. *Implementing DMA on ARM SMP Systems*. 2009. Disponivel na INTERNET: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0228a/index.html, 2011.

[2] TRENT, R. Cache organization and memory management of the intel nehalem computer architecture. *University of Utah Computer Engineering*, Utah, December 2009.

[3] HENNESSY, J. L. et al. Mips: a vlsi processor architecture. *Technical Report*, Stanford University, November 1981.

[4] MOORE, G. E. Cramming more components onto integrated circuits. *Electronics*, v. 38, n. 8, p. 114–117, April 1965.

[5] HENNESSY, J. L.; PATTERSON, D. *Livro - Organização E Projeto De Computadores: A Interface Hardware-Software*. 2. ed. [S.l.]: Ltc Editora, 2000.

[6] THORNTON, J. E. Ieee annals of the history of computing. In: *The CDC 6600 project*. [S.l.: s.n.], 1980. p. 338–348.

[7] RUSSELL, R. M. The cray-1 computer system. *CACM*, v. 21, n. 1, p. 63–72, 1978.

[8] FRANK, S. J. A tightly coupled multiprocessor system speeds memory-access times. *Electronics*, p. 164–169, January 1984.

[9] SEITZ, C. L. et al. The design of the caltech mosaic c multicomputer. *Proceedings of the 1993 symposium on Research on integrated systems*, MIT Press Cambridge, March 1993.

[10] SEITZ, C. L. The cosmic cube. *Communications of the ACM*, New York, v. 28, n. 1, January 1985.

[11] LENOSKI, D. et al. The directory-based cache coherence protocol for the dash multiprocessor. *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, June 1990.

[12] POPOVICI, K. et al. *Embedded Software Design and Programming of Multiprocessor System-on-Chip*. 1. ed. [S.l.]: Springer, 2010.

[13] TORTATO, J. J. *Projeto e implementação de multiprocessador em dispositivos lógicos programáveis*. Dissertação (Dissertação de Mestrado) — UFPR – Universidade Federal do Paraná, Curitiba - PR, Agosto 2009.

[14] BARBIERO, A. A. *Ambiente de suporte ao projeto de sistemas embarcados*. Dissertação (Dissertação de Mestrado) — UFPR – Universidade Federal do Paraná, Curitiba - PR, Julho 2006.

[15] (OSCI), O. S. I. *SystemC*. 1999. Disponivel na INTERNET: http://www.systemc.org, 2011.

[16] KRASNER, J. Embedded software development issues and challenges. *Embedded Market Forecasters*, July 2003.

[17] SARDMIPS. 2006. Disponivel na INTERNET: http://opencores.org/project,sardmips, 2011.

[18] FLYNN, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, IEEE Computer Society Washington, v. 21, n. 9, September 1972.

[19] HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: A quantitative approach*. 2. ed. San Francisco, California: Morgan Kaufman, 1996.

[20] TANENBAUM, A. S. *Organização Estruturada de Computadores*. 5. ed. [S.l.]: Prentice-Hall, 2006.

[21] GUPTA, R.; SHERIDAN, T.; WHITNEY, D. Experiments using multimodal virtual environments in design for assembly analysis. *Presence: Teleoper*, v. 6, n. 3, p. 318–338, 1997.

[22] LOVETT, T.; CLAPP, R. Sting: a cc-numa computer system for the commercial marketplace. *Proceedings of the 23rd annual international symposium on Computer architecture*, Philadelphia, p. 308–317, May 1996.

[23] STALLINGS, W. *Arquitetura e organização de computadores*. 5. ed. São Paulo: Editora Afiliada, 2003.

[24] BASKETT, F.; JERMOLUK, T. A.; SOLOMON, D. The 4d-mp graphics superworkstation: computing + graphics = 40 mips + 40 mflops and 100,000 lighted polygons per second. *In: Digest of papers, thirty-third IEEE computer society int'l conference*, p. 468–471, 1988.

[25] PATEL, J. H. Retrospective: a low-overhead coherence solution for multiprocessors with private cache memories janak h. patel. *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, ACM New York, p. 206–215, August 1998.

[26] SWEAZEY, P.; SMITH, A. J. A class of compatible cache consistency protocols and their support by the ieee futurebus. *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, IEEE Computer Society Press Los Alamitos, June 1986.

[27] THACKER, C. P.; STEWART, L. C. Firefly: a multiprocessor workstation. *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*, IEEE Computer Society Press Los Alamitos, November 1987.

[28] HANGOUëT et al. *MiniMIPS Project*. 2004. Disponivel na INTERNET: http://opencores.org/project,minimips, 2011.