Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Engenharia – Faculdade de Informática

Curso de Engenharia de Computação

Memória Cache em uma Plataforma Multiprocessada

Volume Final do Trabalho de Conclusão

Autores
Antonio A. de Alecrim Jr.
Rafael Fraga Garibotti

Orientador
Prof. Dr. Fernando Gehm Moraes

Porto Alegre, junho de 2007.

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

Um MPSoC (Multi-processor system-on-chip) consiste de uma arquitetura composta por diversos recursos, incluindo múltiplos processadores, módulos de hardware dedicados, memórias e um meio de interconexão.

Um número crescente de aplicações embarcadas possui requisitos estritos de desempenho, consumo de potência e custo. Exemplos dessas aplicações são encontrados nas áreas de telecomunicações, multimídia, redes e entretenimento. O atendimento de cada um desses requisitos separadamente é uma tarefa difícil e a combinação deles constitui um problema extremamente desafiador. Contudo, MPSoCs possibilitam a adaptação da arquitetura do sistema de forma a melhor atender os requisitos de uma determinada aplicação: a atribuição de poder computacional onde é necessário ao se personalizar o chip, removendo elementos desnecessários (e.g. UART, IDE, SATA) a uma determinada aplicação reduzindo o custo do chip e o consumo de potência devido a redução dos elementos. Isso mostra que MPSoCs não são apenas chips multiprocessadores, os quais apenas utilizam da vantagem da alta densidade de transistores para colocar mais processadores em um único chip, e não tentam atender as necessidades da aplicação. Ao contrário, MPSoCs são arquiteturas personalizadas que fazem um balanço entre as restrições da tecnologia VLSI com os requisitos da aplicação.

Diferentes meios de interconexão são utilizados em MPSoCs, tais como: conexão ponto a ponto, barramento único, barramento hierárquico e redes intra-chip. O barramento é muito utilizado, contudo, não é escalável suportando poucas dezenas de núcleos. A mais nova forma de interconexão são as redes intra-chips, também denominadas NoCs (do inglês, Network on Chip) [MOR04]. Essas redes vêm sendo pesquisadas com o intuito de resolver problemas relacionados à comunicação de dados entre componentes de sistema. Dentre esses problemas, encontram-se a baixa escalabilidade, e a ausência de paralelismo, uma vez que a interconexão, através de um barramento compartilhado, permite que apenas uma comunicação possa ser estabelecida em um dado momento. Tais sistemas multiprocessados contêm diversos processadores com memórias locais, os quais podem acessar uma memória compartilhada através de um subsistema de comunicação. A memória compartilhada, que normalmente contém múltiplos módulos de memória, permite compartilhamento de código e dados entre processadores, a baixo custo e de maneira eficiente.

Memória compartilhada em MPSoCs possibilita um modelo de programação simples (que permite fácil desenvolvimento de software paralelo). Entretanto, devido a competição pelo uso dos recursos compartilhados, a latência média de acesso à memória compartilhada tende a ser maior, o que degrada o desempenho. A maneira usual de reduzir o acesso à memória comum é a utilização de **Memórias Cache**. Existem dois tipos de memória cache em sistemas multiprocessados: Caches Compartilhadas e Caches Privadas.

Apesar da utilização de caches compartilhadas reduzir a média de tempo de acesso a

memória, o seu uso é limitado, pois elas não diminuem a disputa por recursos compartilhados. A introdução de caches privadas é a maneira mais correta de lidar com esse problema. Uma cache privada é capaz de satisfazer a maioria das referências à memória, reduzindo a necessidade de acesso à memória compartilhada.

No entanto, a existência de caches privadas introduz um novo problema conhecido como **Problema de Coerência de Cache**. Múltiplas cópias do mesmo dado podem simultaneamente existir em diferentes caches e, se os processadores estão habilitados a atualizar suas próprias cópias, um cenário de memória inconsistente começa a aparecer, levando ao mau funcionamento do programa. As principais razões para a ocorrência de inconsistência de dados incluem o compartilhamento de dados e os processos de migração. Um sistema é dito ser coerente se todas as leituras, por qualquer processador, retornam o valor produzido pela última operação de escrita, sem importar qual processador realizou esta operação [STA03]. Para manter esta coerência, o sistema deve incorporar algum método que defina o completo e consistente conjunto de operações de acesso à memória compartilhada, prevenindo a inconsistência entre caches, e garantindo a execução do programa com dados corretos.

1.1 Motivação

A necessidade de assegurar o funcionamento de sistemas de computação tem exigido esforços crescentes no sentido de evitar ocorrências de inconsistência da memória, ou seja, o uso de um dado não atualizado.

A motivação do presente trabalho reside na importância das arquiteturas MPSoCs para os atuais sistemas embarcados, e na necessidade de desenvolver arquiteturas escaláveis. Assim, a implementação de um sistema com estas características permitirá mais opções aos desenvolvedores, podendo, estes, escolher entre usar trocas de mensagens entre tarefas, ou a utilização de uma memória compartilhada para tal finalidade, dispondo também um mecanismo coerente de memória cache.

Além de maior liberdade de programação, o desenvolvimento de arquiteturas paralelas, usando memórias caches, abre novas perspectivas relacionadas ao desempenho.

1.2 Objetivos

Este trabalho de conclusão estende o trabalho de mestrado de Cristiane Raquel Woszezenki [WOZ06], o qual propôs a criação de uma plataforma MPSoC no nível físico (VHDL nível RTL), com uma infra-estrutura de hardware e uma infra-estrutura de software capaz de gerenciar a execução de tarefas no sistema. O presente trabalho acrescentará novos módulos à plataforma, inserindo, primeiramente, uma memória compartilhada. Em seguida acrestentam-se caches, proporcionando um melhor desempenho do conjunto.

O objetivo deste trabalho é aplicar os conhecimentos adquiridos durante o curso, desenvolvendo uma estrutura multi-processada com suporte a caches.

2 Referencial Teórico

Uma das técnicas mais importantes que se consolidaram ao longo do tempo em sistemas de memória é a utilização de memórias cache, pois permite acesso mais rápido a informações armazenadas em um nível mais lento de uma hierarquia de memória, nível que geralmente possui maior capacidade de armazenamento em relação à memória cache.

Como a cache é um nível de memória intermediário mais rápido, porém limitado em tamanho (devido ao alto custo tecnológico), é utilizado o princípio da localidade.

O princípio da localidade consiste no fato de que as referências à memória feitas por um processo não são aleatórias. É muito mais provável que itens de memória que serão acessados, dentro de um período razoável de tempo, sejam vizinhos dos itens acessados recentemente, chamada de localidade espacial, e que esses itens previamente acessados sejam acessados novamente, chamada de localidade temporal, podendo manter nesse nível intermediário, cópias das vizinhanças das informações acessadas no nível mais lento.

Devido ao princípio da localidade, podemos trabalhar com estas cópias das vizinhanças durante um bom tempo, sem ter que acessar o sistema mais lento. Isto leva à situação em que o programa enxerga o espaço de endereçamento do sistema grande, com tempos de acesso do sistema rápido.

Para que o sistema funcione, é necessário que o conjunto das vizinhanças acessadas se insira no sistema intermediário por um tempo satisfatório. Desta forma, o processador busca os dados requisitados na cache e, se o dado referenciado encontra-se lá, diz-se que ocorreu um cache hit e o acesso é completado em poucos ciclos de relógio, comparado a um cache miss. Mas, se o dado não está presente na cache, diz-se que ocorreu um cache miss.

Quando acontece um cache miss, um bloco (unidade que possui várias palavras de informação), contendo o dado referenciado e os dados armazenados em sua vizinhança, é copiado da memória principal para a memória cache. Após ou durante (depende da implementação) a transferência deste bloco de dados para a memória cache, procedimento que consome vários ciclos de relógio para ser completada, a informação que gerou o miss é direcionada ao processador.

2.1 Mapeamentos de Memória Cache

As memórias *cache* podem ser estruturadas de diferentes formas. A memória *cache* é formada por uma memória RAM e pelo seu **diretório** (armazena informações globais em relação ao conteúdo das caches).

A memória RAM é organizada em **blocos** de dados, provenientes da memória principal. As caches podem ser **mapeadas diretamente**, **ter mapeamento totalmente associativo** ou **mapeamento associativo por conjunto**.

A organização mais simples é a da *cache* **mapeada diretamente**, Figura 1. Neste tipo de

cache temos acesso simultâneo ao *tag* (nome dado a um campo do diretório utilizado para endereçar os blocos) e aos dados. A taxa de acerto é baixa se dois (ou mais) blocos mapeados no mesmo slot são utilizados alternadamente, reduzindo-se assim o desempenho.

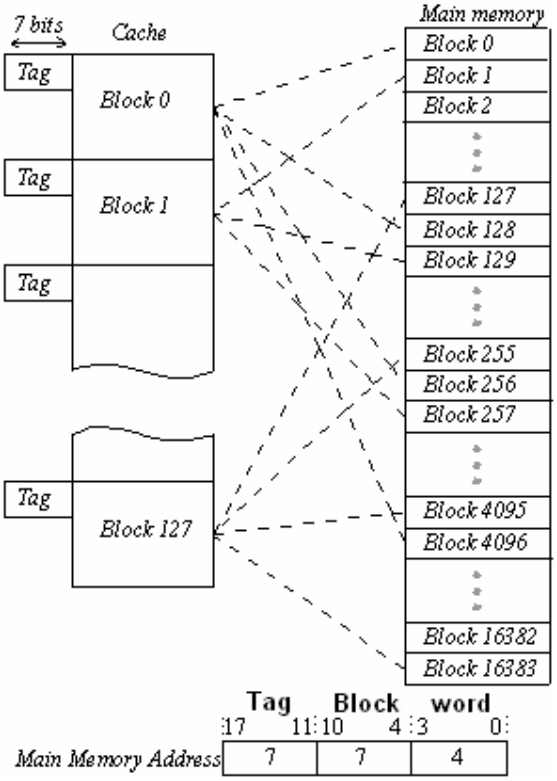

Figura 1 – Exemplo de Memória cache com mapeamento direto.

Na cache mapeada diretamente, uma determinada palavra da memória principal só pode ocupar uma única posição na cache, havendo ainda uma tag separada para cada bloco.

No exemplo, mostrado na Figura 1, cada bloco tem 16 words (4 bits), o "Main Memory Address" mostra a configuração para endereçamento à memória principal, sendo que os bits de 17 a 11 são usados para comparação (*cache hit*, *cache miss*) com o tag, os bits de 10 a 4 representam o índice do bloco e os bits de 3 a 0 representam o endereçamento da palavra no bloco.

A tag contém informações sobre um endereço de memória, que permite identificar se a informação desejada está ou não na cache. Ela só precisa conter a parte superior do endereço, correspondente aos bits que não estão sendo usados, como índice da cache.

Os índices da memória cache são usados para selecionar a única entrada da cache que corresponde ao endereço fornecido. Além disso, existe um bit de validade associado a cada tag, para indicar se uma entrada contém ou não um endereço válido.

As *caches* com **mapeamento totalmente associativo** têm alto desempenho e alto custo de hardware, porém a flexibilidade de mapeamento torna necessário implementar um algoritmo de substituição. A substituição de um bloco só é necessária quando a cache está cheia.

Na cache com mapeamento totalmente associativo, uma determinada palavra pode ocupar qualquer posição na cache. Como visto no exemplo mostrado na Figura 2, cada bloco tem 16 words

(4 bits). Os bits de tag [17..4] são usados para comparação. Neste caso, para saber se um bloco está na cache, é necessário pesquisar todos os blocos. Se o bloco referenciado não for encontrado (ou seja, ocorreu um *cache miss*), a lógica de controle da memória cache se encarrega de copiar o bloco apropriado dos níveis inferiores da hierarquia de memória em um bloco livre. Mas se a cache estiver cheia, é necessária a substituição de um bloco.

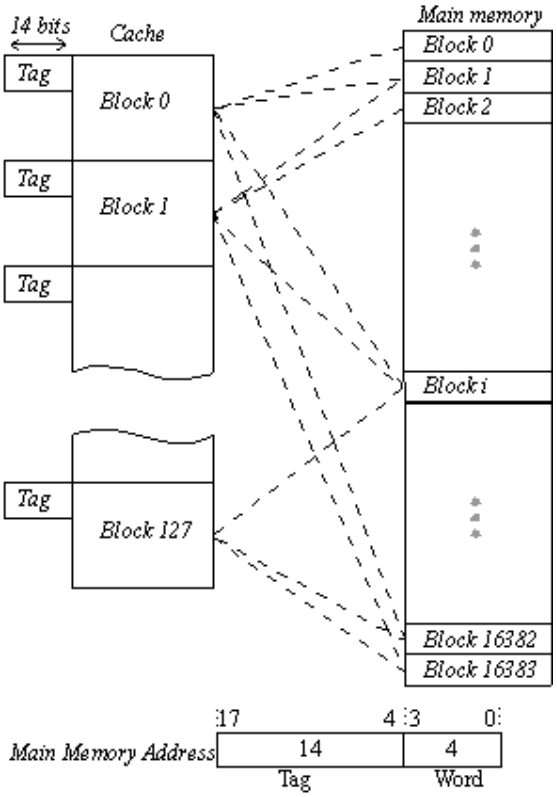

Figura 2 – Memória cache com mapeamento totalmente associativo.

A questão central na substituição é a escolha do bloco a ser descartado. Uma forma simples seria escolher aleatoriamente qualquer um dos blocos do conjunto selecionado. A escolha aleatória é atrativa devido à sua simplicidade de implementação, porém, esta política aumenta a chance de ser escolhido um bloco que será referenciado em breve. Assim, com a escolha aleatória existe o risco de um aumento excessivo no número de *cache miss*.

Em geral, é adotada uma política de escolha mais segura, tal como LFU (*Least Frequently Used*) ou LRU (*Least Recently Used*). A política LFU determina que o bloco a ser removido seja o que possui uma menor freqüência de acesso, já o algoritmo LRU diz que o bloco a ser substituído é aquele que não é referenciado há mais tempo. Estes critérios de escolha baseiam-se no princípio da localidade temporal.

Caches com **mapeamento associativo por conjunto**, Figura 3, tem compromisso entre mapeamento direto e totalmente associativo. Este mapeamento reduz o tamanho e o custo da comparação associativa, e é usado pela maioria das CPUs atuais [GOO83]. É importante ressaltar que o tamanho do bloco, o número de blocos por conjunto (Slot), e o número de conjuntos, podem variar, resultando em configurações diferentes.

A memória cache associativa por conjunto, ao receber o endereço de uma locação de memória, interpreta o endereço da seguinte forma. O endereço é, logicamente, dividido em três campos: *word*, formado pelos bits menos significativos; *conjunto* (set), formado pelos *n* bits seguintes; e *tag*, composto pelos *m* bits mais significativos do endereço. A memória cache usa os bits do campo *word* para selecionar uma informação específica dentro de um bloco. O campo *set* é usado para selecionar um dos conjuntos, enquanto o campo *tag* é utilizado para verificar se o dado referenciado se encontra em algum dos blocos do conjunto selecionado.

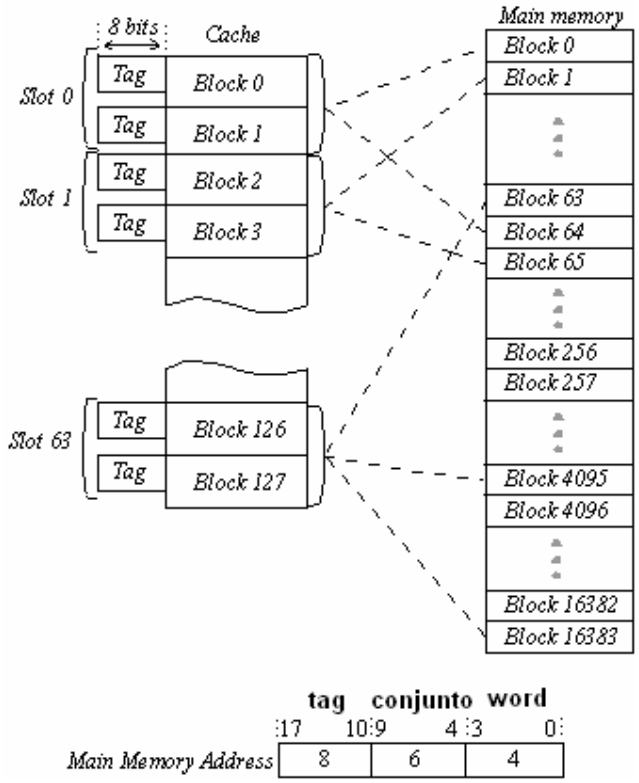

Figura 3 – Memória cache com mapeamento associativo por conjunto.

Em um acesso, a memória cache seleciona o conjunto e compara o tag do endereço recebido com os *tags* armazenados nas entradas de diretório do conjunto selecionado. O bloco pode ser colocado em qualquer elemento deste conjunto. Se o tag no endereço coincide com algum dos *tags* no diretório, isto significa que o bloco com a informação referenciada encontra-se no bloco associado à entrada do diretório que contém o tag coincidente. Este bloco é, então, selecionado, e o a informação dentro deste bloco é acessada. Isto pode ser feito concorrentemente em um único ciclo de relógio.

2.2 Política de Atualização da Memória Cache

Para manter coerência entre cache e memória principal, políticas de escrita, como *write-through* ou *write-back,* devem ser utilizadas para atualizações. A política *write-through* escreve os dados da cache na memória principal toda vez que os dados da cache forem atualizados, possibilitando a consistência dos dados.

A política *write-through* tem que escrever a palavra na memória principal a cada

atualização, pois os dados deverão sempre ser atualizados na memória principal. Já a política de *write-back* escreve unicamente na cache, somente quando o bloco for removido da cache ele é inteiramente atualizado na memória principal, possuindo como desvantagem gerar um alto tráfego entre memória principal e cache.

Dependendo da política de escrita adotada, caso o item a ser escrito não esteja presente na cache, ele pode ser trazido para a cache para ser atualizado ou é atualizado diretamente na memória principal sem trazê-lo para a memória cache, política de atualização que vai além das duas mencionadas anteriormente.

2.3 Memória Virtual

Este assunto é abordado para explicar técnicas de endereçamento de memória, como que funciona a tradução dos endereços gerados por um processador para o endereço físico de memória. Apresentam-se estes conceitos por razões de completude, i.e., situar o leitor nos temas referentes à gerência de memória.

A memória principal disponível em um computador é, em geral, bem menor do que o espaço de endereçamento de memória permitido pelo processador. O esquema de memória virtual foi criado para permitir a execução de programas cujas exigências, quanto ao tamanho da memória, sejam maiores do que a capacidade de memória instalada no sistema.

Em um sistema sem memória virtual, o endereço gerado pelo programa em execução é o próprio endereço usado para acessar a memória principal. O mesmo não acontece em um sistema com memória virtual, onde o endereço gerado pelo programa, ou **endereço virtual**, é diferente do **endereço real,** usado para acessar a memória principal. Os possíveis endereços virtuais que podem ser gerados pelo programa formam o **espaço de endereçamento virtual**, enquanto os endereços na memória principal constituem o **espaço de endereçamento real**. Sob o ponto de vista de um programa, a memória disponível é aquela representada pelo espaço de endereçamento virtual. O espaço de endereçamento virtual visto e utilizado pelo programa pode ser bem maior do que o espaço de endereçamento real, retirando do programa as limitações impostas pela capacidade da memória física de fato existente no sistema.

É importante perceber que o espaço de endereçamento virtual é uma técnica de endereçamento em que sob o ponto de vista do programa, as instruções e dados estejam armazenados dentro do espaço de endereçamento, na realidade eles podem estar na memória principal ou virtual. Esta distinção, entre endereços e espaços de endereçamento, exige um mecanismo que faça a correspondência entre o endereço virtual, gerado pelo programa, e o endereço real, que é usado para acessar a memória principal.

A técnica de memória virtual permite que as instruções e os dados do programa que se encontram no espaço virtual, não estejam necessariamente presentes na memória principal, no momento em que são referenciados. Assim, além do mapeamento anteriormente mencionado, é necessário um mecanismo para o carregamento automático na memória principal das instruções e dados, que são referenciados pelo programa dentro da sua memória virtual, e que não se encontram

presentes na memória física.

O mapeamento entre endereços virtuais e reais é realizado por um componente do sub-sistema de memória, denominado **tradutor de endereços dinâmicos**, ou DAT (*Dynamic Address Translator*). Para realizar este trabalho, o DAT utiliza uma **tabela de mapeamento,** localizada na memória principal. Esta tabela de mapeamento permanece na memória principal, durante a execução do programa. Ao receber um endereço virtual, o DAT converte este endereço em endereço real ao consultar sua tabela de mapeamento.

Este tipo de mapeamento não é feito no nível de cada locação de memória, pois isto exigiria uma tabela de mapeamento com um número de entradas igual ao tamanho do espaço de endereçamento virtual. Para manter um tamanho de tabela aceitável, o mapeamento é feito no nível de blocos. Cada entrada na tabela de mapeamento contém o endereço-base de um bloco, ou seja, o endereço real, a partir do qual o bloco está armazenado na memória principal. Em uma **memória virtual paginada**, os blocos são chamados **páginas** e possuem **tamanho fixo**. Em uma **memória virtual segmentada**, os blocos são chamados **segmentos**, e podem ter **tamanhos variáveis**.

Na paginação, Figura 4, o espaço de endereçamento lógico de um processo pode ser *não contínuo*, alocando-se blocos para a memória física, sempre que existir espaço disponível. A paginação divide a memória física em partes de tamanho fixo, chamadas de *blocos* (*frames*), e divide a memória lógica em partes do mesmo tamanho, chamadas de *páginas*. A paginação mantém controle de todos os *blocos* livres e, para executar um programa com n *páginas*, necessitam encontrar n *blocos* livres para carregar o programa.

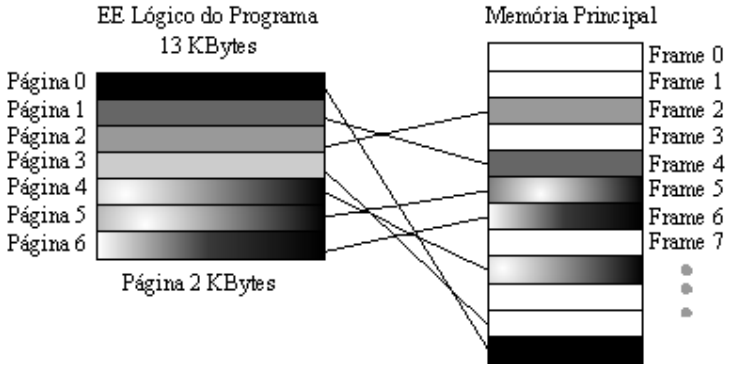

Figura 4 – Memória paginada – Espaço de endereçamento (EE) lógico do programa e memória principal.

Define-se uma Tabela de Páginas (*page table*) para traduzir o endereço lógico em físico, e sua fragmentação é interna. A Figura 5 mostra um esquema de tradução de endereço. O endereço gerado pela CPU é divido em:

- *Número da Página* (p) – usada como um índice em uma tabela de páginas que contém o endereço base de cada página na memória física;
- *Offset da Página* (f) – usado como deslocamento, endereço base.
- *Posição na Página* (d) – combinado com o *endereço base* (f) para definir o endereço de

memória que é enviado à unidade de memória.

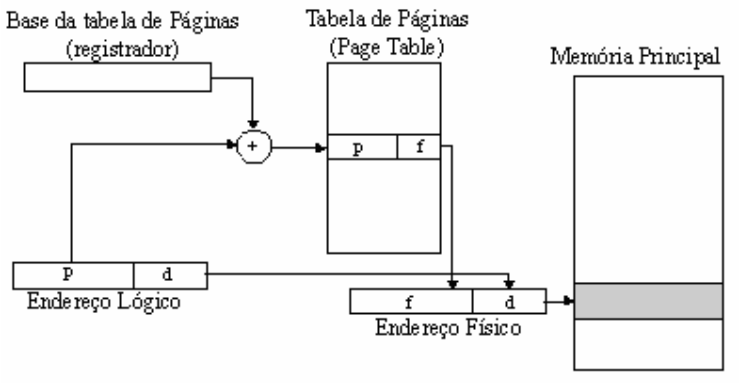

Figura 5 – Conversão de endereços na paginação.

A Tabela de Páginas contém uma proteção de memória que é implementada através de bits de proteção, associados a cada bloco. Se o bit está "válido", indica que a página associada está no espaço de endereçamento lógico do processo e, portanto, o acesso é legal, mas se está "inválido", isto indica que a página não está no espaço de endereçamento lógico do processo.

Na segmentação, Figura 6, um programa é uma coleção de segmentos, e um segmento é uma unidade lógica, como, por exemplo, programa principal, função, variáveis locais, variáveis globais, etc...

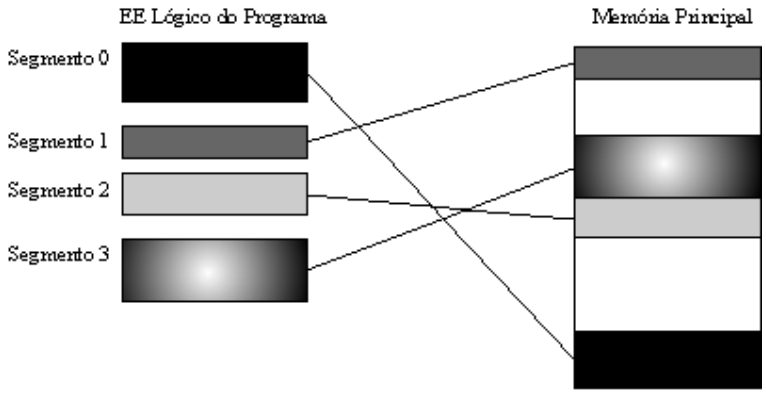

Figura 6 – Memória segmentada – Espaço de endereçamento (EE) lógico do programa e memória principal.

A Tabela de Segmentos (segment table) mapeia endereços físicos bidimensionais. Cada entrada na tabela possui uma base que contem o endereço físico inicial, no qual o segmento reside na memória e, o limite, onde é especificado o tamanho do segmento. A Figura 7 mostra um esquema de tradução de endereço. O endereço gerado pela CPU é divido em:

- *Número do Segmento* (s) – usada como um índice em uma tabela de segmentos que encontram a base de um segmento na memória e verificam se o deslocamento requisitado se encontra dentro do limite máximo do segmento;
- *Posição no Segmento* (d) – usado para definir o endereço de memória que é enviado a unidade de memória.

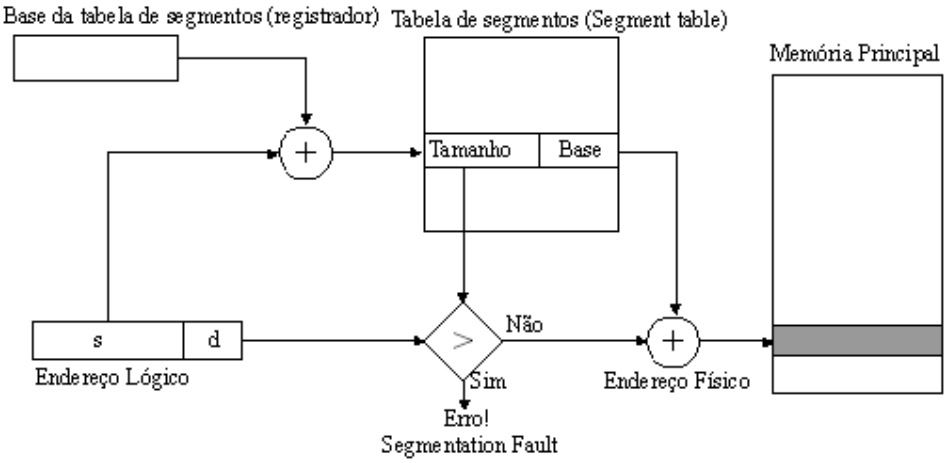

Figura 7 – Conversão de endereços na segmentação.

A realocação é dinâmica e por tabela de segmento. Contém segmentos compartilhados com o mesmo número de segmento. Possui fragmentação externa. Ela tem proteção, e com cada entrada na tabela de segmento é associado um bit de validação, e tem privilégios de leitura/escrita/execução. Os bits de proteção são associados com segmentos, isto é, o compartilhamento de código ocorre em nível de segmento. Uma vez que segmentos variam em tamanho, a alocação de memória é um problema dinâmico.

3 Protocolos de Coerência de Cache

Uma plataforma multiprocessada que dispõe de uma memória compartilhada e a possibilidade de um ou mais processadores possuírem cópia do mesmo dado em uma memória local, memória cache, por exemplo, gera um problema de coerência de dados. Este problema, definido como problema de coerência de cache, torna necessário um controle sobre a forma que os dados são distribuídos ou acessados.

As soluções para o problema de coerência de cache podem ser divididas em duas categorias: *software* e *hardware*.

As soluções por software geralmente se baseiam em uma análise dos códigos dos programas pelo compilador. O compilador separa os itens de dados em dois grupos: *cachable* (que podem ser armazenados em cache), e *non-cachable* (não podem ser armazenados em cache). O sistema operacional tenta evitar que os dados *non-cachable* sejam copiados para a cache, mantendo-os sempre na memória principal. Isso pode levar a uma utilização ineficiente da cache, visto que alguns dados compartilhados podem ser de uso exclusivo de um processador, em algum intervalo de tempo.

Outras técnicas permitem uma análise mais detalhada dos códigos, para determinar períodos livres para variáveis compartilhadas, nelas o compilador insere instruções no código gerado para forçar a coerência de cache em períodos críticos [GOO89].

As abordagens em software tentam evitar a necessidade de um *hardware* adicional para tratar do problema da coerência, deixando para o compilador ou o sistema operacional tratar o problema, mas, por serem efetuadas em tempo de compilação, tendem a tomar decisões que não aproveitam a cache da melhor maneira possível [STA03].

As soluções em hardware são as mais utilizadas em sistemas multiprocessadores comerciais, e são conhecidas como **Protocolos de Coerência de Cache.** Estes protocolos devem ser capazes de detectar acessos incoerentes à memória e garantir a coerência dos blocos da cache (invalidando ou atualizando o bloco), em tempo de execução [LIL93].

Os protocolos de coerência de cache se diferem, principalmente, na maneira em que eles distribuem informações a respeito de escritas da memória em outros processadores, no sistema. Segundo essas características, os esquemas de coerência de cache por hardware podem ser divididos em dois grupos: ***Protocolos de Diretório*** e ***Protocolos Snoopy***.

Os Protocolos de Diretório mantém as informações de cópias e compartilhamento dos blocos da cache de todos os processadores centralizado em um local, chamado diretório.

Protocolos Snoopy são implementados nos controladores de cada cache. Neste protocolo os controladores ficam monitorando o barramento para saber se eles possuem alguma cópia do bloco solicitado. Assim, as informações de compartilhamento do bloco são mantidas em cada cache.

Neste Capítulo, apresentam-se os conceitos básicos de alguns protocolos de coerência de

cache.

3.1 Protocolos de Diretório

Estes protocolos armazenam informações a respeito de onde as cópias dos blocos estão localizadas. A responsabilidade para manutenção da coerência neste grupo de protocolos é predominantemente delegada a um controlador central, que é normalmente uma parte do controlador da memória principal.

Um **diretório**, geralmente localizado na memória principal, armazena informações do estado global com relação ao conteúdo das várias caches locais. Quando ocorre uma requisição dos controladores da cache local, o controlador central analisa o diretório e fornece os comandos necessários para transferir dados entre memória e caches, ou entre caches. Ele é também responsável por manter o estado da informação atualizado, e cada ação local que possa afetar o estado global de um bloco deve ser relatada ao controlador central.

Os esquemas baseados em diretório se apresentam, basicamente, em três categorias possíveis: diretórios com mapeamento completo, diretórios limitados e diretórios encadeados. Nas soluções clássicas (diretórios com mapeamento completo), cada entrada de diretório tem um bit por processador e mais dirty. Um exemplo deste esquema de diretório pode ser encontrado em [GOO89]. Em diretórios limitados, existe um número fixo de bits para indicar cópias em processadores, restringindo o número de cópias simultâneas de qualquer bloco. Em diretórios encadeados, um mapeamento completo é emulado, distribuindo o diretório entre as caches [CHA90].

O diretório pode ser centralizado, juntamente com uma memória compartilhada centralizada, ou também pode ser distribuído, assim como a memória. Neste caso, o diretório não se torna um gargalo, permitindo que acessos a diferentes entradas de diretórios possam ser realizados de forma distribuída [HEN03]. Esses diretórios distribuídos garantem escalabilidade. Entradas de diretório são organizadas na forma de listas simples, ou duplamente ligadas, onde todas as caches que compartilham o mesmo bloco são acrescentadas a uma lista. Comandos do controlador da cache, fornecidos ao início da lista, percorrem a mesma por meio de ponteiros.

Em [HEN03], um exemplo simples de protocolo de diretório armazenado de forma distribuída é apresentado, o qual implementa as operações: falha de leitura da cache e gravação de um bloco da cache. Para implementar este protocolo, Figura 8, são mantidos três estados para cada bloco da cache:

- *Compartilhado* - o bloco na memória está atualizado e um ou mais processadores possuem em sua cache tal bloco;
- *Não inserido na cache* - nenhum processador tem uma cópia do bloco da cache;
- *Exclusivo* - o bloco na memória está desatualizado e somente um processador tem a cópia do bloco na cache (proprietário).

Quando o bloco está no estado compartilhado e ocorrer uma falha de leitura, o processador solicitante recebe os dados da memória e é adicionado ao conjunto de compartilhadores do bloco. Se houver uma falha de escrita, todos os processadores recebem mensagens para invalidarem seus blocos de cache, e o estado do bloco se torna exclusivo, indicando no conjunto de compartilhadores que o único processador a possuir uma cópia foi o que realizou a solicitação.

Quando um bloco está no estado não inserido na cache e ocorrer uma falha de leitura, o estado do bloco se torna compartilhado, e a cache local recebe os dados da memória. Caso ocorra uma falha de escrita, o processador solicitante se torna proprietário, definindo o estado do bloco como exclusivo. O conjunto de compartilhadores do bloco reflete esse estado, mantendo indicação de cópia de bloco somente para esse processador.

Quando o bloco está no estado exclusivo e ocorrer uma falha de leitura, o processador proprietário recebe uma mensagem de busca de dados, mudando o estado do bloco para compartilhado, e fornecendo ao solicitante o bloco desejado. É adicionado ao conjunto de compartilhadores o processador solicitante.

Ao substituir um bloco, o processador proprietário deverá realizar write-back de dados, atualizando a cópia em memória, e removendo o processador do conjunto de compartilhadores de tal bloco. Em falhas de escrita, uma mensagem é enviada ao antigo proprietário, invalidando seu bloco na cache. Em seguida, o conjunto de compartilhadores mantém somente o processador solicitante, mantendo o estado do bloco como exclusivo para ele.

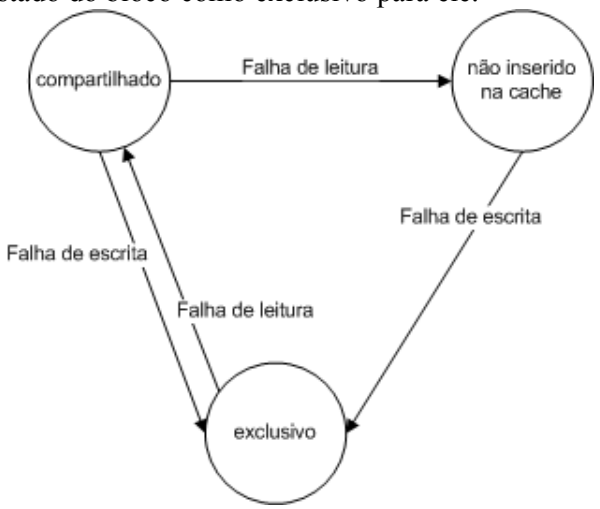

Figura 8 – Protocolo diretório.

Os *directory protocols* apresentam as desvantagens causadas pelo uso de um ponto central e pelo *overhead* de comunicação entre os vários controladores de cache e controlador central. Entretanto, esses sistemas são mais escaláveis, suportando um número maior de processadores, e envolvendo barramentos múltiplos ou redes de interconexão.

3.2 Protocolos Snoopy

Diferente do caso anterior, que é caracterizado por um método centralizado, os *snoopy protocols* utilizam distribuição para tratar coerência de cache. Eles são baseados nas ações de

controladores de cache local, e suas informações de estado local sobre o dado na cache. Todas as ações com o bloco compartilhado atual devem ser anunciadas a todas as outras caches através de *broadcast*. Controladores de cache local são capazes de "espionar" a rede, e são capazes de reconhecer as ações e condições que determinem alguma reação, de acordo com o protocolo utilizado para preservar a coerência.

Snoopy protocols são ideais para multiprocessadores baseados em barramento, pelo fato de que o barramento compartilhado favorece o *broadcast*. Entretanto, as transmissões de mensagens para coerência no barramento compartilhado causam um tráfego adicional. Conseqüentemente, apenas um sistema com um número médio ou pequeno de multiprocessadores pode ser suportado com *snoopy protocols*.

Existem dois protocolos de monitoração básicos, chamados ***write-invalidate*** e ***write-update*** (ou *write-broadcast*). No protocolo *Write-invalidate* (Invalidação), antes da escrita em um bloco, uma mensagem é enviada pelo barramento com o objetivo de invalidar todas as cópias presentes nos demais processadores. Assim, esse processador pode efetuar operações de escrita neste bloco da cache enquanto outro processador não requisitar este bloco, dando a idéia de leitores-escritor, ou seja, pode haver vários leitores para um bloco da cache, mas apenas um escritor. No caso do protocolo *Write-update* (Atualização), ao ser realizada uma escrita em um bloco da cache por um processador, ela é atualizada em todas as cópias presentes nas caches dos outros processadores através do envio do bloco alterado pelo barramento.

Em geral, o mais eficiente desses protocolos depende do padrão de leituras e escritas da memória [STA03]. A abordagem de invalidação é a mais usada nos sistemas multiprocessados atuais, pois sugere menor tráfego no barramento. Mas nenhum dos dois métodos é capaz de melhorar a performance para todos os tipos de cargas de trabalho. Por esse motivo, alguns protocolos propostos combinam as duas políticas. Eles iniciam com *broadcast* de escritas, mas quando uma longa seqüência de escritas locais é encontrada ou prevista, o sinal de invalidação do bloco é enviado. Estas soluções procuram adaptar os esquemas de coerência de maneira a melhorar o desempenho.

3.2.1 Write-Once

Historicamente é o primeiro protocolo de invalidação que foi proposto na literatura [GOO83] (Figura 8). Nesse esquema, os blocos em uma cache podem estar em um de quatro estados. *Invalid:* informação não presente na cache. *Valid:* informação presente na cache e possivelmente em outras caches, todas cópias idênticas à memória. Reserved: informação presente somente nesta cache e de forma idêntica à memória. Dirty: informação presente somente nesta cache, mas a cópia da memória está desatualizada. Estes quatro estados estão representados na Figura 9, onde as linhas tracejadas representam as ações no barramento e as linhas contínuas ações internas à cache, esta máquina se inicia ao ocorrer um read miss ou um write miss.

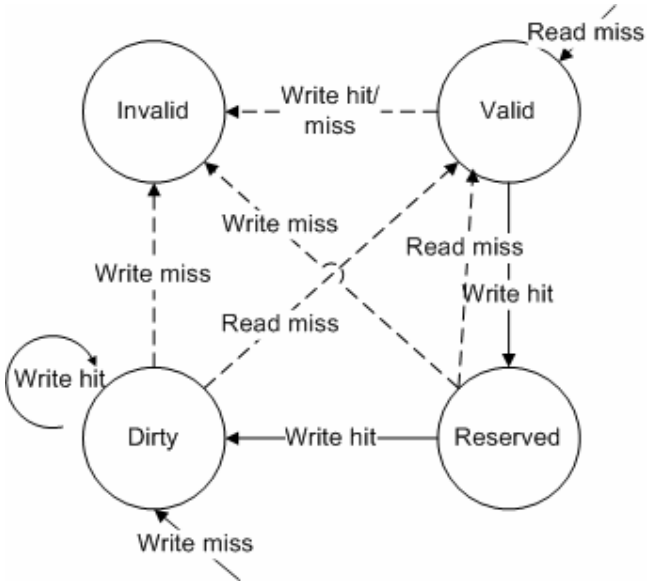

Figura 9 – Protocolo Write-Once.

Em um read miss, se a informação solicitada está em um bloco em outro processador que esteja no estado Dirty, este processador fornece o bloco solicitado, realizando um write-back para a memória. Caso o bloco não esteja no estado Dirty, então o bloco vem da memória principal. Todas as caches com uma cópia do bloco definem seu estado como *Valid*.

Em um write hit, caso o estado do bloco seja Dirty, a escrita é realizada sem alteração de estado. Se o estado for Reserved, então o estado do bloco é mudado para Dirty. Se o estado for *Valid*, o bloco é gravado na memória por write-through e seu estado mudado para Reserved. O estado *Valid* quer dizer que é um dado compartilhado, e esta informação deve ser colocado no barramento para as outras cópias serem invalidadas.

Quando ocorre um write miss, o bloco que não está presente na cache pode estar somente na memória ou replicado em outra cache. Se este bloco estiver presente somente na memória, copia-se o bloco da memória para a cache. Se este bloco estiver replicado em outra cache é necessário verificar se ele está marcado como *Dirty*, se este bloco estiver como *Dirty*, é copiado desta cache para a que recebeu o comando write. Após a cache que gerou o write miss receber o bloco, a escrita é realizada, este bloco tem seu estado alterado para *Dirty* e todas as outras cópias deste bloco, se existirem, vão para o estado *Invalid*.

3.2.2 Berkeley

Foi desenvolvido pela Universidade da Califórnia para ser aplicado em uma máquina RISC multiprocessada, ele usa transferências diretas entre caches [KAT85].

O protocolo Berkeley usa a idéia de linha de cache proprietária. Em qualquer momento, um bloco pode estar de forma exclusiva em apenas uma das caches, ou na memória. Existem quatro estados: *Invalid*, *Valid*, *Shared Dirty* e *Private Dirty*. Quando um bloco é compartilhado, apenas o proprietário contém o bloco com a linha na cache no estado *Shared Dirty*; todos os outros têm a

linha no estado *Valid*. Portanto, uma linha de cache pode ser *Shared Dirty* ou *Private Dirty* em apenas uma cache (a proprietária). Blocos no estado Shared Dirty e Private Dirty, quando escolhidos para substituição na cache, são gravados na memória principal por write-back.

Os estados do protocolo Berkeley estão representados na Figura 10, as linhas tracejadas representam as ações no barramento e as linhas contínuas ações internas à cache, esta máquina se inicia ao ocorrer um read miss ou um write miss.

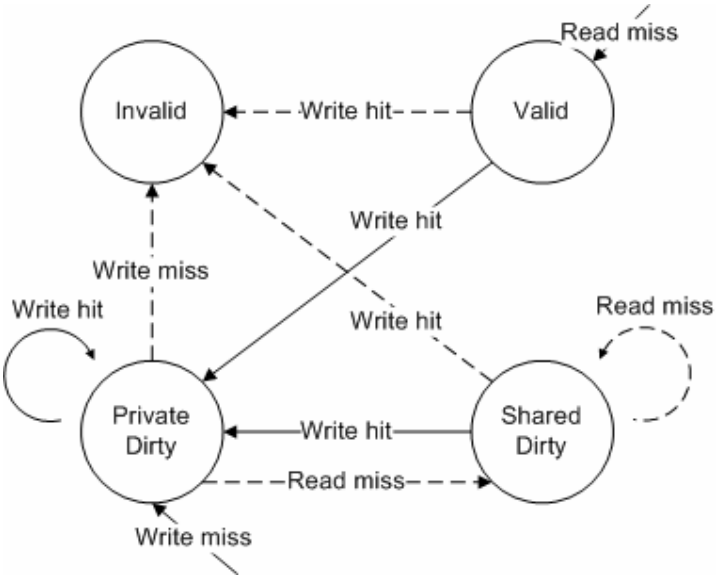

Figura 10 – Protocolo Berkeley.

Em um read *miss*, a cache que tem o bloco nos estados *Shared Dirty* ou *Private Dirty* transfere diretamente o bloco para a cache que fez o pedido, mudando seu estado para Shared Dirty. Se nenhuma cache possui o bloco ou ele está em qualquer outro estado, então o bloco vem da memória principal. O bloco da cache que fez o pedido tem seu estado alterado para *Valid*.

Em um *write hit*, se o estado do bloco é Private Dirty, então o estado não é alterado. Se o estado é *Valid* ou Shared Dirty, o estado é alterado para Private Dirty e um broadcast é feito pelo barramento para que todas as outras caches possam invalidar suas cópias.

Em um *write miss*, o bloco é colocado no estado Private Dirty e se torna a proprietária, e todas as cópias em outras caches são *Invalid*.

3.2.3 Illinois

Este método [PAP84] se baseia no fato de que falhas de acesso a blocos podem ser tratadas procurando por blocos em outras caches, ou na memória principal, e também assume que a cache que requisitou pode determinar a origem do bloco. A melhora de desempenho proposta por esse método é que ele percebe que invalidações para acertos de escrita não são necessárias em blocos privados não modificados, pois é possível determinar, a partir dos estados, se o bloco é ou não compartilhado. Assim como os anteriores, também são usados quatro estados: *Invalid*, *Read Private (Exclusive)*, *Shared* e *Dirty*. Também é utilizada a abordagem de write-back, mas somente quando o bloco está no estado Dirty. Estes quatro estados estão representados na Figura 11, as linhas

tracejadas representam as ações no barramento e as linhas contínuas ações internas à cache, esta máquina se inicia ao ocorrer um read miss ou um write miss.

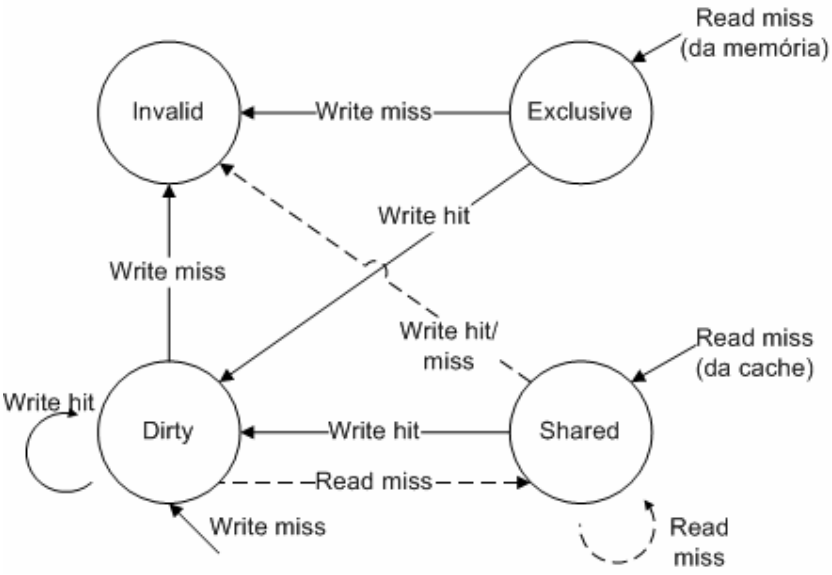

Figura 11 – Protocolo Illinois.

Em um *read miss*, a cache com maior prioridade que tenha uma cópia, vai colocar o bloco no barramento, e gravar o bloco na memória principal, se o estado for Dirty. Todas as caches que possuem uma cópia do bloco mudam seu estado para *Shared*. Se o bloco vier da memória principal, então nenhuma outra cache tem o bloco, e o estado é definido como *Exclusive*.

Em um *write hit*, um bloco Dirty é escrito sem transições, e um bloco *Exclusive* é mudado para Dirty. Se o bloco é *Shared*, então é necessário invalidar todas as cópias existentes, e o bloco gravado também é definido como Dirty.

Em um *write miss*, todas as caches com cópias vão para *Invalid,* e o bloco é carregado como Dirty.

3.2.4 Firefly

É o esquema usado no workstation Firefly [THA84]. Este protocolo apresenta quatro estados, mas utiliza apenas três: *Read Private (Exclusive), Shared,* e *Dirty*.

O estado *Invalid* é utilizado apenas para uma condição inicial para a linha da cache. Por ser um protocolo de atualização, e não de invalidação, como os anteriores, escritas na cache geram *broadcasts* e escritas à memória, ou seja, todas as outras caches compartilhando o bloco "espionam" o barramento e numa escrita atualizam suas cópias, ao invés de invalidá-las. Dessa forma, nenhuma linha da cache será inválida depois de ser carregada.

Existe ainda uma linha de barramento especial chamada *SharedLine,* que é utilizada para indicar que outras caches estão compartilhando o bloco. Estes estados estão representados na Figura 12, onde as linhas tracejadas representam as ações no barramento e as linhas contínuas ações internas à cache, esta máquina se inicia ao ocorrer um read miss ou um write miss.

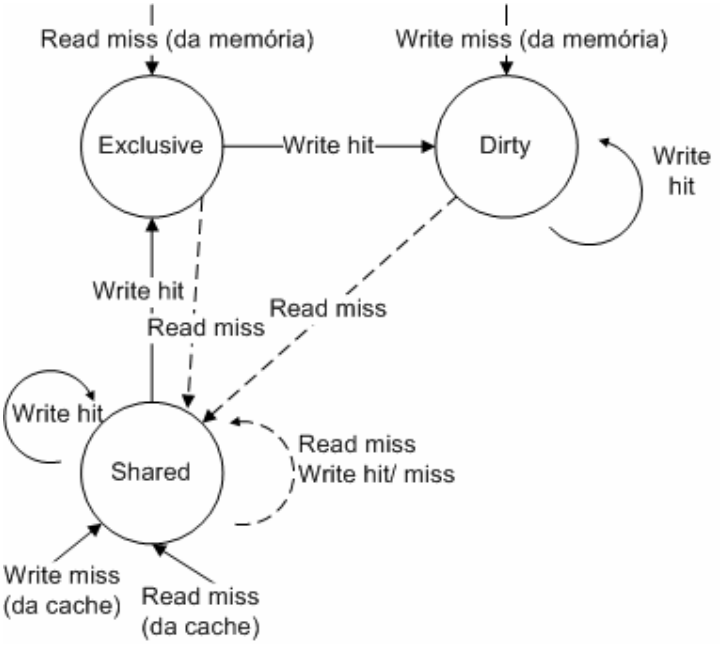

Figura 12 – Protocolo Firefly.

Em um read miss, se qualquer outra cache tem uma cópia do bloco, ela comunica a cache requisitante pela SharedLine e seus estados passam a ser Shared e, caso o bloco seja Dirty, ele é gravado na memória principal. Se nenhuma cache possui o bloco, este vem da memória principal e é definido para ficar no estado Exclusive.

Em um *write hit*, nenhuma alteração de estado é realizada caso o estado seja Dirty. Se o estado for *Exclusive*, então é modificado para Dirty.

Quando o estado for *Shared* é que aparece a principal modificação desse protocolo: como pode haver cópias em outras caches e não há invalidações, então, todas as outras caches compartilhando o dado, "pegam" o dado do barramento e atualizam suas cópias. Além disso, as caches ativam o SharedLine, para que a cache requisitante verifique se ainda existem caches compartilhando o bloco: se essa linha estiver com sinal alto, o compartilhamento existe, e então o estado é mantido em *Shared*. Caso contrário, não há cópias, e o estado pode ser definido como *Exclusive*.

Em um *write miss*, se nenhuma cache possui o bloco, este é carregado da memória no estado Dirty, e depois escrito. Se alguma cache o possui, então o bloco é carregado no estado *Shared,* e a palavra é escrita na memória, sendo que os blocos que possuem uma cópia são atualizados.

3.2.5 MESI

No protocolo MESI [STA03], largamente utilizado em sistemas multiprocessadores comerciais, tais como Pentium e PowerPC, um bloco pode ter quatro estados que são chamados de protocolos MESI, devido aos seus estados: *Modified, Exclusive, Shared e Invalid*. Estes estados estão representados na Figura 13, onde as linhas tracejadas representam as ações no barramento e as

linhas contínuas ações internas à cache, esta máquina se inicia ao ocorrer um read miss ou um write miss.

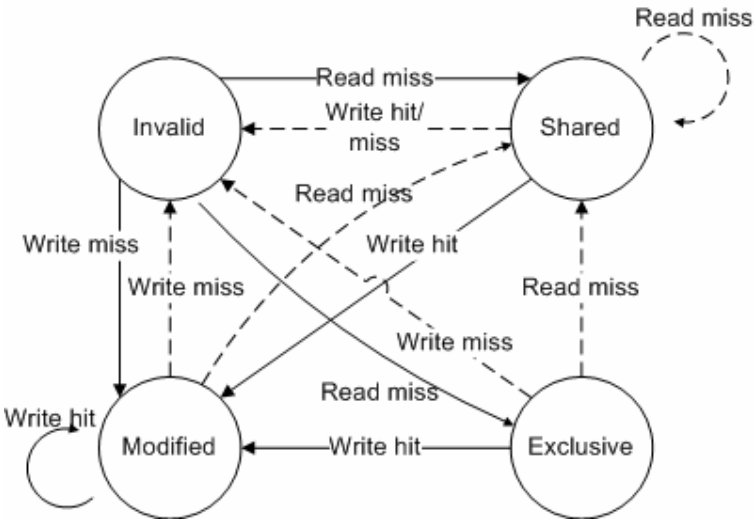

Figura 13 – Protocolo MESI.

Em um *read miss*, se alguma cache tem o bloco no estado *Modified*, então ela fornece o bloco para a cache que o requisitou, e ambos definem seu estado para *Shared*. Caso alguma cache tenha o bloco no estado *Shared* ou *Exclusive*, então a cache que fez o pedido faz a leitura do bloco da memória principal, e todas as caches com cópias mudam seu estado para *Shared*. Se nenhuma cache possui o bloco requisitado, então o processador carrega o bloco da memória principal, e define seu estado como *Exclusive*.

Em um *write hit*, caso o estado do bloco seja *Modified*, nenhuma transição é necessária. Se o estado for *Exclusive*, então é alterado para *Modified*. Caso o estado do bloco seja *Shared*, é necessário invalidar outras cópias existentes. O bloco passa de *Shared* para *Modified*.

Em um *write miss*, o bloco é carregado da memória principal, caso não existam cópias, e marcado como *Modified,* e logo após é realizada a escrita. Se alguma cache possui uma cópia do bloco, ela escreve no barramento, e marca seu estado para *Invalid*. O processador da cache que fez a requisição lê esta linha, realiza sua escrita, e marca como Modified. Todas as outras cópias existentes são invalidadas.

4 MPSoC – Multi-Processor System-on-Chip

O MPSoC utilizado no presente trabalho é denominado HMPS, desenvolvido por Cristiane Woszezenki [WOS06], disponível em [MOR07]. Neste MPSoC os nodos são processadores embarcados idênticos de propósito geral.

No HMPS, um nodo mestre aloca as tarefas nos nodos escravos e cada escravo que recebe uma tarefa é capaz armazenar todas as informações referentes a esta tarefa em memória e executá-la. Para tanto, é necessário um microkernel que forneça os serviços básicos de um sistema operacional, necessários para o controle da execução das tarefas no ambiente multiprocessado e multitarefa.

O nodo mestre contém as informações de alocação de todas as tarefas a serem executadas, relacionando um conjunto de tarefas por nodo escravo.

Na inicialização do sistema, o nodo mestre carrega as tarefas de um repositório, enviando-as aos nodos escravos designados para executá-las. Como este envio de tarefas ocorre apenas na inicialização do sistema, denomina-se esta alocação como **alocação estática** que pode ser realizada aleatoriamente ou manualmente. Mas esta plataforma suporta também a **alocação dinâmica**, no qual o nodo mestre envia novas tarefas aos nodos escravos mediante requisição destes, durante a execução do sistema.

A infra-estrutura desta plataforma contém os seguintes componentes:

- NoC Hermes [MOR04]: realiza a interconexão dos núcleos e o roteamento de pacotes entre os mesmos, Figura 14 – a.
- Plasma [PLA06]: nodo processador que executa a aplicação. Cada nodo processador possui uma memória local, a qual não é acessível a outros processadores, Figura 14 – b.
- NI (Network Interface) [WOS06]: faz a interface entre o processador e a rede, Figura 15.
- DMA (Direct Memory Access) [WOS06]: transfere para a memória do processador o código-objeto das tarefas enviadas por um nodo mestre, Figura 15.

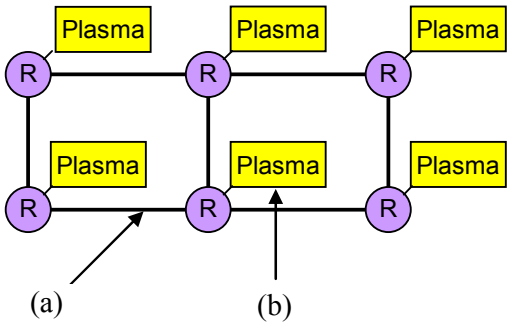

Figura 14 – (a) NoC Hermes; (b) Nodo Plasma.

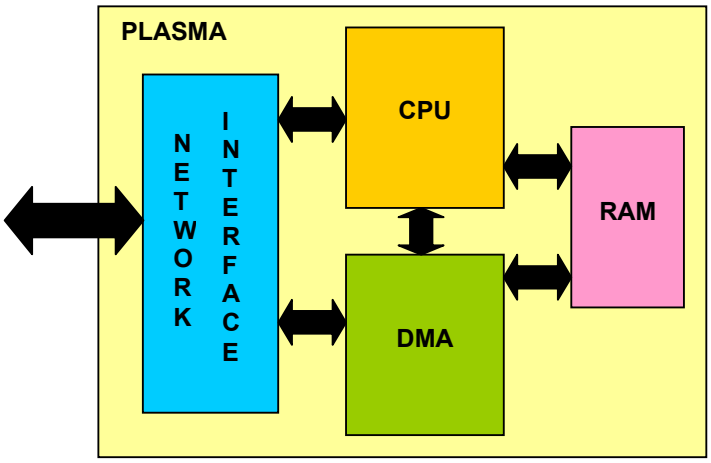

Figura 15 – Visão interna do nodo plasma, contendo os núcleos: interface de rede (NI), a CPU (processador plasma, arquitetura MIPS), DMA e RAM.

4.1 NoC Hermes

A NoC HERMES possui um mecanismo de comunicação denominado *chaveamento de pacotes*, no qual os pacotes são roteados individualmente entre os nodos sem o estabelecimento prévio de um caminho. Este mecanismo de comunicação requer o uso de um modo de roteamento para definir como os pacotes devem se mover através dos roteadores. A NoC HERMES utiliza o modo de roteamento *wormhole*, no qual um pacote é transmitido entre os roteadores em *flits*. Apenas o *flit* de cabeçalho possui a informação de roteamento. Assim, os *flits* restantes que compõe o pacote devem seguir o mesmo caminho reservado pelo cabeçalho.

A NoC HERMES utiliza uma topologia malha, ilustrada na Figura 16. Essa topologia é justificada em função da facilidade de desenvolver o algoritmo de roteamento, inserir núcleos e gerar o leiaute do circuito. Nesta NoC, os núcleos estão conectados à rede através de roteadores, que estão numerados conforme coordenada xy definidos na arquitetura.

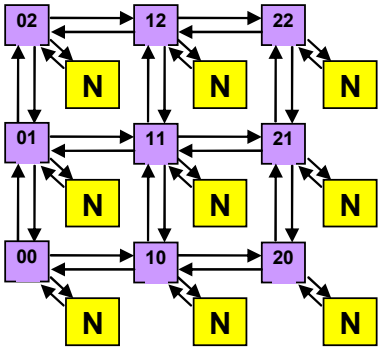

Figura 16 – Topologia malha para a NoC Hermes.

O roteador Hermes possui uma lógica de controle de chaveamento centralizada e 5 portas bidirecionais: East, West, North, South e Local. A porta Local estabelece a comunicação entre o roteador e seu núcleo local. As demais portas ligam o roteador aos roteadores vizinhos. Cada porta unidirecional (entrada ou saída) do roteador corresponde a um canal físico.

A cada porta de entrada é adicionado um *buffer* para diminuir a perda de desempenho com

o bloqueio de *flits*. A perda de desempenho acontece porque quando um *flit* é bloqueado em um dado roteador os *flits* seguintes do mesmo pacote também são bloqueados em outros roteadores. Com a inserção de um *buffer* o número de roteadores afetados pelo bloqueio dos *flits* diminui. O *buffer* inserido no roteador Hermes funciona como uma fila FIFO (First In First Out) circular, cuja profundidade é parametrizável.

A configuração da rede adotada é:

- Topologia malha 3 *x* 2, suportando 6 nodos de processamento;
- Largura de *flit* igual a 16 bits;
- Profundidade dos *buffers* igual a 8 *flits*;
- Roteamento XY e arbitragem *Round Robin.*

4.2 Processador Plasma

O Processador Plasma é um RISC de 32 bits com um subconjunto de instruções da arquitetura MIPS [HEN98]. Seu código (VHDL) é aberto, disponível através do *OpenCores* [OPE06]. O *pipeline* de instruções do processador contém três estágios: *busca, decodificação* e *execução*. Diferentemente da definição original do processador MIPS, a organização de memória do Processador Plasma é Von Neumann e não Harvard. Além disso, o Processador Plasma oferece suporte ao compilador C (gcc) e tratamento de interrupções.

A versão original do processador Plasma foi modificada para suportar o *mecanismo de paginação* tornando possível a execução de múltiplas tarefas sobre a mesma CPU, Figura 17. Este mecanismo de paginação oferece segurança no acesso à memória evitando violação de endereços. Isso quer dizer que uma tarefa residente na página p_x nunca conseguirá acessar qualquer endereço na página p_y (sendo $x \neq y$), uma vez que todo endereço lógico gerado pela tarefa será concatenado com a página p_x.

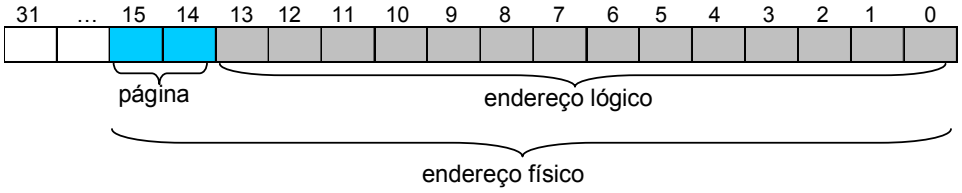

Figura 17 – Composição do endereço no processador plasma.

O processador Plasma original não oferece suporte a interrupções de software. No entanto, a aplicação pode fazer chamadas de sistema, requisitando um serviço ao sistema operacional como, por exemplo, envio e recebimento de mensagens. Dessa forma, foi necessária a inclusão da instrução syscall, prevista na arquitetura MIPS, ao conjunto de instruções do processador Plasma. Esta instrução gera uma interrupção desviando o fluxo de execução do processador para um endereço pré-estabelecido, onde a chamada de sistema é tratada.

4.3 Network Interface (NI)

A NI foi desenvolvida para realizar a interface entre o processador e a NoC HERMES. Ela é responsável por:

- Enviar pacotes para a rede, segmentando os dados deste pacote em flits. Onde a NI recebe do processador um pacote cujos dados possuem 32 bits de tamanho e os divide em segmentos de 16 bits (tamanho do flit da NoC);
- Receber pacotes da rede armazenando-os em um buffer. Quando existir um pacote completo no buffer ou quando o buffer estiver cheio, a NI interrompe o processador para que este receba os dados. Os segmentos de 16 bits referentes ao conteúdo do pacote recebido da rede são agrupados e repassados para o processador com palavras de 32 bits;
- Repassar o código-objeto de tarefas recebido da rede, através do DMA, para a memória;
- Informar ao microkernel do processador qual a sua localização na rede.

4.4 Direct Memory Access (DMA)

O DMA foi desenvolvido para permitir a alocação de tarefas nos processadores do sistema. O código objeto de uma tarefa é enviado pela rede para um determinado processador. O DMA é responsável por transferir este código objeto para a memória do processador.

Assim como qualquer outro pacote, quem recebe o código objeto da tarefa é a NI, armazenando-o no *buffer* de recebimento. Uma vez o pacote recebido, as seguintes operações são realizadas:

1. A NI interrompe a CPU informando a chegada de um pacote;
2. O *microkernel*, que executa na CPU e faz parte da infra-estrutura de software do sistema, interpretará o pedido de interrupção como nova tarefa a ser alocada. O *microkernel* obtém o identificador da tarefa e o tamanho do código objeto da mesma e verifica a disponibilidade de página livre na memória. A CPU informa ao DMA o endereço da memória a partir do qual o código objeto deve ser transferido e o tamanho do código objeto;
3. O DMA faz acessos à NI para ler o código objeto e acessos à memória para escrever o mesmo. Quando o código já estiver alocado, o DMA interrompe a CPU informando que uma nova tarefa está na memória;
4. O *microkernel* faz as inicializações da tarefa, e a partir disso a tarefa executará quando for escalonada.

Este mecanismo permite que o processador realize a execução das suas tarefas em paralelo com a recepção de novas tarefas. A comunicação entre o *microkernel* e o DMA ocorre através de registradores mapeados em memória.

A máquina de estados do DMA é mostrada na Figura 18. O estado inicial é Swait, no qual é conhecido o endereço da memória a partir do qual o código objeto vai ser transferido e o tamanho do código objeto. Também neste estado a interrupção é desativada, após a CPU informar que reconheceu a interrupção. Se a CPU informa ao DMA que pode iniciar a transferência, o estado

avança para Scopy. Neste estado, cada dado referente ao código objeto é buscado na NI e escrito na memória do processador. A cada escrita, o endereço da memória é incrementado e o tamanho do código objeto é decrementado. Quando o tamanho do código objeto chega à zero, máquina avança para o estado avança para Send, onde a escrita na memória é desabilitada e a CPU é interrompida. O estado passa a ser Swait novamente.

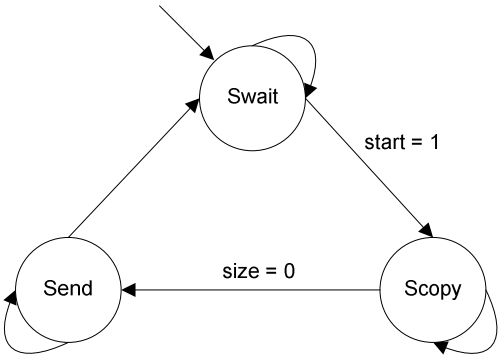

Figura 18 – Máquina de estados do DMA.

5 Hierarquia de memória no MPSoC HMPS

Nesta Seção é apresentada a arquitetura desenvolvida durante este Trabalho de Conclusão. O objetivo desta arquitetura é garantir uma comunicação segura e rápida entre os processadores e a memória compartilhada, criada para esta arquitetura. Foi incluído também memórias caches em cada processador, para se ter um maior desempenho na comunicação.

Foi escolhido o **mapeamento direto**, pela sua simplicidade de implementação.

A política de escrita é a **write-back**, neste caso o processador escreve os dados diretamente na memória cache, cabendo ao controlador a escrita posterior da informação na MP. Assim, o processador fica livre mais rapidamente para executar outras operações. Em contrapartida, a latência do controlador pode induzir problemas de consistência de dados na MP, tratados por protocolos de consistência de cache.

A coerência de cache é baseada em diretórios, sendo controlado na MP. Permitindo assim que os comandos de consistência sejam enviados apenas para as caches que possuam cópia do objeto. Para realizar esta operação, uma entrada na memória diretório esta associando o objeto a cada processador.

5.1 Memória Principal (MP)

A Memória Principal, situada no nodo de rede 10, Figura 19, é uma memória compartilhada, podendo ser acessada e ter suas informações alteradas por qualquer processador presente na NoC. O controlador desta memória é capaz de interpretar os pacotes da rede de escrita ou leitura. No caso de escrita, o controlador escreve na memória os dados contidos no pacote de escrita e, no caso de leitura, o controlador envia ao processador que realizou a requisição um pacote contendo os dados lidos a partir do endereço que foi solicitado e na quantidade que também foi definida no pacote de leitura.

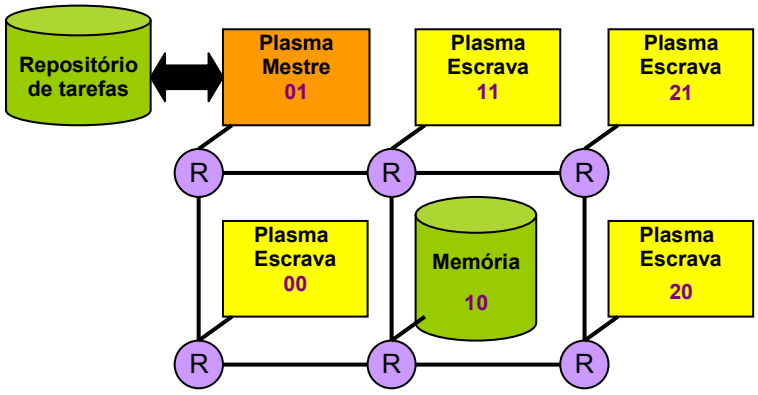

Figura 19 – Plataforma MPSoC com memória compartilhada.

A arquitetura da MP, mostrada na Figura 20, é dividida em 3 partes: **NI**, que é a interface entre a NoC e o controlador de memória; o **Controlador de memória**, que faz todo o gerenciamento para escrita na memória principal e cuida da coerência dessas escritas e leituras; a **memória**, que contém as informações armazenadas.

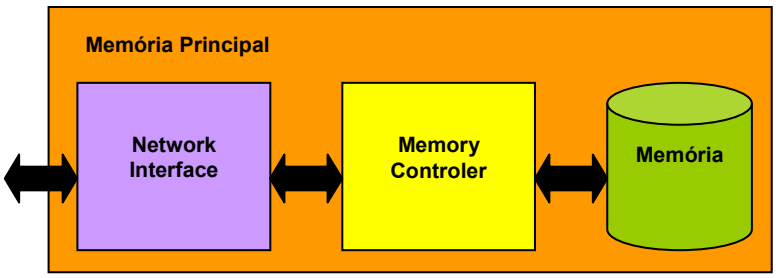

Figura 20 – Arquitetura da memória principal.

A memória principal (MP) é dividida em blocos (cada bloco representa uma unidade de informação) com tamanho igual a 256 bytes (64 words). A MP é capaz de armazenar *k* objetos, sendo que cada objeto pode ter *n* blocos. A MP foi dimensionada com 128 KB, resultando em uma capacidade de 512 blocos. A MP é endereçada a palavra (word), e pode ser vista como um vetor de 32768 endereços (512 *blocos* * 64 $\frac{word}{bloco}$) de 32 bits. Para endereçar a MP são necessários 17 bits, sendo os 9 mais significativos para endereçar o bloco, os 6 intermediários endereçam uma palavra no bloco e os 2 menos significativos um byte na palavra. Para endereçamento, não são usados os 2 últimos bits, resultando em 15 bits de endereço, como mostrado na Figura 21.

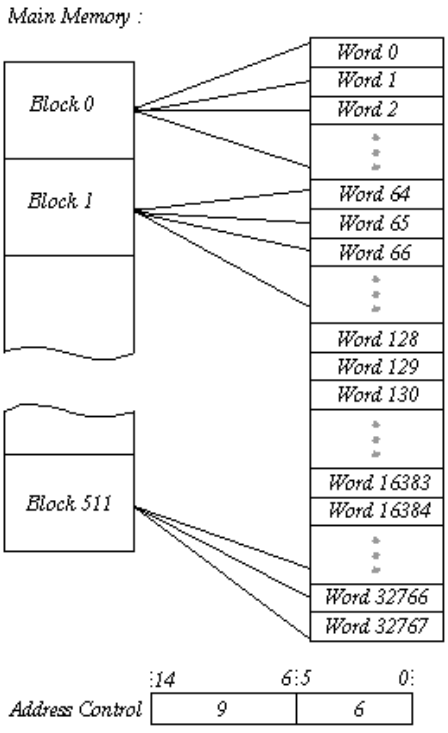

Figura 21 – Estrutura de endereçamento da memória compartilhada.

A MP contém uma **memória diretório**, conforme a Figura 22. Este diretório serve para armazenar informações referentes à configuração e ao uso dos blocos da MP, informações também usadas para manter a coerência de memória. Seus campos são:

- *Objeto*: é o campo de endereço da tabela, o qual indexa um determinado bloco;
- *Bloco Inicial*: contém o valor do bloco inicial de um objeto;
- *Bloco Final*: contém o valor do bloco final de um objeto;
- *Inicializado*: indica que o objeto foi inicializado;
- *Shared*: se shared = '1', indica que o objeto esta compartilhado entre as CPUs. Se shared = '0', indica que o objeto esta exclusivo à uma CPU, usado para uma escrita no bloco. Este campo só é considerado quando inicializado = '1';
- *CPU N*: indica que a CPU N (N podendo ser de 1 a 4) contem todo ou parte do objeto em sua cache. Se o bit referente à CPU for igual a '1'. Tem 4 CPUs porque é uma arquitetura 2 x 3, sendo um processador mestre e uma memória, restando 4 processadores escravos (que são indicadas pela CPU).

Objeto	Bloco Inicial	Bloco Final	Inicializado	Shared	CPU 1	CPU 2	CPU 3	CPU 4
1								
2								
.....								
k								

Figura 22 – Diretório para coerência de cache.

Para se ter acesso ao módulo MP, foi necessária a implementação de funções de leitura e escrita. Foram implementadas as rotinas de software *NewObject*, *ReadGlobal* e *WriteGlobal*, as quais se comunicam com o wrapper do processador para a comunicação com a MP através da NoC.

- *NewObject*: inicializa um novo objeto na MP. Esta função envia à MP o número do objeto, e os índices dos blocos inicial e final. Atualiza o campo *inicializado* do diretório colocando em '1'. Seu modo de usar é: *NewObject(&obj_mp).*
- *ReadGlobal*: função que recebe duas estruturas, a 1ª estrutura contém os valores lidos no bloco da MP, a 2ª é uma estrutura com as informações pedidas para a MP. Seu modo de usar é: *ReadGlobal(&msg_mp, &block_mp).*
- *WriteGlobal*: função que recebe uma estrutura, que é uma estrutura que possui a informação a ser escrita no bloco da MP. Seu modo de usar é: *WriteGlobal(&msg_mp).*

A Figura 23 apresenta as estruturas (structs) utilizadas por estas funções.

```
/*-------------------------
 * struct Object MP
 *-------------------------*/
typedef struct {
    int objeto;
    int block_start;
    int block_end;
} ObjectMP;
```

```
/*-------------------------
 * struct Message MP
 *-------------------------*/
typedef struct {
    int length;
    int objeto;
    int block;
    int pos_start;
    int pos_end;
    int msg[MAXMSG];
} MessageMP;
```

```
/*-------------------------
 * struct Block MP
 *-------------------------*/
typedef struct {
    int objeto;
    int bloco;
    int pos_start;
    int pos_end;
} BlockMP;
```

Figura 23 – Estrutura da memória compartilhada.

Estas funções enviam pacotes de leitura/escrita ao nodo de rede que foi destinado à MP. Este ao receber os pacotes, os interpreta e executa a ação requisitada. As ações são executadas pelo seu ***controlador***.

5.1.1 Controlador da MP

Este controlador foi desenvolvido em VHDL. Devido a sua complexidade (47 estados), iremos analisar por partes, indicando os comandos e qual sua seqüência na máquina de estados. Os serviços suportados pelo controlador da MP são:

- *Object*: serve para tratar mensagens contendo objetos, como a inicialização de um novo objeto;
- *Request_mp*: trata um read miss;
- *Deliver_mp*: trata um write miss;
- *Block*: são semelhantes ao tipo objeto, só que este trata mensagens contendo blocos. É usado para atualização da MP, **write-back**;
- *Invalidade*: comando para a MP invalidar um bloco do controlador de cache;
- *Exclusive*: comando usado pelo controlador de cache para pedir à MP a exclusividade de um bloco, necessário para um *write*.

5.1.1.1 OBJECT

Para se manipular os objetos, foram criados dois comandos para efetuar a troca de objetos pela NoC. O comando *NEW_OBJECT* cria um objeto na MP. Durante a execução deste comando a MP recebe do processador as informações e atualiza o **diretório**. Já o comando *REQUEST_OBJECT* serve para ler um objeto da MP, esta comunicação é entre um controlador de cache e a MP. Este comando é usado para atualizar a tabela do controlador de cache. Os estados do controlador utilizados para tratar estes serviços (Figura 24) compreendem:

- S0 à S3: Recebe as informações do header do pacote. Note que todos os blocos recebidos têm essas informações como padrão.
- S4 à S6: Recebe as informações para criar um novo objeto com seu respectivo bloco inicial e bloco final (*NEW_OBJECT*).

- S7: Recebe a informação sobre qual objeto o controlador de cache esta pedindo (*REQUEST_OBJECT*).
- S8 à S11: Se o objeto foi inicializado, ela envia o objeto pedido pelo controlador de cache (*DELIVER_OBJECT*), caso contrário, envia um pacote informando que não tem o objeto pedido (*NO_OBJECT*).

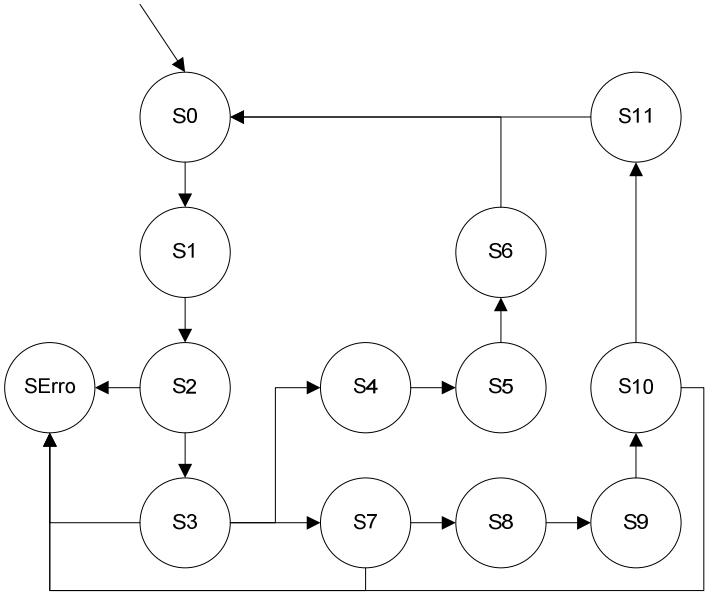

Figura 24 – Máquina de estados dos comandos: *New_Object, Request_Object, Deliver_Object e No_Object.*

5.1.1.2 REQUEST_MP

Comando usado para ler um bloco da MP, esta comunicação é entre um controlador de cache e a MP. Onde:

- S0 à S12 e S14 à S17: Recebe as informações sobre qual bloco o controlador de cache está pedindo.
- S30: Estado que verifica se este bloco esta compartilhado ou exclusivo. Se compartilhado, vai para o estado *S19*, caso contrário vai para o estado *S31*.
- *S19 à S29: Envia o bloco pedido pelo controlador de cache.*
- S31: Estado que verifica se já pode ser enviado o bloco que estava exclusivo. Se sim, vai para o estado *S19*, caso contrário, temos que verificar a coerência deste bloco com o outro processador que estava com ele em exclusivo, indo para o estado *S38*.
- S38 à 40: Estados em que é enviado um pacote para o processador dono do bloco em exclusivo e pede para seu processador entrar pause, para poder se comunicar com o controlador de cache, com este pause, o controlador de memória cache já sabe que a MP vai querer um bloco.

5.1.1.3 DELIVER_MP

Comando para escrever um bloco da MP, esta comunicação é entre um controlador de

cache e a MP. Os estados do controlador utilizados para tratar estes serviços (Figura 25) compreendem:

- S0 à S17: Recebe as informações sobre qual bloco o controlador de cache está escrevendo.
- S18: Estado em que o controlador está enviando a mensagem.

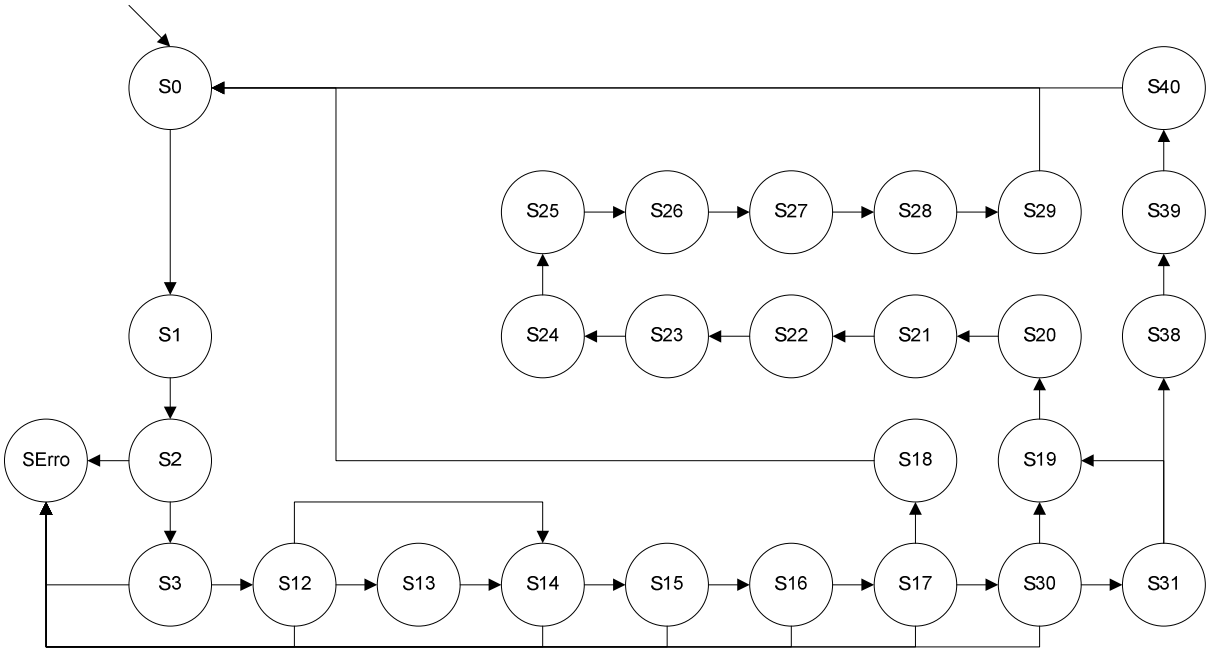

Figura 25 – Máquina de estados dos comandos: *Request_MP e Deliver_MP.*

5.1.1.4 BLOCK

Seção de comandos referentes à troca de blocos pela NoC, nela inclui os comandos *PAUSE_PROC_BLOCK*, *PAUSE_BLOCK*, *REQUEST_BLOCK*, *NO_BLOCK* e *CHANGE_BLOCK*. O 1° comando é um pacote enviado pela MP onde faz parar o processador, entrando em ação o controlador de cache, já o 2° comando é a resposta do controlador, indicando que esta pronto para devolver os blocos que serão pedidos pela MP. O 3° comando é o pacote requisitando um bloco. E os dois últimos comandos são as respostas dadas pelo controlador, se modificou ou não o bloco requerido pela MP. Os estados do controlador (Figura 26) utilizados para tratar estes serviços compreendem:

- S0 à S2 e S36: Recebe um pacote dizendo que o processador não mudou o bloco pedido pela MP (*NO_BLOCK*).
- S0 à S3 e S37: Recebe um pacote dizendo que o processador mudou o bloco pedido pela MP (*CHANGE_BLOCK*), este pacote já contém todo o bloco mudado.
- S19 à S29: Envia uma requisição de bloco para o controlador de cache (*REQUEST_BLOCK*).
- S31: Estado em que verifica se a MP já pode enviar o bloco que estava exclusivo. Se sim, vai para o estado *S19*, caso contrário, é necessário verificar a coerência deste bloco com o outro processador que estava com ele em exclusivo, indo para o estado *S38*.

- S38 à 40: Estados em que se envia um pacote para o processador entrar em estado de pause e o controlador de cache assumir o barramento (*PAUSE_PROC_BLOCK*), com este pause, o controlador já sabe que a MP vai querer um bloco.

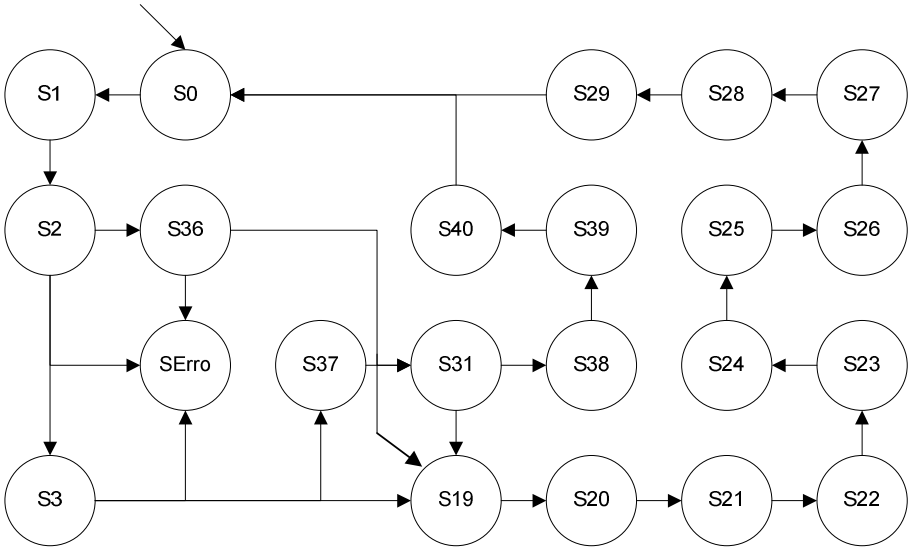

Figura 26 – Máquina de estados dos comandos: *Pause_Proc_Block, Pause_Block, No_Block e Change_Block.*

5.1.1.5 INVALID e EXCLUSIVE

Seção de comandos referentes a invalidações de objetos e blocos ou pedidos de exclusividade de blocos pela NoC, nela inclui os comandos *PAUSE_PROC_INVALID*, *PAUSE_INVALID*, *DELIVER_INVALID, INVALID_OK, REQUEST_EXCLUSIVE* e *DELIVER_EXCLUSIVE*. O 1° comando é um pacote enviado pela MP onde faz parar o processador, entrando em ação o controlador de cache, já o 2° comando é a resposta do controlador, indicando que está pronto para invalidar o bloco que será passado pela MP. O 3° comando é o pacote escrevendo qual bloco será invalidado. O 4° comando é a resposta dada pelo controlador, dizendo que invalidou o bloco informado pela MP. O 5° comando é o pedido de um controlador de cache para ter um bloco exclusivo. E o último comando indica que a MP está avisando o controlador de cache para atualizar seus dados, pois ele será dono exclusivo daquele bloco. Os estados do controlador (Figura 27) utilizados para tratar estes serviços compreendem:

- S0 à S3: Recebe as informações do header do pacote, neste caso, poderá ser um pacote recebido do controlador de cache indicando que esta pronto para receber a informação de qual bloco ele irá invalidar (*PAUSE_INVALID*).
- S14 à S15: Estado que recebe um pacote onde o processador requisita exclusividade de um bloco (*REQUEST_EXCLUSIVE*).
- S41 à S44: Envia um pacote com o bloco para o controlador de cache invalidar (*DELIVER_INVALID*).
- S45: Recebe a resposta do controlador de cache que invalidou o bloco pedido pela MP (*INVALID_OK*).

- S32: Estado de decisão, para saber se invalidará um processador, ou se dirá a um processador que será exclusivo.
- S33 à S35: Estado que envia um pacote para o processador mudar o estado do bloco para exclusivo (*DELIVER_EXCLUSIVE*).
- S31 e S38 à S40: Estado que envia um pacote para o processador entrar em estado de pause e o controlador de cache assumir o barramento (*PAUSE_PROC_INVALID*), com este pause, o controlador já sabe que a MP deseja invalidar um bloco.

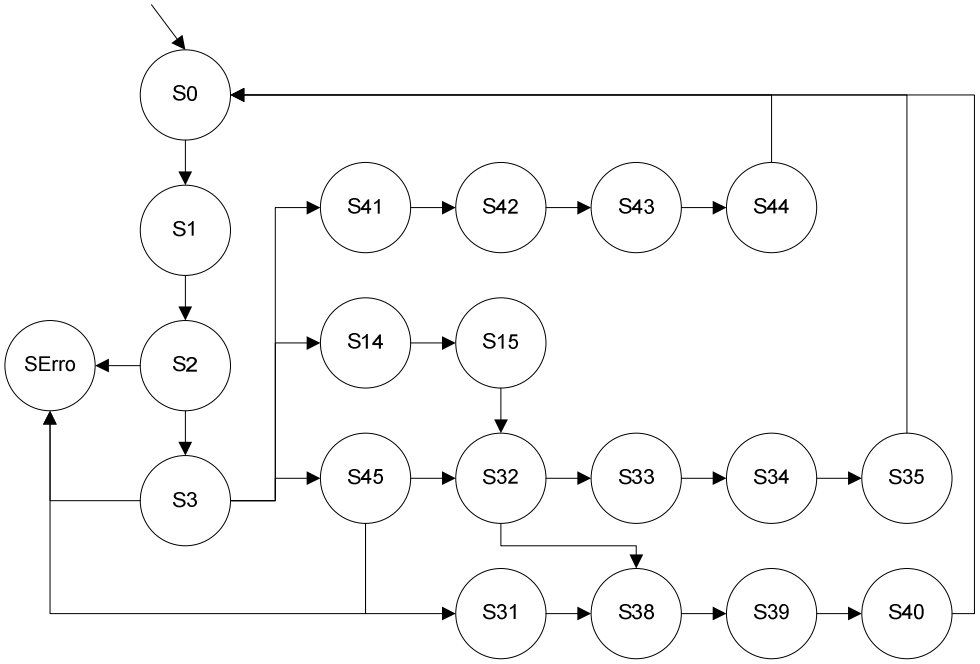

Figura 27 – Máquina de estados dos comandos: Pause_Proc_Invalid, Pause_Invalid, Deliver_Invalid, Invalid_Ok, Request_Exclusive e Deliver_Exclusive.

5.2 Memória Cache

Cada processador Plasma possui uma memória cache para dados, que serve para reduzir o tempo de acesso à um dado que já esteja presente nela, pois evita do processador ter que utilizar a NoC para requisitá-lo à MP. Diagrama de blocos do nodo processador com a cache é apresentado na Figura 28.

Como característica de uma memória cache, esta memória possui um tamanho limitado, sendo necessário uma gerência sobre a manipulação dos dados, algo que é feito pelo controlador de memória cache.

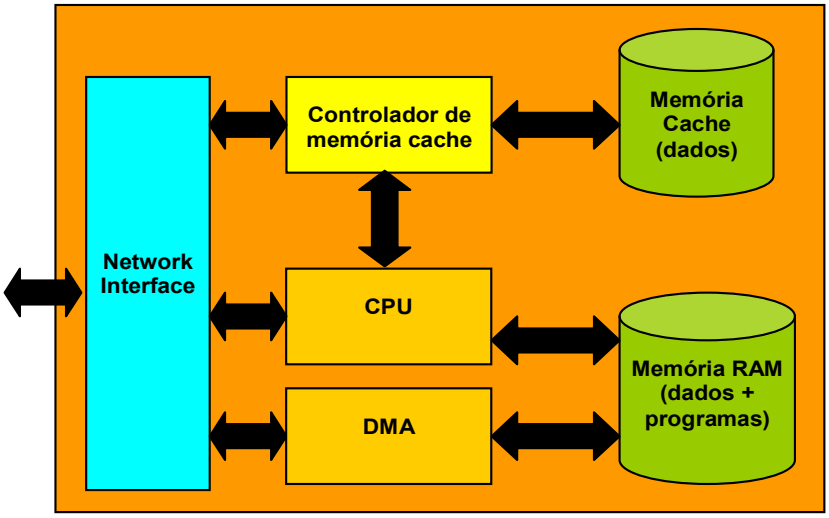

Figura 28 – Arquitetura do nodo plasma com memória cache.

A Memória Cache foi dividida em blocos, possuindo capacidade total para 8KB (13 bits), capacidade que proporciona 32 blocos por CPU, sendo os 5 mais significativos para endereçar o bloco, os 6 intermediários endereçam uma palavra no bloco e os 2 menos significativos um byte na palavra. Para endereçamento, não são usados os 2 últimos bits, resultando em 11 bits de endereçamento, apresentado na Figura 29.

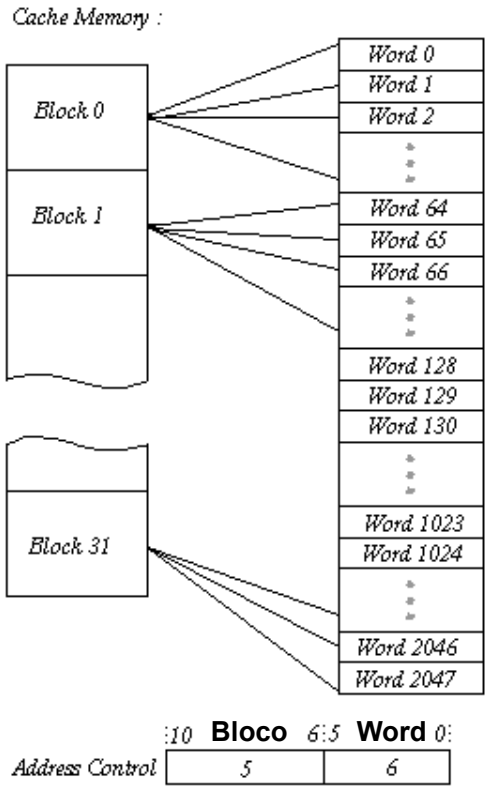

Figura 29 – Estrutura de endereçamento da memória cache.

Para se ter acesso a essa memória cache, foi necessária a inclusão de funções que iniciem este processo de comunicação. Foram incluídos mecanismos de software para acesso a esta

memória cache, correspondendo às rotinas ReadLocal e WriteLocal.

- *Read:* função que recebe uma estrutura que possui a informação a ser lida no bloco da memória cache, contendo a descrição do que será lido da memória cache. Seu modo de usar é: *ReadLocal(&msg_cache).*
- *Write:* função que recebe uma estrutura que possui a informação a ser escrita no bloco da memória cache, que contém uma descrição do que será escrito na memória cache. Seu modo de usar é: *WriteLocal(&msg_cache).*

A Figura 30 apresenta a estrutura (struct) utilizada nas rotinas de acesso à memória cache.

```
/*-------------------------
 * struct Message Cache
 *-------------------------*/
typedef struct {
    int objeto;
    int bloco;
    int pos;
    int value;
} MessageCache;
```

Figura 30 – Estrutura da memória cache.

5.3 Controlador de Cache

A memória cache é uma memória gerenciada por um controlador (controlador de memória cache). Este controlador possui tabelas de tradução de endereços (**Obj2Block** e **TagMemory**), política de mapeamento **direto**, política de escrita **write-back**, interface com o processador Plasma e a NoC (Figura 31).

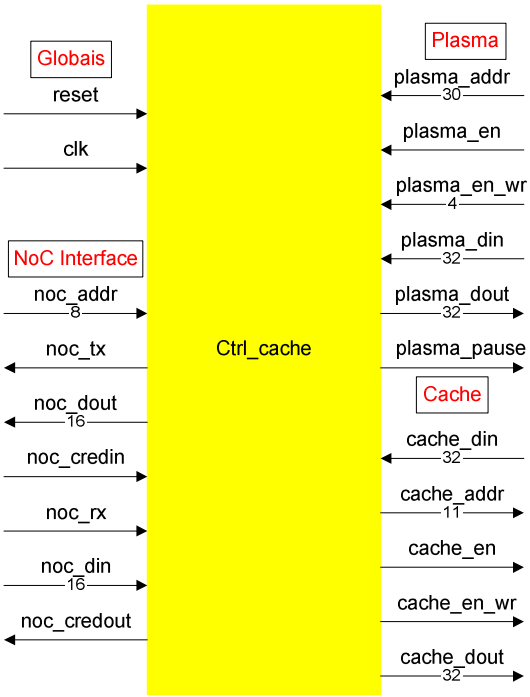

Figura 31 – Interface do controlador de cache.

O controlador de cache foi implementado em VHDL. Sua estrutura de blocos é mostrado na Figura 32, utilizando mapeamento direto para indexar os elementos armazenados na memória cache. O endereço gerado pelo processador Plasma é dividido em duas partes principais:

- Objeto: utilizado para indexar um objeto na tabela Object2Block.
- Elemento: os bits 14 a 6 endereçam o número do bloco, e os bits de 5 a 0 endereçam uma palavra no bloco.

Cada controlador de memória cache requer duas memórias auxiliares: Object2Block, TagMemory:

- Object2Block: é uma tabela de conversão de um dado número de objeto em um número de bloco na MP. Onde se encontram o Bloco Inicial e Bloco Final de um objeto, além do sinal Shared, que indica se aquele objeto está sendo utilizado na cache local ou não.
- TagMemory: é uma tabela com 32 entradas (lembrar 8KB/256B), contendo os campos Tag (valor do bloco na cache), Exclusive (bit que indica se o bloco é exclusivo à esta memória cache) e valid (bit indicando se este bloco possui os dados válidos).

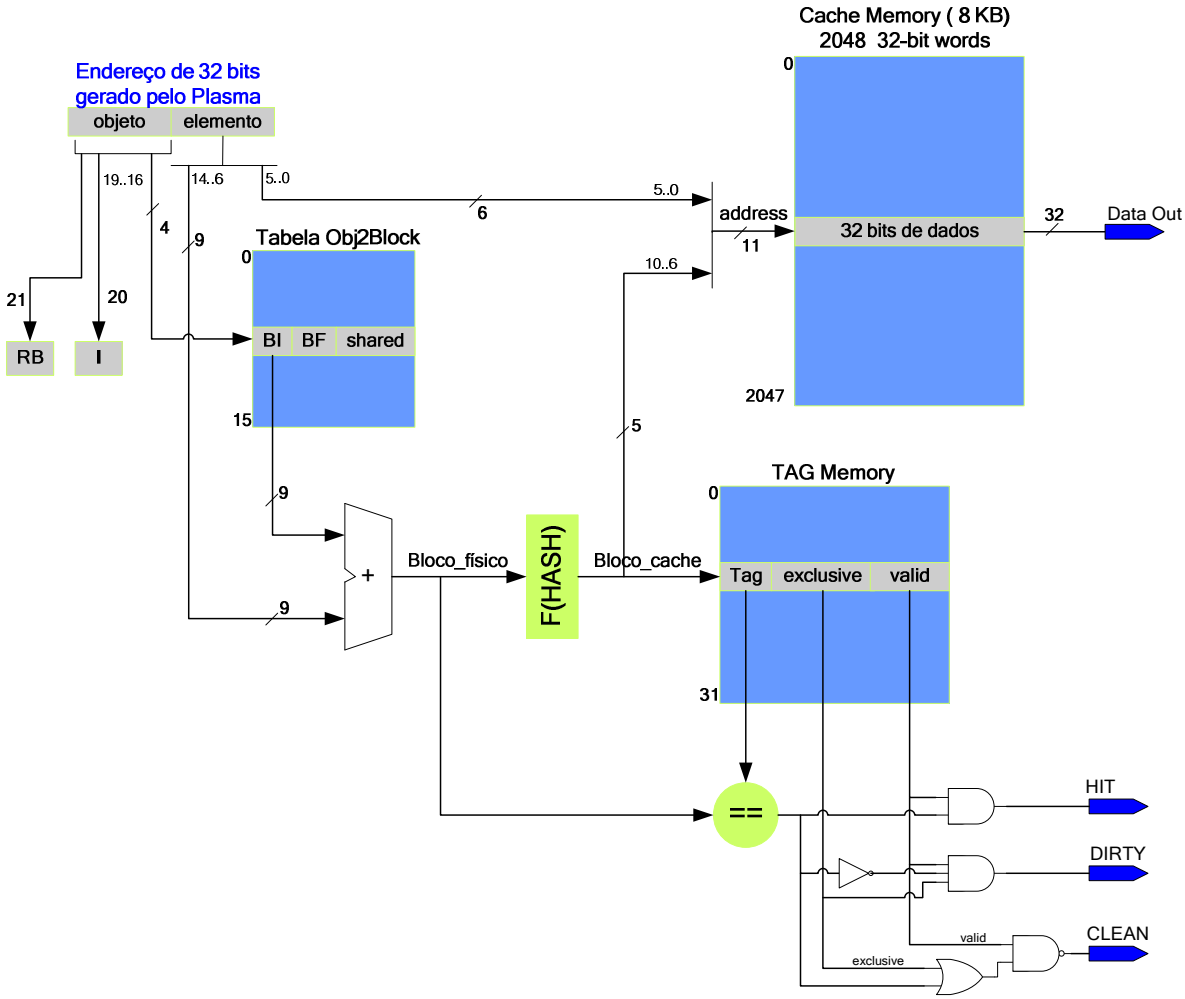

Figura 32 – Estrutura de blocos do controlador de cache.

Ainda na Figura 32, temos dois registradores *RequestBlock* (**RB)** e *Invalid* (**I**). Onde:

- RB: indica que a MP quer atualizar seu diretório. Quando setado este registrador, o processador entra em pause e o controlador começa a comunicação com a MP, a fim de enviar-lhe um bloco;
- I: indica que a MP quer invalidar um objeto. Quando setado este registrador, o processador entra em pause e o controlador começa a comunicação com a MP, a fim de tirar este objeto da tabela do controlador de cache.

Podem-se observar sinais de Hit, Dirty e Clean. Sinais interpretados pelo controlador no que diz respeito sobre o estado em que se encontra o objeto na qual a linha da Tag Memory referencia. A interpretação destes sinais pode ser visto na Tabela 1.

Tabela 1 – Decisão da TAGMemory.

Válido	Tags	Exclusive	Resultado
0	0	0	**CLEAN='1'** - independentemente dos sinais de comparação (==) e exclusive, se o campo válido está zerado, pode-se carregar um novo bloco na posição endereçada por *bloco_cache*.
0	0	1	
0	1	0	
0	1	1	
1	0	0	**CLEAN='1'** – mesmo o válido estando em '1', se o bloco não está alterado (exclusive=0), pode-se sobrescrever sobre o bloco atual.
1	0	1	**DIRTY='1'** – um outro bloco está no lugar (tags==0) e este bloco está alterado (exclusive=1). Logo, deve-se fazer um ***write-back*** deste bloco, antes de se escrever um bloco novo.
1	1	0	**HIT='1'** – a comparação dos tags é verdadeira e o bloco está válido
1	1	1	

Este controlador possui 20 estados, apresentados na Figura 33, onde:

- W: Estado em espera, fica aguardando alguma requisição de leitura ou escrita;
- G1: Estado de decisão, em que verifica se a MP quer invalidar um bloco (pause no processador por invalidação), se a MP quer requisitar um bloco em que este controlador é dono exclusivo (pause no processador para atualização da MP), ou se for outra operação, é verificado se o objeto está compartilhado (shared com valor igual a '1') indo para outro estado de decisão, caso contrário, irá para um estado onde requisitará dados referente ao objeto;
- G2: Estado que envia um pacote requisitando as configurações referentes ao objeto solicitado pela MP. Configurações de Bloco Inicial (BI) e Bloco Final (BF);
- G3: Estado que recebe um pacote com as configurações do objeto (BI e BF) da memória principal e atualiza sua tabela obj2block, colocando também shared em '1';
- G4: Estado de decisão, se for um Read Hit vai para o estado R1, se for um Write Hit e ele estiver com o bloco em exclusivo, irá para o estado W1, se o sinal clean = '1' então ele requisitará os dados referente a este bloco à MP (estado S1), caso o sinal dirty = '1', então ele enviará o bloco à MP, sobrescrevendo o dado antigo, fazendo assim um flush dos dados da MP;
- G7: Estado onde sinaliza a ocorrência de algum erro;
- R1: Estado que tira o processador do estado busy, estado em que o processador está em pause aguardando uma resposta, e lê o dado requisitado da Memória Cache;

- W1: Estado que tira o processador do estado busy, estado em que o processador está em pause aguardando uma resposta, e escreve o dado na Memória Cache;
- WB: Estado que envia o bloco alterado para a MP, onde será atualizado;
- S1: Estado que recebe da MP um pacote com os dados de um bloco físico;
- S2: Estado de atualização da tabela tagmemory (tag = n° do bloco, exclusive = '0', valido = '1');
- W2: Estado que envia um pacote solicitando permissão à MP para obter acesso exclusivo ao bloco físico;
- W3: Estado que recebe um pacote com permissão de exclusividade sobre o bloco da MP, atualizando sua tabela tagmemory (exclusive = '1');
- I1: Estado que envia um pacote à MP informando que o processador entrou em pause e que espera um pacote sobre qual bloco ela quer invalidar;
- I2: Estado que recebe um pacote dizendo qual bloco de qual objeto o MP quer invalidar da memória cache;
- I3: Estado que envia um pacote à MP confirmando a invalidação daquele bloco na sua memória cache;

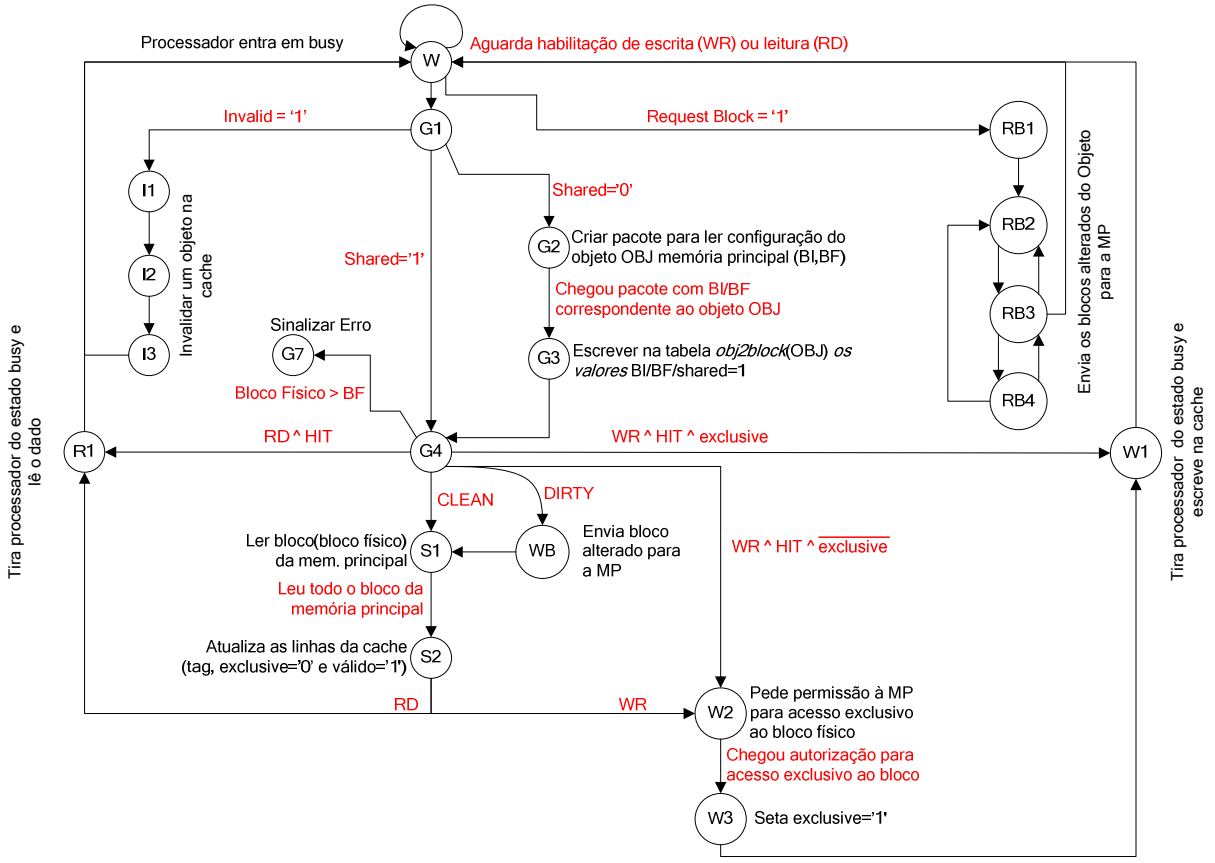

Figura 33 – Máquina de estados do controlador de cache.

- RB1: Estado que envia um pacote à MP informando que o processador entrou em pause e que espera um pacote sobre qual bloco a MP quer atualizar;
- RB2: Estado que recebe um pacote dizendo qual bloco de qual objeto o MP quer os dados, neste estado também, o controlador verifica se escreveu neste bloco;
- RB3: Estado que envia um pacote à MP informando se modificou o bloco ou não, indo para o estado RB4 caso tenha alterado o bloco, caso contrário verifica se é o ultimo bloco do objeto ou não, se sim, finaliza as operações de atualização da MP, caso contrário, volta ao estado RB2 esperando pelo próximo bloco;
- RB4: Estado que envia os dados do bloco modificado pelo controlador, após, verifica se é o ultimo bloco do objeto ou não, se sim, finaliza as operações de atualização da MP, caso contrário, volta ao estado RB2 esperando pelo próximo bloco.

5.4 Plasma_New (Plasma Cache)

O Plasma Cache é o módulo desenvolvido neste trabalho para suportar memória cache. Ele é baseado no módulo modificado do trabalho de mestrado da Woszezenki [WOS06], acrescentando 2 módulos, a cache e o controlador de cache, mostrado na Figura 34.

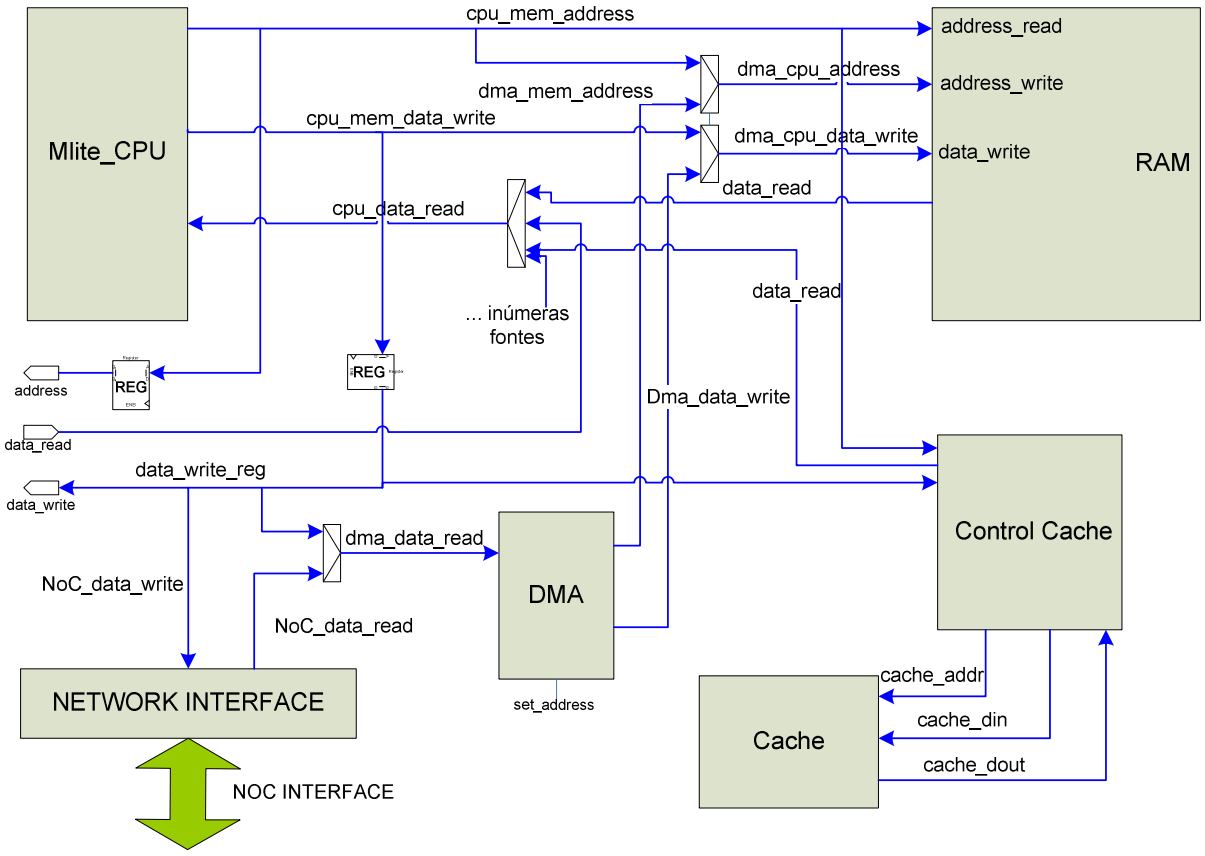

Figura 34 – Ligação da arquitetura do processador plasma com memória cache (principais sinais).

Os blocos compreendem:

- Mlite_CPU: Processador Plasma.
- DMA: Módulo para transferir o código-objeto de tarefas que chegam na Network Interface para a memória do processador.
- RAM: Memória para dados e programa.
- Network Interface: Módulo de interface entre o processador Plasma e a rede NoC.
- Cache: Memória Cache de 8KB, usado só para dados.
- Control Cache: Controlador que irá tratar as escritas e leituras na memória cache.

6 Estratégia de Validação Funcional da MPSoC com memória

A fim de se validar a arquitetura construída, foram implementados programas de teste para detectar possíveis falhas e também quantizar o tempo necessário para a memória cache retornar uma resposta em determinadas situações (cache miss, cache hit).

A bateria de testes conta com a Memória Principal inicializada com dois objetos, conforme pode ser visto na Tabela 2. Estes objetos são lidos e alterados pelos programas de teste durante sua execução, tanto os dados na MP quanto as suas réplicas presente nos processadores e os seus respectivos sinais de controle devem ser coerentes ao final dos testes.

Tabela 2 – Memória Diretório da Memória Principal com dois objetos inicializados.

Objeto	Bloco Inicial	Bloco Final	Inicializado	Válido	CPU 1	CPU 2	CPU 3	CPU 4
0	0	0	1	1	0	0	0	0
1	1	2	1	1	0	0	0	0
.....								
k								

Foram efetuados onze testes que estão descritos nos itens de 6.1 a 6.11. Cada teste tem um ponto de verificação a ser observado, que nada mais é do que observar se o funcionamento dos componentes envolvidos no teste seguem o funcionamento especificado.

6.1 Write Miss Sem Objeto

O programa causa um write miss na memória cache para testar a seqüência de trazer dados da MP. A Memória Cache deve setar o *exclusive* referente a este bloco recém lido.

- Ponto de verificação: o diretório deverá apresentar o campo válido com ‘0’ e a CPU *x* deve também ir a ‘1’ indicando que o processador é dono de objeto.

Este teste começa com a MP tendo o objeto 1 inicializado (Tabela 3), com bloco inicial no endereço 0 e o bloco final no endereço 2, inicializado em ‘1’, válido em ‘1’, CPU 1 até CPU 4 em ‘0’ mostrando que nenhuma CPU possui cópia deste objeto em sua memória local (memória cache).

Tabela 3 – Memória diretório da MP inicializada.

Objeto	**Bloco Inicial**	**Bloco Final**	**Inicializado**	**Válido**	**CPU 1**	**CPU 2**	**CPU 3**	**CPU 4**
0	3	3	1	1	0	0	0	0
1	**0**	**2**	**1**	**1**	**0**	**0**	**0**	**0**
.....								

Interação deste programa de teste 1 com a memória cache:

- Escrita em um bloco do objeto 1;
- Configuração deste objeto não consta na memória cache (MC), o controlador de MC vai do estado W para o estado G1 e prepara a requisição de configuração do bloco;
- Requisição de configuração do bloco é enviada durante o estado G2 (Figura 36 – 2);

- Requisição recebida pela MP (Figura 36 – 3);
- MP envia a configuração (Figura 36 – 4);
- Configuração recebida pelo controlador de MC durante o estado G3 (Figura 36 – 6), configuração escrita na **object2Block** do controlador de MC.

Após o controlador da MC receber os dados referentes à configuração do objeto 1, passa para o estado G4, ao verificar a ausência do Bloco Físico referente ao objeto 1 na MC é necessário requisitar este conteúdo à MP:

- Controlador vai para o estado S1 (Figura 36 – 7) para ler o Bloco Físico;
- Durante o estado S1, os dados são requisitados à MP (Figura 36 – 8);
- MP altera o valor referente ao objeto em sua memória diretório de 0xB0 para 0xB2 (Figura 36 – 9). O valor 0xB0 é equivalente a 1011**0000** em binário, os quatro últimos bits estão em zero mostrando que nenhuma CPU possui o objeto. Quando este valor passa para 0xB2, que é 1011001**0** em binário, pode-se notar que dentre os quatro últimos bits o penúltimo foi alterado de zero para um, indicando que a CPU 3 possui uma cópia do objeto;
- MP envia os dados (Figura 36 – 10) referentes ao bloco (64 palavras) que foi requisitado pelo processador;
- Dados recebidos no controlador de MC (Figura 36 – 11);
- Os dados são gravados na tag memory (Figura 36 – 12) do controlador de MC durante o estado S2.

Como é desejado efetuar uma escrita em um bloco deste objeto, é necessário requisitar acesso exclusivo ao objeto à MP:

- Requisita acesso exclusivo ao objeto durante o estado W2 (Figura 36 – 13)
- MP recebe solicitação de acesso exclusivo (Figura 36 – 14);
- MP marca em sua memória diretório qual processador plasma tem acesso exclusivo a este objeto (Figura 36 – 15) ao passar o valor de 0xB2 para 0xA2 (Tabela 4), indicando que o valor referente a este objeto armazenado na MP não é mais válido, acaba de ser **invalidado**;
- MP envia a permissão de acesso exclusivo ao processador Plasma que a solicitou (Figura 36 – 16);
- Controlador de MC recebe a permissão de acesso exclusivo da MP (Figura 36 – 17);
- Durante o estado W3 o exclusivo é passado de ‘0’ para ‘1’ (Figura 36 – 18).

Finalmente com a configuração do objeto 1 armazenada no controlador de MC, os dados referentes ao objeto 1 armazenados na MC e a garantia de exclusividade sobre ele, é momento efetuar a operação originalmente solicitada pelo processador, uma escrita na MC:

- Escrita é efetuada durante o estado W1 (Figura 36 – 19);
- Escrita completada, o controlador retorna ao estado inicial W (Figura 36).

A seqüência de eventos acima descrita pode ser visualizada na Figura 35. Observar nesta Figura a seqüência de 4 eventos: (1) requisição e recepção da descrição do objeto; (2) requisição e recepção do bloco; (3) requisição de exclusividade; (4) escrita na memória cache.

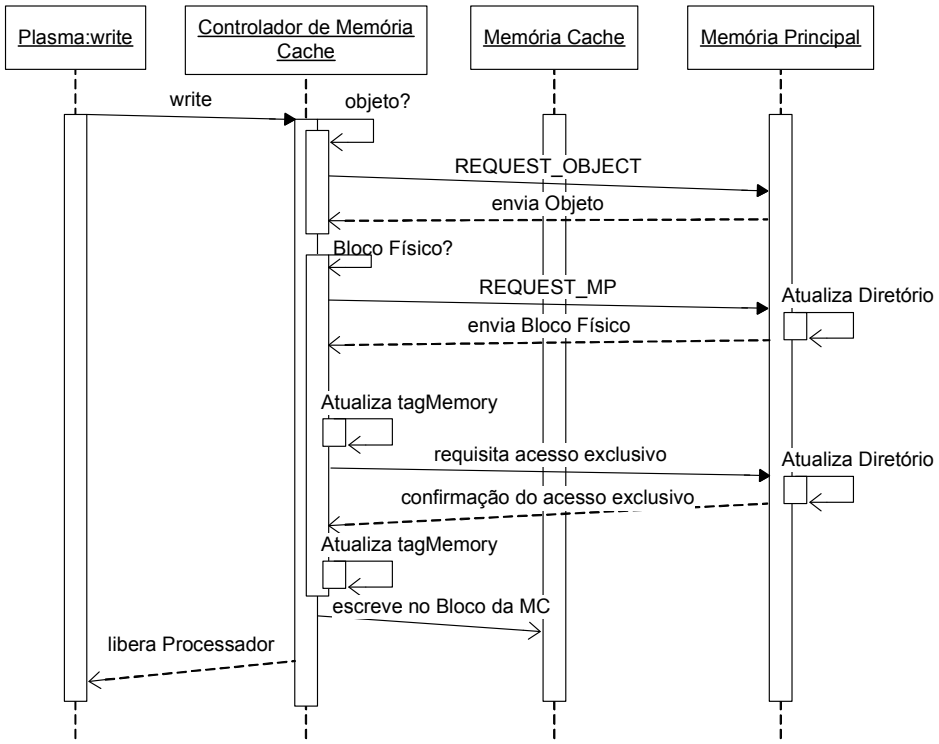

Figura 35 – Diagrama de seqüência para um write miss sem objeto ter sido inicializado.

Tabela 4 – Memória diretório da MP demonstrando objeto em posse da CPU 3 e inválido na MP.

Objeto	Bloco Inicial	Bloco Final	Inicializado	Válido	CPU 1	CPU 2	CPU 3	CPU 4
0	3	3	1	1	0	0	0	0
1	**0**	**2**	**1**	**0**	**0**	**0**	**1**	**0**
.....								

As transições de estados, Figura 36, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G2: 1 clock;
- G2 – G3: 28 clocks;
- G3 – G4: 28 clocks;
- G4 – S1: 1 clock;
- S1 – S2: 514 clocks, quantidade relativamente grande se comparado a um cache hit devido a necessidade de se ler todo o bloco da MP através da NoC;
- S2 – W2: 1 clock;

- W2 – W3: 40 clocks;
- W3 – W1: 23 clocks;
- W1 – W: 5 clocks.

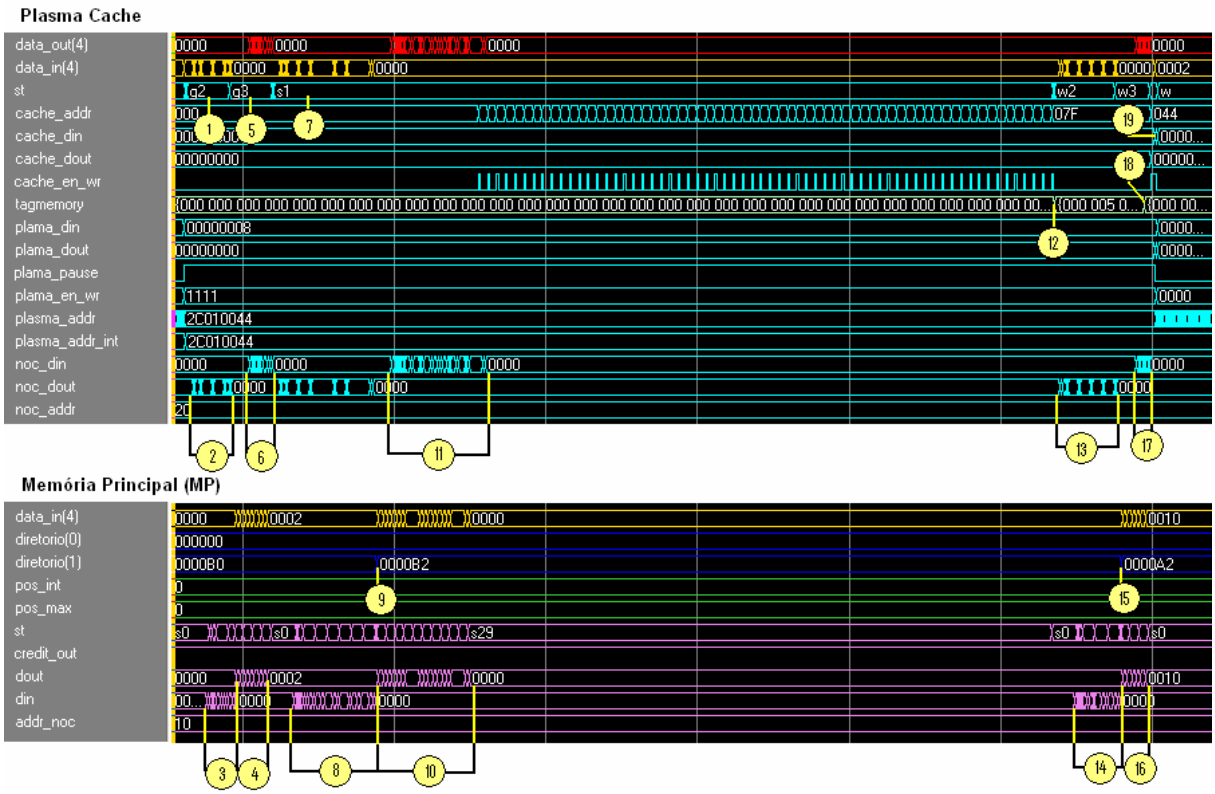

Figura 36 – Formas de onda para o teste de write miss com a MP já inicializada.

6.2 Read Miss Sem Objeto

O programa executa um read em um bloco ainda não presente na Memória Cache, obrigando o controlador de Memória Cache trazer os dados da MP.

- Ponto de verificação: o diretório deverá apresentar o campo válido com '1' (ou seja, não alterado) e a CPU *x* deve também ir a '1' indicando que o processador é dono de objeto.

Primeiramente este programa, referente ao teste 2, interage com a memória cache através de uma tentativa de leitura em um bloco do objeto 1:

- Configuração do objeto 1 não consta na memória cache (MC), o controlador de MC vai do estado W para o estado G1 a fim de preparar a requisição de configuração do objeto 1 à MP;
- Requisição é enviada durante o estado G2 (Figura 37 – 1);
- Requisição é recebida pela MP (Figura 37 – 2);
- MP envia os dados requisitados (Figura 37 – 3);
- Controlador de MC recebe os dados durante o estado G3 (Figura 37 – 4) e os armazena;

Após o controlador de MC receber os dados referentes à configuração do objeto 1, passa

para o estado G4, ao verificar a ausência do bloco físico referente ao objeto 1 na MC é necessário requisitar este conteúdo à MP:

- Controlador vai para o estado S1 (Figura 37 – 5);
- Requisita o Bloco Físico (Figura 37 – 6);
- MP recebe a requisição do Bloco (Figura 37 – 7);
- MP marca em sua memória diretório que a CPU 3 possui uma cópia do objeto (Figura 37 – 8) ao alterar o valor de 0xB0 para 0xB2 (Tabela 5);
- MP envia os dados referentes ao objeto para a CPU 3 (Figura 37 – 9);
- Controlador lê o Bloco Físico (Figura 37 – 10);
- Durante o estado S2 estes dados de configuração referentes ao objeto são gravados na **tag memory** do controlador de MC (Figura 37 – 11).

Por fim, durante o estado R1, se realiza a leitura solicitada pelo processador, que ao ser completada o controlador retorna ao seu estado inicial W indicando que a operação foi finalizada.

Tabela 5 – Memória diretório da MP demonstrando objeto em posse da CPU 3.

Objeto	Bloco Inicial	Bloco Final	Inicializado	Válido	CPU 1	CPU 2	CPU 3	CPU 4
0	3	3	1	1	0	0	0	0
1	**0**	**2**	**1**	**1**	**0**	**0**	**1**	**0**
.....								

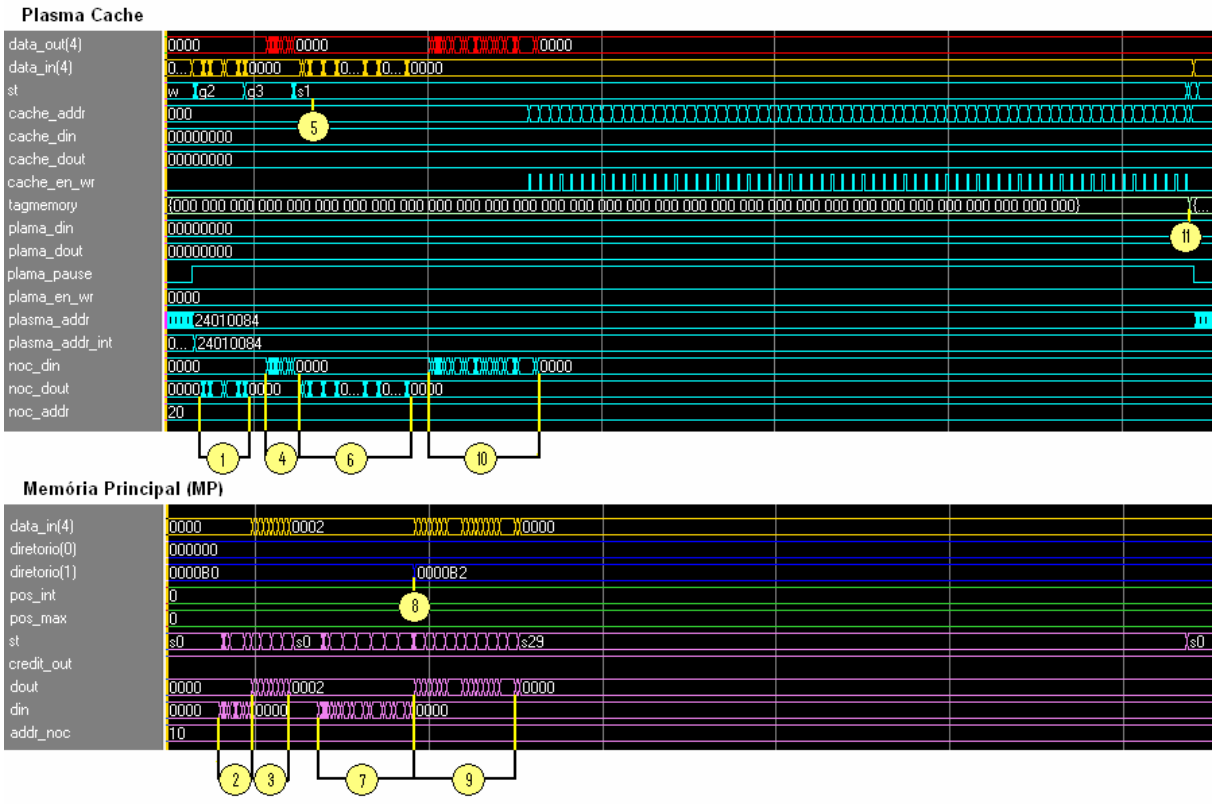

Figura 37 – Formas de onda para o teste de read miss com a MP já inicializada.

As transições de estados, Figura 37, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G2: 1 clock;
- G2 – G3: 28 clocks;
- G3 – G4: 28 clocks;
- G4 – S1: 1 clock;
- S1 – S2: 514 clocks;
- S2 – R1: 1 clock;
- R1 – W: 5 clocks.

6.3 Write Hit

Este teste serve somente para testar a correta escrita de um bloco na Memória Cache em estado exclusivo.

- Ponto de verificação: memória cache recebendo o dado correto para escrita.

Este programa de teste inicia sua interação com a memória cache através de uma tentativa de escrita em um Bloco Físico do objeto 1:

- Processador Plasma endereça a memória cache (Figura 38 – 1);
- Processador entra em pause (Figura 38 – 2);
- Como se trata de uma escrita, o processador libera em seu barramento de escrita o valor 9 (Figura 38 – 3);
- O controlador de memória cache inicia seu processamento ao entrar no estado G1 (Figura 38 – 4)
- O controlador de memória cache ao detectar que a configuração do objeto já está presente passa para o estado G4 (Figura 38 – 5);
- Com os dados sendo válidos e com a comparação de tag verdadeira, detecta-se o cache hit, passando para o estado W1 (Figura 38 – 6);
- Em W1 a memória cache recebe o valor a ser escrito (Figura 38 – 7); juntamente com uma habilitação (Figura 38 – 8); de escrita;
- Após os dados serem escritos, o processador é liberado para continuar seu processamento (Figura 38 – 9);
- Controlador de memória cache retorna ao seu estado inicial (Figura 38 – 10);

As transições de estados, Figura 38, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G4: 1 clock;
- G4 – W1: 1 clock;

- W1 – W: 5 clocks.

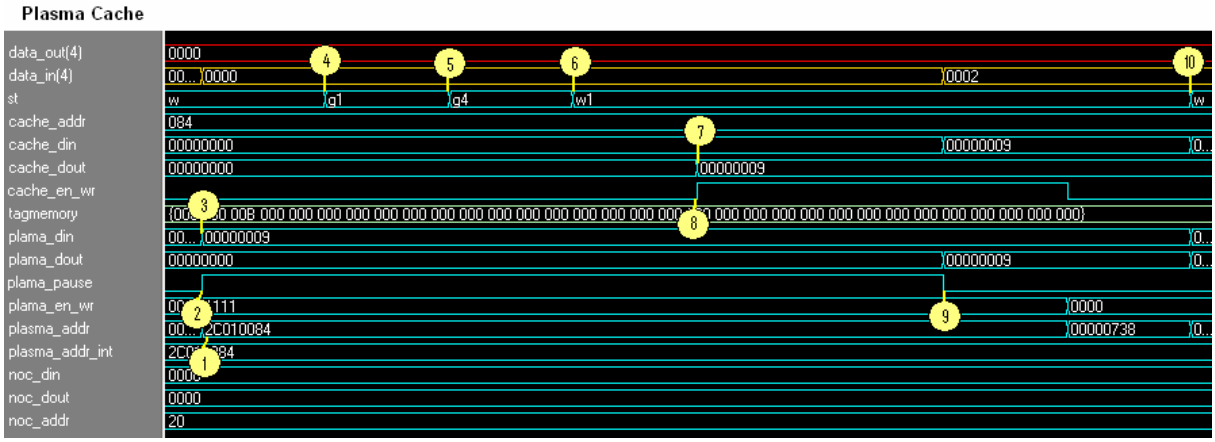

Figura 38 – Formas de onda para o teste de write hit.

Logo, o tempo de escrita em um bloco que já está em estado exclusivo consome apenas 8 ciclos de clock.

6.4 Read Hit

Este teste serve somente para testar a correta leitura de um bloco da Memória Cache em estado exclusivo.

- Ponto de verificação: memória cache enviando o dado correto em uma leitura.

Este programa de teste inicia sua interação com a memória cache através de uma tentativa de leitura em um Bloco Físico do objeto 1:

- Processador Plasma endereça a memória cache (Figura 39 – 1);
- Processador entra em pause (Figura 39 – 2);
- O controlador de memória cache inicia seu processamento ao entrar no estado G1 (Figura 39 – 3)
- O controlador de memória cache ao detectar que a configuração do objeto já está presente passa para o estado G4 (Figura 39 – 4);
- Com os dados sendo válidos e com a comparação de tag verdadeira, detecta-se o cache hit passando para o estado R1 (Figura 39 – 5);
- Em R1 a memória cache envia o valor lido (Figura 39 – 6);
- Após os dados serem lidos, o processador é liberado para continuar seu processamento (Figura 39 – 7);
- Controlador de memória cache retorna ao seu estado inicial (Figura 39 – 8);

As transições de estados, Figura 39, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G4: 1 clock;
- G4 – R1: 1 clock;

- R1 – W: 5 clocks.

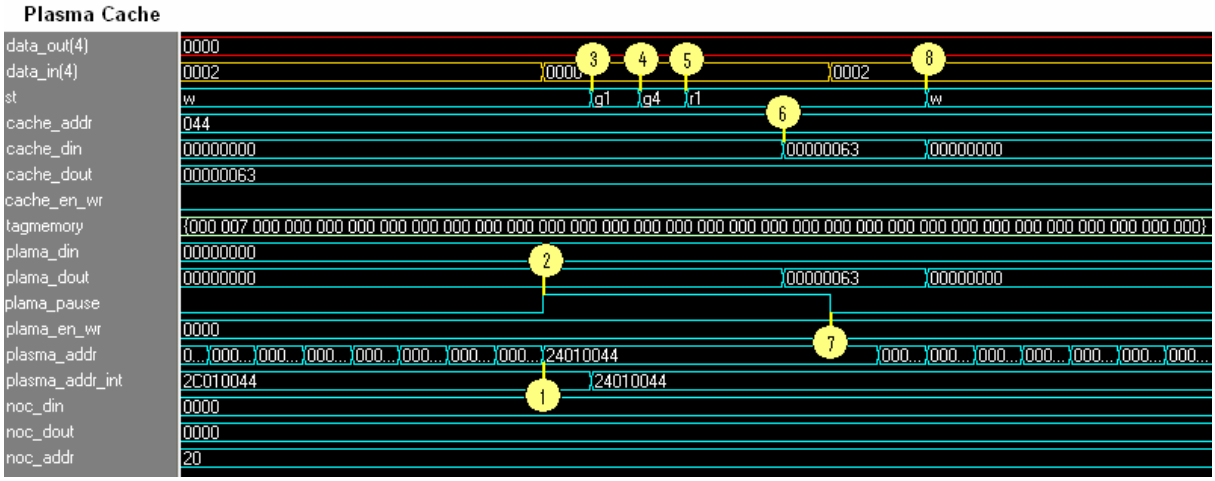

Figura 39 – Formas de onda para o teste de read hit.

Como o Bloco já estava presente na Memória Cache, 8 ciclos de clock foram consumidos para se efetuar a leitura deste dado.

6.5 Read Miss Com Objeto

Teste em que se deve verificar o correto funcionamento da máquina de estados do controlador de Memória Cache ao se tratar de uma ocorrência de read miss.

- Ponto de verificação: estados da máquina de controle: G1 → G4 → S1 → S2 → R1.

Programa inicia em uma tentativa de leitura em um objeto que já tenha sua configuração presente no controlador de MC, mas seu conteúdo não. O controlador de MC ao ter sua máquina de estados iniciada, durante o estado G1 verifica a presença da configuração do objeto a ser lido então passa para o estado G4:

- Durante o estado G4 é detectada a ausência do conteúdo do objeto na MC, então passa para o estado S1 (Figura 40 – 1) para ler o conteúdo do objeto da MP;
- Requisição dos dados é enviada para a MP (Figura 40 – 3);
- MP envia os dados de configuração (Figura 40 – 4);
- O controlador recebe os dados de configuração (Figura 40 – 5);
- Durante o estado S2 os dados de configuração referentes ao objeto são gravados na **tag memory** do controlador de MC (Figura 40 – 6);
- Os dados lidos são entregues ao processador durante o estado R1.

As transições de estados, Figura 40, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G4: 1 clock;
- G4 – S1: 1 clock;

- S1 – S2: 514 clocks;
- S2 – R1: 1 clock;
- R1 – W: 5 clocks.

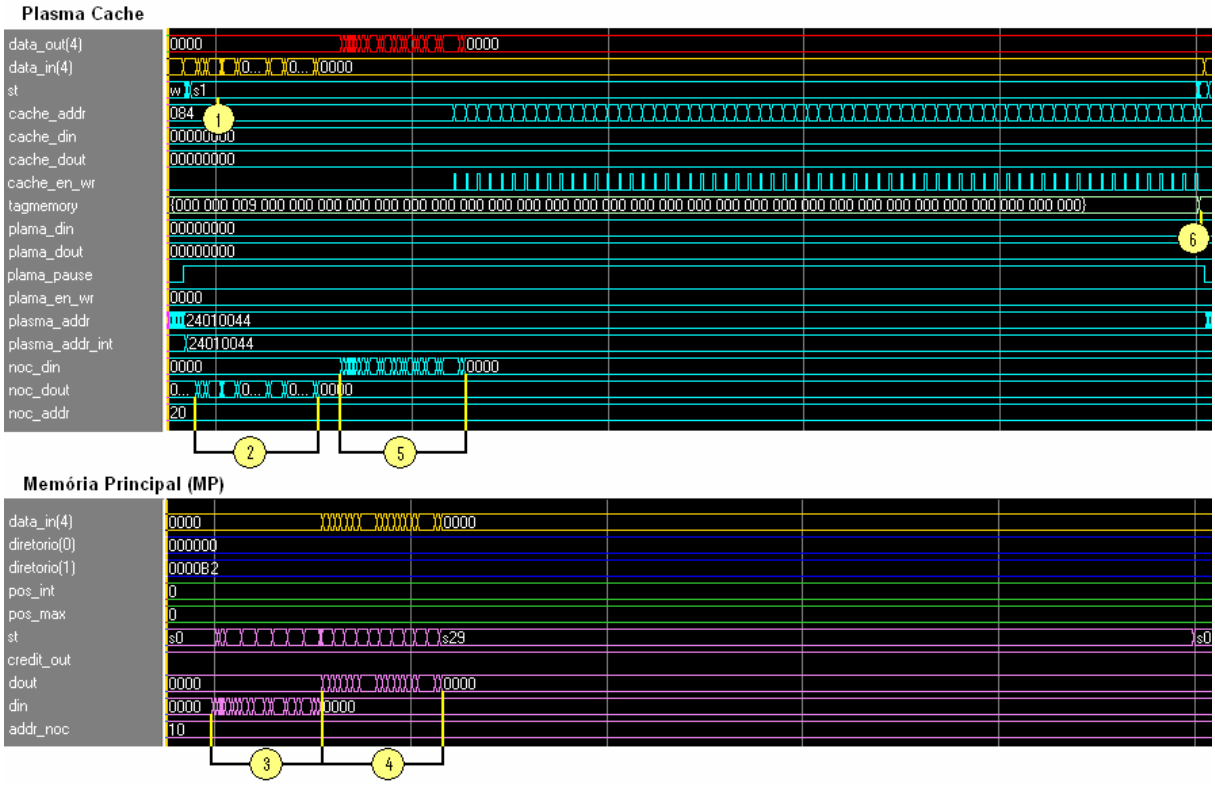

Figura 40 – Transições de estados durante um read miss.

6.6 Write Miss Com Objeto

Teste em que se deve verificar o correto funcionamento da máquina de estados do controlador de Memória Cache ao se tratar de um caso de write miss.

- Ponto de verificação: estados da máquina de controle: G1 → G4 → S1 → S2 → W2 → W3 → W1.

Programa inicia em uma tentativa de escrita em um objeto que já tenha sua configuração presente no controlador de MC, mas seu conteúdo não. O controlador de MC ao ter sua máquina de estados iniciada, durante o estado G1 verifica a presença da configuração do objeto a ser lido então passa para o estado G4:

- Durante o estado G4 é detectada a ausência do conteúdo do objeto na MC, então passa para o estado S1 (Figura 41 – 1);
- Requisição dos dados é enviada (Figura 41 – 2);
- MP recebe a requisição dos dados (Figura 41 – 3);
- MP marca na memória diretório (Figura 41 – 4) qual processador está requisitando os dados;
- Os dados são enviados (Figura 41 – 5);

- Controlador de MC recebe os dados (Figura 41 – 6);
- Durante o estado S2 estes dados são gravados no controlador de MC, na tag memory (Figura 41 – 7).

Para efetuar a escrita em um bloco deste objeto, é necessário requisitar acesso exclusivo ao objeto à MP:

- Durante o estado W2 é enviada a solicitação de acesso exclusivo ao objeto (Figura 41 – 8);
- MP recebe a requisição de acesso exclusivo (Figura 41 – 9);
- MP marca em sua memória diretório qual processador tem acesso exclusivo (Figura 41 – 10);
- MP envia a permissão (Figura 41 – 11) ao processador que a requisitou;
- O controlador de MC recebe a permissão durante o estado W3 (Figura 41 – 12);
- Controlador tem o valor exclusivo referente ao objeto alterado de '0' para '1' (Figura 41 – 13).

Com a configuração do objeto armazenado no controlador de MC, os dados referentes ao objeto armazenados na MC e a garantia de exclusividade sobre ele, é momento efetuar a operação de escrita na MC, ação de escrita é efetuada durante o estado W1, que ao ser completada o controlador retorna ao estado inicial W.

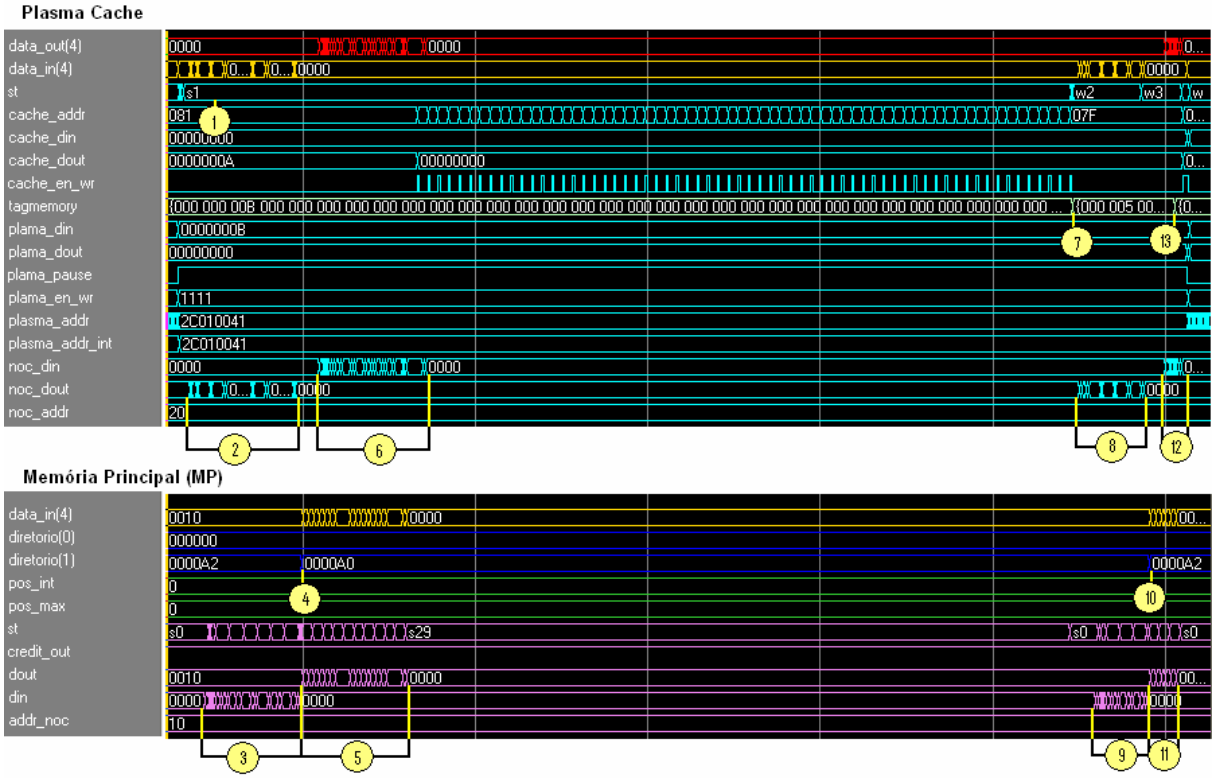

Figura 41 – Transições de estados durante um write miss.

As transições de estados, Figura 41, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G4: 1 clock;

- G4 – S1: 1 clock;
- S1 – S2: 515 clocks, quantidade relativamente grande se comparado a um cache hit devido a necessidade de se ler todo o bloco da MP através da NoC;
- S2 – W2: 1 clock;
- W2 – W3: 40 clocks;
- W3 – W1: 23 clocks;
- W1 – W: 5 clocks.

6.7 Seqüência de Write Hits

Programa de teste que ocasiona uma série de write hit na Memória Cache a fim de se testar esta função através de um teste de stress.

- Ponto de verificação: correta escrita dos dados na Memória Cache.

O programa de teste 7 foi construído com o uso da função write_local dentro uma estrutura de repetição a fim de se efetuar uma extensa escrita em determinadas posições da memória cache, através da Figura 42 pode-se visualizar uma pequena amostra destas escritas sobre uma posição da MC.

Write Hit no Bloco 1 do objeto 1:

- Estado de escrita (Figura 42 – 1);
- Endereço 68 da memória cache é informado (Figura 42 – 2);
- Valor 99 é disposto no barramento de escrita da memória cache (Figura 42 – 3);
- Sinais de escrita da memória cache são habilitados (Figura 42 – 4);
- Valor 99 é armazenado no endereço 68 da memória cache (Figura 42 – 5), endereço do Bloco 1 do objeto 1.

Write Hit no Bloco 3 do objeto 1:

- Estado de escrita (Figura 42 – 6);
- Endereço 196 da memória cache é informado (Figura 42 – 7);
- Valor 98 é disposto no barramento de escrita da memória cache (Figura 42 – 8);
- Sinais de escrita da memória cache são habilitados (Figura 42 – 9);
- Valor 98 é armazenado no endereço 196 da memória cache (Figura 42 – 10), endereço do Bloco 3 do objeto 1.

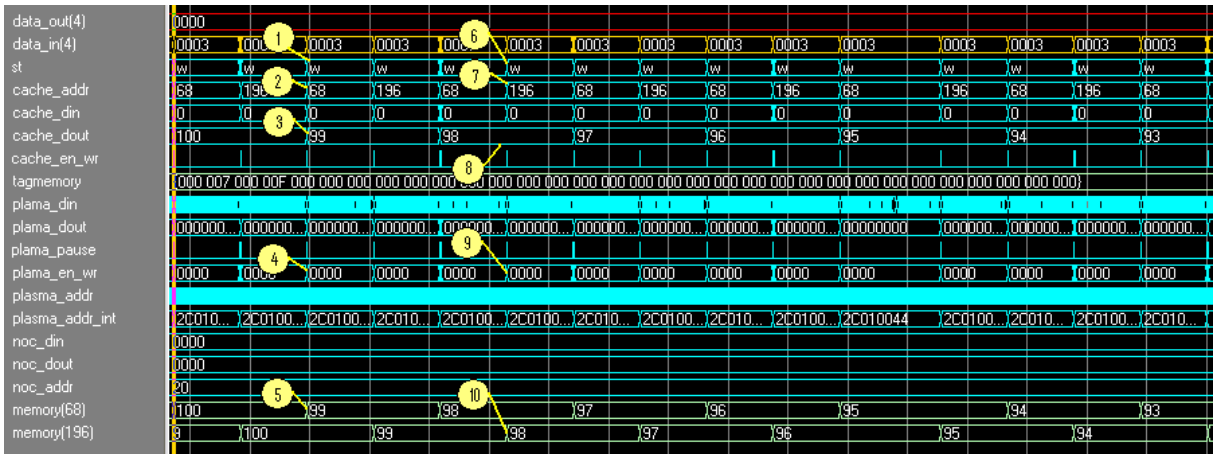

Figura 42 – Seqüência de Write Hits.

6.8 Seqüência de Read Hits

Programa de teste que ocasiona uma série de read hits na Memória Cache a fim de se testar esta função através de um teste de stress.

- Ponto de verificação: correta leitura dos dados na Memória Cache.

Construído com o uso da função read_local dentro uma estrutura de repetição a fim de se efetuar uma extensa leitura em determinadas posições da memória cache, através da Figura 43 pode-se visualizar uma amostra destas leituras sobre uma posição da MC.

Read Hit no Bloco 1 do objeto 1:

- Estado de leitura (Figura 43 – 1);
- Endereço 68 da memória cache é informado (Figura 43 – 2);
- Memória cache dispõe o valor 1 no barramento de leitura (Figura 43 – 3);
- Valor que foi disposto no barramento de leitura é o valor que está armazenado no endereço 68 da memória cache (Figura 43 – 4), endereço referente ao Bloco 1 do objeto 1.

Read Hit no Bloco 3 do objeto 1:

- Estado de leitura (Figura 43 – 5);
- Endereço 196 da memória cache é informado (Figura 43 –6);
- Memória cache dispõe o valor 3 no barramento de leitura (Figura 43 – 7);
- Valor que foi disposto no barramento de leitura é o valor que está armazenado no endereço 196 da memória cache (Figura 43 – 8), endereço referente ao Bloco 3 do objeto 1.

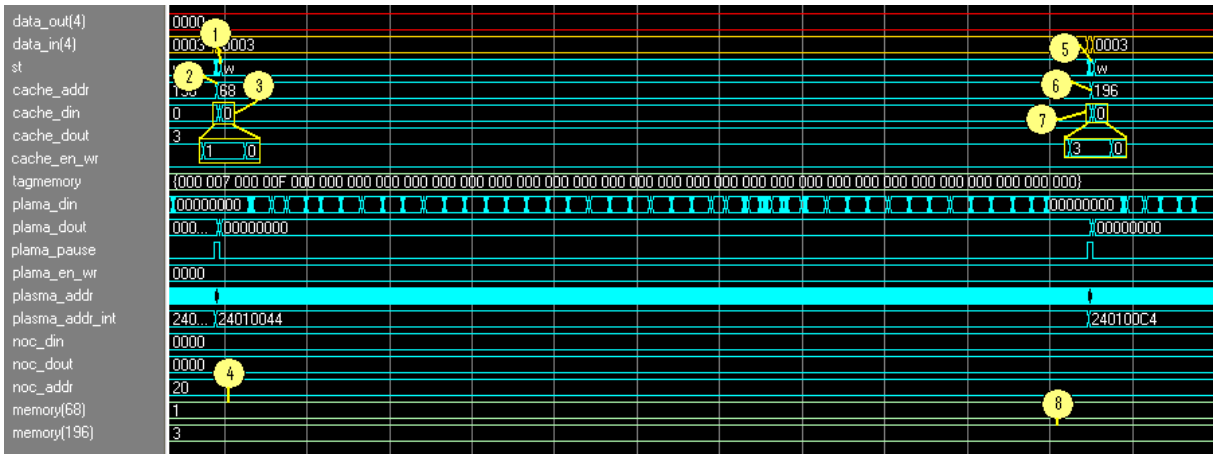

Figura 43 – Seqüência de Read Hits.

6.9 Recepção de um pedido de objeto alterado por parte da MP

Para se efetuar este teste, foram criados dois programas, cada um rodando em um processador diferente, em um determinado ponto o programa 2 solicita à MP um determinado objeto que esteja sob domínio do programa 1 (podendo ter sido alterado ou não), forçando o programa 1 enviar o bloco deste objeto à MP (caso este objeto tenha sido alterado) ou apenas responder com uma mensagem informando a MP que este objeto não foi alterado, a fim de que a MP passe a informação correta ao programa 2, que a solicitou.

- Ponto de verificação: o controlador de Memória Cache terá dois tipos de pacotes como opções de retorno: (1) pacote dizendo bloco não alterado, caso hit = '0' ou hit = '1' e exclusive = '0'; (2) pacote com o bloco alterado, neste caso com hit = '1' e exclusive = '1'.

Na execução deste teste, a CPU B, leu o objeto da MP, efetuou uma escrita sobre um de seus blocos que o invalidou na MP.

Após a CPU B ter invalidado este objeto na MP, o processo que estava rodando na CPU A efetuou uma leitura sobre o mesmo objeto:

- CPU A não contém uma cópia deste objeto, o solicita à MP (Figura 44 – 1 e Figura 44 – 3);
- MP possui este objeto em sua memória diretório marcado como inválido, solicita à CPU B que envie os dados alterados (Figura 44 – 5);
- MP recebe os dados válidos (Figura 44 – 8);
- MP envia os dados para a CPU A (Figura 44 – 11);
- O controlador de MC da CPU A recebe os dados (Figura 44 – 12).

Quando a CPU A recebe estes dados da MP, o controlador de MC passa para o estado S2 para gravá-los, e finalmente passa para o estado R1 para lê-los e enviá-los ao seu processador.

As transições de estados para um read miss com flush de dados, Figura 44, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G2: 1 clock;
- G2 – G3: 28 clocks;
- G3 – G4: 28 clocks;
- G4 – S1: 1 clock;
- S1 – S2: 1476 clocks, esta quantidade de consumo de ciclos de clock foi atingida devido ao fato de ter sido realizado um flush dos dados que estavam armazenados em uma Memória Cache de outro Processador Plasma;
- S2 – R1: 1 clock;
- R1 – W: 5 clocks.

As transições de estados para um write miss com flush de dados, consomem os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – G2: 1 clock;
- G2 – G3: 28 clocks;
- G3 – G4: 28 clocks;
- G4 – S1: 1 clock;
- S1 – S2: 1476 clocks, esta quantidade de consumo de ciclos de clock foi atingida devido ao fato de ter sido realizado um flush dos dados que estavam armazenados em uma Memória Cache de outro Processador Plasma;
- S2 – W2: 1 clock;
- W2 – W3: 40 clock;
- W3 – W1: 347 clock;
- W1 – W: 5 clocks.

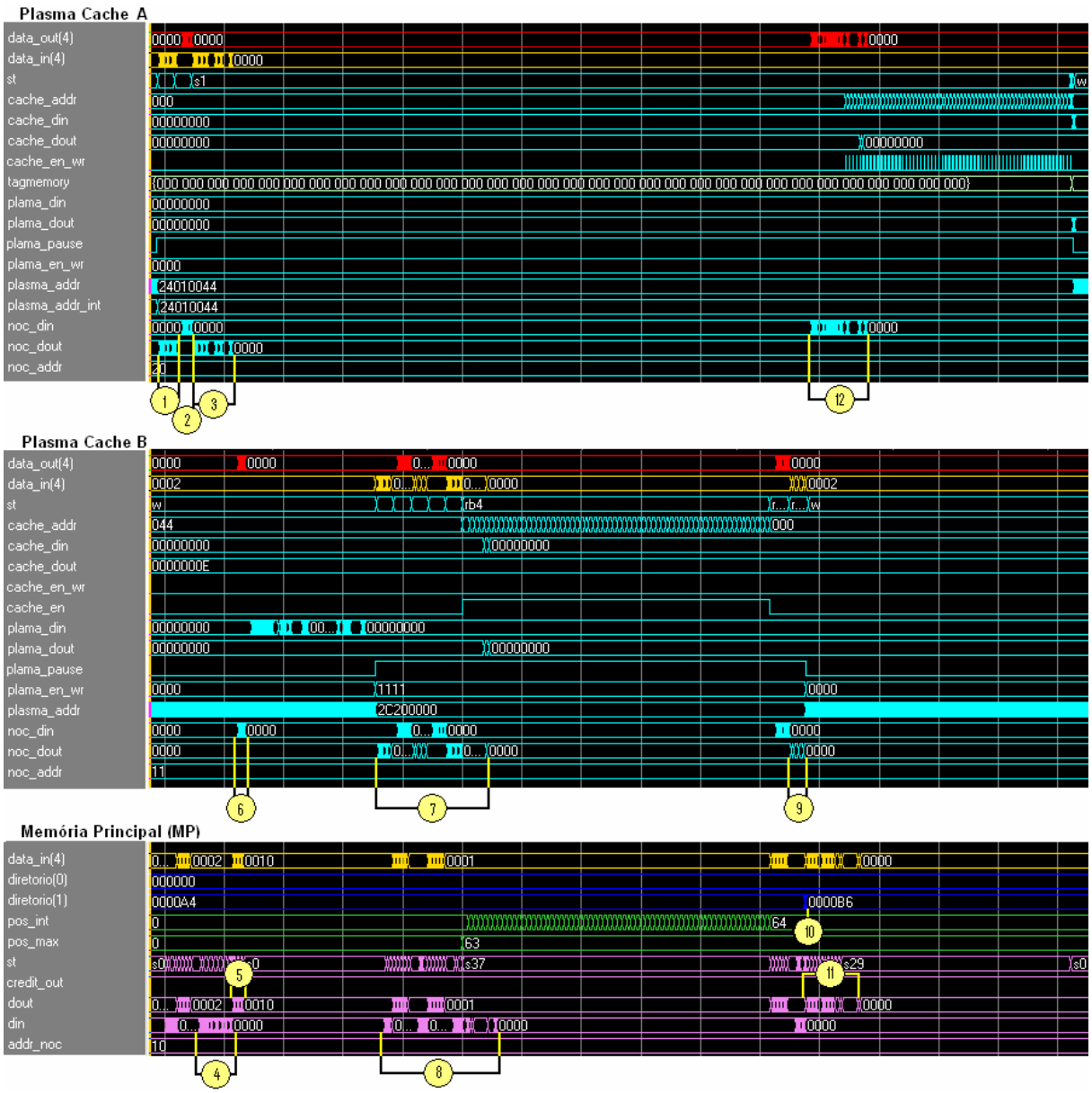

Figura 44 – CPU A recebendo um objeto alterado da CPU B.

6.10 Recepção de um pedido de invalidação de objeto por parte da MP

Para se efetuar este teste, foram criados dois programas, cada um rodando em um processador diferente, os dois programas contendo uma cópia do mesmo objeto em suas memórias locais (Memória Cache), em um determinado momento, um dos programas faz uma escrita em um dos objetos e faz a atualização deste objeto na MP, fazendo com que a MP envie sinal de invalidação para a cópia do mesmo objeto no outro processador.

- Ponto de verificação: o controlador de Memória Cache deve marcar o objeto como inválido através de uma mensagem recebida da MP.

O programa rodando na CPU B possui uma cópia do mesmo objeto que a CPU A que inicia

uma escrita sobre este objeto, como a CPU A não possui o bloco físico na qual deseja escrever referente a este objeto é iniciado o seguinte processo:

- Controlador de MC requisita o Bloco Físico à MP (Figura 45 – 1);
- CPU B que estava rodando seu processo teve de parar sua execução normal ao receber uma notificação da MP (Figura 45 – 2);
- CPU B inicia o processo de invalidação de sua cópia do objeto em que a CPU A está para efetuar uma escrita (Figura 45 – 3);
- Durante o processo de invalidação a CPU B se comunica com a MP a fim de informá-la que a invalidação está sendo processada (Figura 45 – 4, 5 e 6);
- MP informa à CPU A que ela pode continuar o processo de escrita (Figura 45 – 7);
- Processso de escrita é realizado quando o controlador de MC da CPU A passa do estado W3 para W1.

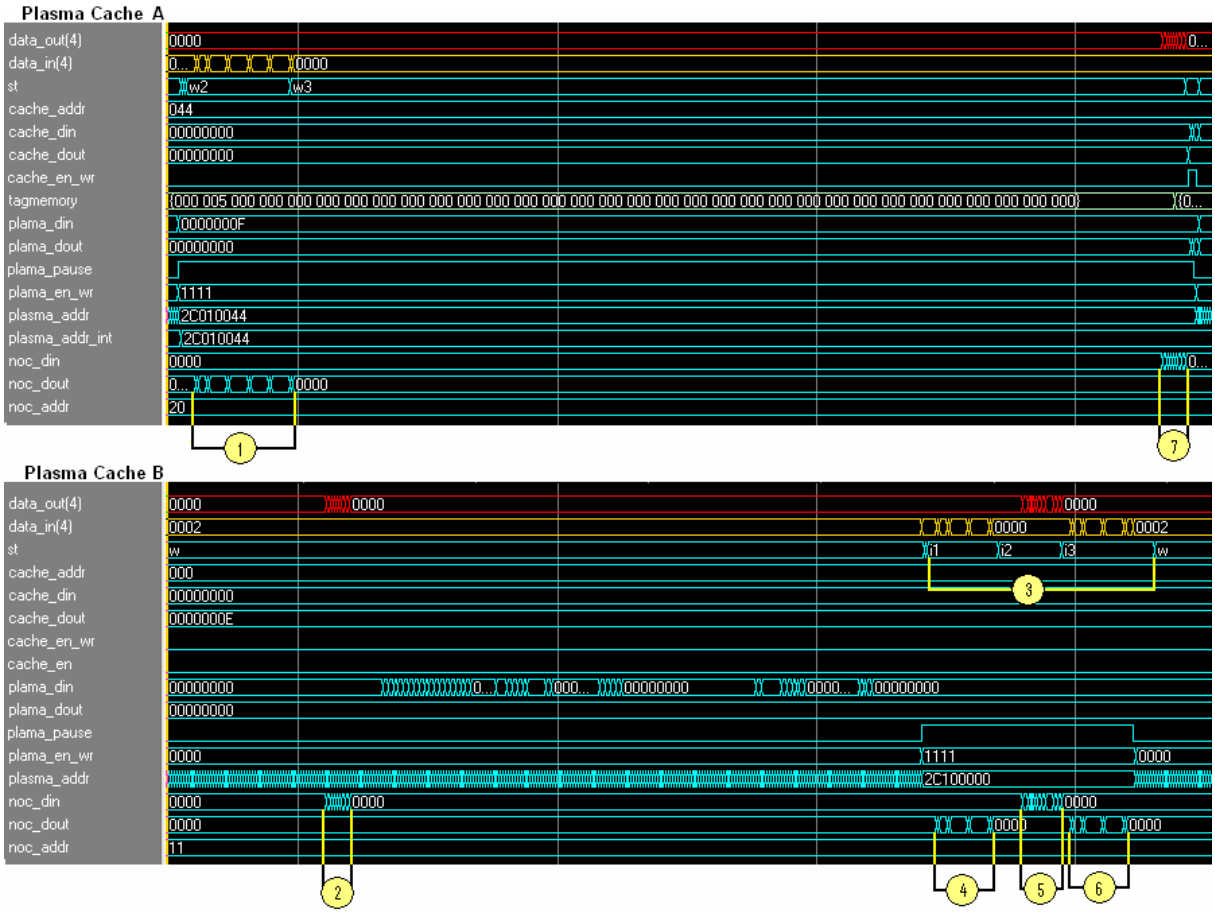

Figura 45 – Formas de onda referente ao teste 10.

As transições de estados, Figura 45, consumiram os seguintes ciclos de clocks:

- W – G1: 1 clock a partir do momento em que o controlador de memória cache é habilitado;
- G1 – I1: 1 clock;
- I1 – I2: 28 clocks;

- I2 – I3: 24 clocks;
- I3 – W: 36 clock;

Um total de 90 ciclos de clock foram consumidos pelo controlador de memória cache para que ele efetuasse a invalidação de um bloco após receber tal instrução através da NoC.

6.11 Teste de Crash

Programa de teste que provoca a máquina de estados do controlador de Memória Cache ir para um estado de erro.

- Ponto de verificação: máquina de estados do controlador de Memória Cache deve ir para o estado G7, estado de erro.

O programa de teste 11 iniciou com uma instrução de leitura de um objeto na qual a configuração deste ainda não estava presente no controlador de MC, como o endereço informado não pertencia ao objeto, o controlador detectou isto e foi para o estado de erro G7 (Figura 46 – 1) a fim de sinalizar este erro.

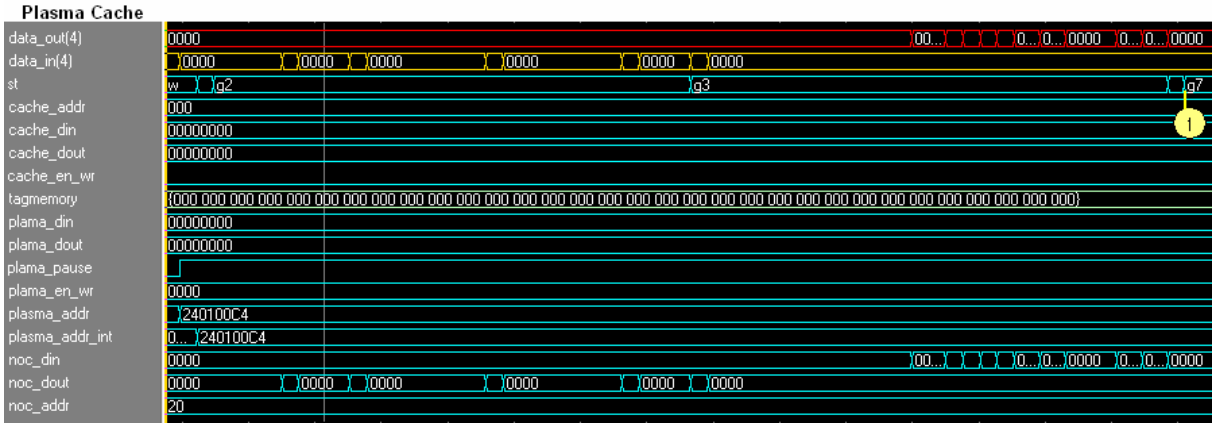

Figura 46 – Transições de estados para o controlador de memória cache na detecção de um erro.

6.12 Testes – Considerações finais

Após todos os testes realizados, uma base de informação sobre o funcionamento do sistema pôde ser adquirida. Testes que serviram para demonstrar o comportamento do controlador de memória cache ao ocorrer um cache hit ou um cache miss, e que também serviram para demonstrar como a Memória Principal reage diante do recebimento de diferentes requisições enviadas pelos processadores presentes na NoC.

Diante das informações referentes ao desempenho do sistema pôde ser feita uma comparação entre os tempos em que o processador Plasma tem que aguardar para ser executada uma ação requisitada ao controlador de memória cache.

Temos a informação de desempenho mensurada em ciclos de clock, podendo ser observada

na Tabela 6, onde:

- Write Miss sem Objeto: consumiu 642 ciclos de clock. Atingiu esta quantidade de ciclos porque durante este write miss é necessário que o controlador de memória cache leia informações referente ao objeto que está disposto na memória principal, para isso tem que passar pela NoC, o que adiciona um enorme atraso no processo;
- Write Miss com Objeto: consumiu 587 ciclos de clock. Atingiu esta quantidade de ciclos porque durante este write miss é necessário que o controlador de memória cache peça permissão à Memória Principal para efetuar a escrita sobre o objeto, para isso tem que passar pela NoC, o que adiciona um enorme atraso no processo;
- Write Miss com Flush de dados (Dirty): um total de 1928 ciclos de clock foram consumidos, pois ao requisitar as informações referentes ao objeto para à MP é necessário aguardar que o outro processador que possui os dados referentes ao objeto referenciado faça o flush de volta à MP. E somente após isto a MP irá enviar os dados para o processador que o requisitou. Contando ainda que existe o atraso gerado ao requisitar permissão de escrita sobre o objeto;
- Write hit: consome apenas 8 ciclos de clock, uma quantidade de ciclos bem inferior ao consumido em um write miss. Isto ocorre pelo fato de que o dado e sua configuração já está presente na memória local, não utiliza a NoC para requisitar informação alguma.
- Read Miss sem Objeto: consumiu 579 ciclos de clock devido à necessidade de ler um objeto da Memória Principal através da NoC;
- Read Miss com Objeto: consumiu 523 ciclos de clock. Atingiu esta quantidade de ciclos porque durante este read miss é necessário que o controlador de memória cache peça à Memória Principal o conteúdo do bloco do objeto referenciado, para isso tem que passar pela NoC, o que adiciona um enorme atraso no processo;
- Read Miss com Flush de dados (Dirty): um total de 1541 ciclos de clock foram consumidos, pois ao requisitar as informações referentes ao objeto para à MP é necessário aguardar que o outro processador que possui os dados referentes ao objeto referenciado faça o flush de volta à MP. E somente após isto a MP irá enviar os dados para o processador que o requisitou;
- Read hit: assim como o write hit, consome somente 8 ciclos de clock devido ao fato de não utilizar a NoC para ler o dado requisitado pelo processador Plasma.

Tabela 6 – Medidas de ciclos clock.

Caso	Ação	Ciclos de clock
1	Write Miss com Flush de dados	1928
2	Write Miss sem Objeto	642
3	Write Miss com Objeto	587
4	Write hit	8
5	Read Miss com Flush de dados	1541
6	Read Miss sem Objeto	579
7	Read Miss com Objeto	523
8	Read hit	8

A diferença de consumo de ciclos de clock entre um *Write Miss sem Objeto* e um *Write Hit* é de 634 ciclos de clock, ou seja, um *Write Hit* utiliza apenas 1,25 % dos ciclos de clock necessários em um *Write Miss sem Objeto*.

Ainda, o *Write Hit* utiliza 1,36% dos ciclos de clock de um *Write Miss com Objeto* e 0,41%

dos ciclos de clock de um *Write Miss com Flush de dados*.

Um *Read Miss* sem Objeto e um *Read Hit* tem a diferença de 571 ciclos de clock, um *Read Hit* usa somente 1,38% dos ciclos de clock utilizados em um *Read Miss sem Objeto*.

O *Read Miss* utiliza 1,53% dos ciclos de clock de um *Read Miss com Objeto* e 0,52% de um *Read Miss com Flush de dados*.

É notável a eficiência de um cache hit ao atender uma requisição de memória muito mais rapidamente do que um cache miss devido ao fato de não haver necessidade de trafegar dados através da NoC para comunicação com a MP.

As diferenças de tempo entre um cache hit e um cache miss podem ser melhores observadas na Figura 47.

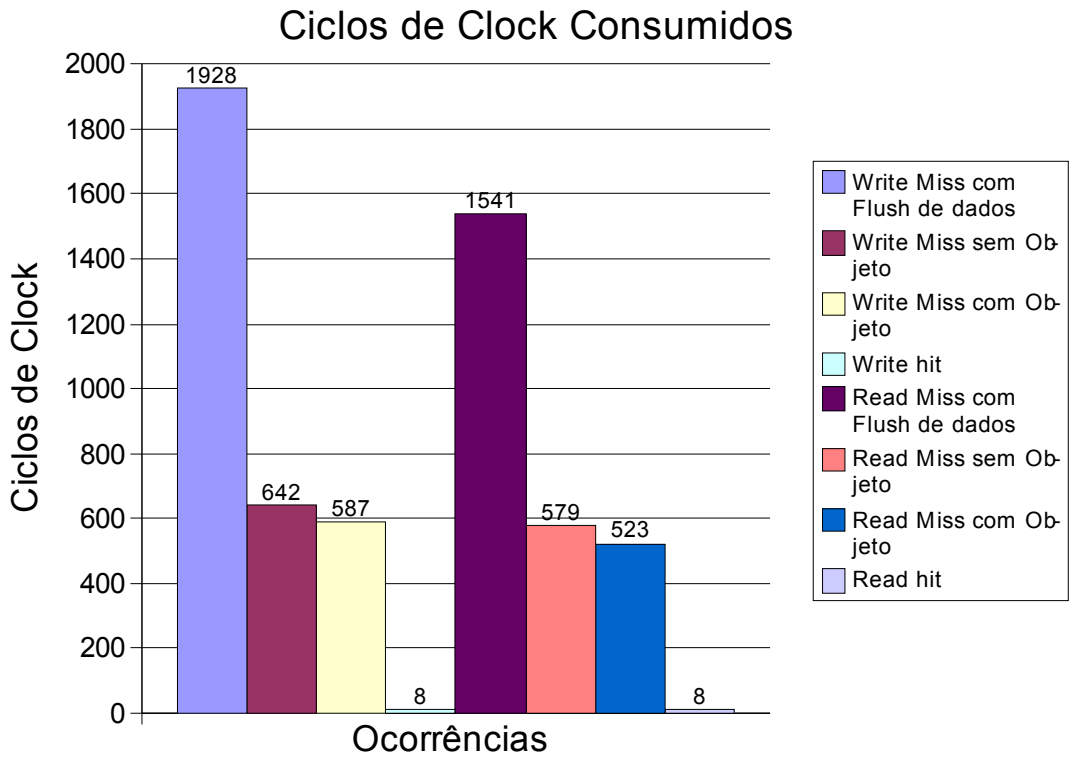

Figura 47 – Ciclos de clock consumidos.

Através destes dados referentes ao desempenho do sistema pode-se visualizar o impacto da política de acesso à memória cache e sua implementação sobre esta plataforma multiprocessada, a escolha de uma ou outra política irá interferir diretamente nos números dispostos nesta Seção.

7 Conclusões e Trabalhos Futuros

O presente Trabalho de Conclusão apresentou o estudo que foi realizado sobre diferentes políticas e técnicas para controle e acesso aos dados dispostos em um sistema de hierarquia de memória. Bem como o desenvolvimento de: memória compartilhada com acesso a uma rede de dados e capacidade de interpretação dos pacotes oriundos desta rede; memórias cache em um sistema multi-processado com políticas para controle e acesso aos dados; implementação de uma camada de software para se obter acesso às memórias, sendo ela local ou remota (compartilhada).

Este trabalho teve por característica a multi-disciplinaridade, requerendo dos autores a aplicação dos conteúdos de arquiteturas e organização de computadores, algoritmos de programação, sistemas operacionais e redes de computadores, assuntos abordados durante o curso de Engenharia de Computação.

A arquitetura desenvolvida adotou como método de projeto o reuso de módulos de propriedade intelectual, evitando-se assim o desenvolvimento de cada componente, permitindo a construção de um sistema de grande complexidade. O trabalho envolveu o desenvolvimento de software para os processadores Plasma, bem como todo o hardware envolvendo as memórias. Destaca-se como uma das dificuldades encontradas ao longo do trabalho o desenvolvimento do hardware, pois o mesmo causou uma complexidade enorme no desenvolvimento dos controladores, tanto da memória principal como da cache, por tratarem não só das escritas e leituras, mas também de coerência nas mesmas. Já no que se refere ao desenvolvimento do software, as maiores dificuldades encontradas foi no entendimento dos assemblys e de suas interfaces com a linguagem C.

Como característica desta nova arquitetura destaca-se a nova forma de comunicação entre os nodos plasma, agora podendo usar uma memória compartilhada para isso, e seu desempenho, já que poderá usar uma memória cache para aumentar o desempenho da comunicação. O sistema mostrou-se funcional, através de simulação VHDL.

A prototipação em FPGA é o primeiro trabalho futuro a ser executado. Outros trabalhos futuros que podem ser realizados a partir deste trabalho de conclusão são:

- Políticas de alocação e liberação de memória;
- Desfragmentação.

8 Referências Bibliográficas

[CHA90] Chaiken, D.; Fields, C.; Kurihara, K.; Agarwal, A. "Directory-based cache coherence in large-scale multiprocessors". Computer, vol. 23, issue 6, Jun 1990, pp.49-58.

[DAE06] Daewook, K.; Manho, K.; Sobelman, G. E. "DCOS: Cache embedded switch architecture for distributed shared memory multiprocessor SoCs". In ISCA '06, 2006, pp. 979-982.

[GAP06] GAPH – Hardware Design Support Group. Disponível em: http://www.inf.pucrs.br/~gaph.

[GOO83] Goodman, J. R. "Using cache memory to reduce processor-memory traffic". In ISCA '83: *Proceeding of the 10th annual international symposium on Computer Architecture*, 1983, pp. 124-131.

[GOO89] Goor, A. J. V. de. "Computer Architecture and Design". Addison-Wesley Publishers, 1989.

[HEN98] Hennessy, L.; Patterson, D. "Computer Organization and Design : the Hardware/Software Interface". San Francisco, CA: Morgan Kaufmann, 1998, pp. 896

[HEN03] Hennessy, J. L.; Patterson, A. "Arquitetura de Computadores uma Abordagem Quantitativa". Editora Campus, 2003.

[KAT85] Katz, R. H.; Eggers, S. J.; Wood, D. A.; Perkins, C. L.; Sheldon, R. G. "Implementing a cache consistency protocol". In ISCA '85: *Proceeding of the 12th annual international symposium on Computer Architecture*, 1985, pp. 276-283.

[LIL93] Lilja, D. J. "Cache coherence in large-scale shared-memory multiprocessors: issues and comparisons". ACM Comput. Surv., , vol. 25, issue 3, Set. 1993, pp.303-338.

[MOL05] MÖLLER, L. "Sistemas Dinamicamente Reconfiguráveis com Comunicação via Redes Intra-chip". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2005, 156 p.

[MON06] Monchiero, M.; Palermo, G.; Silvano, C.; Villa, O. "Efficient Synchronization for Embedded On-Chip Multiprocessors". *Very Large Scale Integration (VLSI) Systems*, vol.14, issue 10, Out. 2006, pp. 1049-1062.

[MOR04] Moraes, F. et al. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration the VLSI Journal, 38(1), October 2004. pp. 69-93.

[MOR07] Moraes, F. em http ://www.inf.pucrs.br/~moraes/, Junho 2007

[OPE06] OPENCORES. Capturado em http://www.opencores.org/, Julho 2006.

[PAP84] Papamarcos, M. S.; Patel, J. H. "A low-overhead coherence solution for multiprocessors with private cache memories". In ISCA '84: *Proceeding of the 11th annual international symposium on Computer Architecture*, 1984, pp. 348-354.

[PET06] Pétrot, F.; Greiner, A.; Gomez, P. "On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures". In DSD'06: *Conference on Digital System Design – 9th EUROMICRO*, 2006, pp. 53-60.

[PLA06] PLASMA processor. Capturado em: http://www.opencores.org/projects.cgi/web/mips/overview, Jul. 2006.

[STA03] Stallings, W. "Arquitetura e Organização de Computadores". Prentice Hall, 2003.

[TAE05] Taeweon, S.; Daehyun, K; Lee, H.-H. S. "Cache coherence support for non-shared bus architecture on heterogeneous MPSoCs". DAC, Jun. 2005, pp. 553-558.

[THA84] Thacker, C. "Private Communication". Jul.1984.

[WOS06] Woszezenki, C. “Alocação de Tarefas e Comunicação Entre Tarefas em MPSoCs”. Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, Dez. 2006.