RAFAEL MENDONÇA SOARES

IMPLEMENTAÇÃO DE UM MULTIPROCESSADOR SIMÉTRICO COM CACHES COERENTES

Trabalho de Graduação apresentado ao curso de Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial para a conclusão do curso.

Orientador: Prof. Dr. Roberto André Hexsel.

CURITIBA
2016

Resumo

Este trabalho descreve a implementação dum modelo em VHDL de um multiprocessador com memória compartilhada. A implementação é baseada na arquitetura MIPS e utiliza-se de um protocolo de coerência de caches baseado em espionagem. O núcleo base da implementação chame-se cMIPS - um modelo sintetizável em VHDL do modelo MIPS de cinco estágios que contém a implementação de todas as instruções do conjunto MIPS32r2. A comunicação das caches com outros periféricos ocorre através de barramento atômico compartilhado, onde os acessos são controlados por arbitragem centralizada. Cada núcleo é ligado diretamente a uma memória de instruções. O texto discute o controle necessário ao circuito de dados da cache para a implementação do protocolo de coerência, a interligação entre os núcleos e os periféricos, e as alterações no núcleo base do sistema para acomodar a implementação *multicore*.

A implementação foi validada através de simulações no software *ModelSim* e *GTKWave*. O trabalho apresenta os resultados preliminares da síntese do multiprocessador num FPGA.

Sumário

Capítulo 1

Introdução

Aumentar desempenho através de paralelismo em nível de *thread* passou a ser uma aposta das grandes indústrias a partir do início da década de 2000. Esta técnica surgiu como uma solução para a barreira térmica que inviabilizou o ganho de desempenho através do aumento da frequência de operação de um único processador [2]. Correntemente, são oferecidos no mercado múltiplos processadores, de menor complexidade, em único circuito integrado, forçando o desenvolvimento de versões paralelas de programas. Estas versões vêm mostrando bons resultado mas também encontram algumas barreiras que impedem o desenvolvimento de sistemas computacionais mais rápidos [5].

Um destes problemas emerge da necessidade de coordenação de processos concorrentes e faz com que a execução seja serializada em alguns momentos. Da Lei de Amdahl sabemos que o ganho de desempenho de um programa paralelo é limitado pela fração de tempo de execução serial. Outro problema em explorar paralelismo está na grande dificuldade em escrever programas paralelos. Uma consequência dos modelos de programação concorrente largamente utilizados, que tornam muito difícil garantir a corretude de versões paralelas de programas [5].

Processos executando em processadores autônomos precisam, de tempos em tempos, se comunicar. Um modelo para a comunicação entre processos consiste em uma única memória compartilhada entre todos processadores do sistema. Esse modelo tem o custo de garantir que todos processadores tenham a mesma visão da memória e é o foco deste trabalho.

O ambiente inicial do problema a ser resolvido com este projeto é um uniprocessador MIPS clássico de cinco estágios - cMIPS, que é um modelo sintetizável em VHDL e implementa o conjunto de instruções MIPS32r2 de forma similar ao projeto descrito em [7]. O caminho de dados tem todas as travas e adiantamentos necessários para a execução correta e eficiente de

código compilado em C. O projeto também conta um ambiente de simulação consistindo de memória RAM e ROM e dispositivos de entrada e saída. Este ambiente pode ser executado em simuladores ou dispositivos lógicos programáveis.

Este trabalho objetiva replicar e estender o projeto e a implementação de um multiprocessador baseado no cMIPS. O objetivo final é obter um modelo VHDL sintezável de um multiprocessador com memória compartilhada que proporcione uma maior capacidade de processamento obtida pela integração de vários processadores simples. Para atingir este objetivo o projeto implementa uma cache de dados ligada a cada núcleo cMIPS que é mantida coerente através do protocolo MESI. A comunicação das caches com outros periféricos ocorre através de barramento atômico compartilhado, onde os acessos são controlados por arbitragem centralizada. Cada núcleo é ligado diretamente a uma memória de instruções. As simulações de corretude funcional do modelo foram feitas com *ModelSim* e *GTKWave*.

O texto está organizado como segue. No Capítulo 2 são expostos os conceitos envolvidos no desenvolvimento do multiprocessador. No Capítulo 3 são apresentados os aspectos de implementação do multiprocessador. O ambiente de testes é apresentado no Capítulo 4 e o Capítulo 5 sintetiza os resultados e discute as possibilidades de trabalhos futuros.

Capítulo 2

Componentes do Sistema

Este Capítulo apresenta os conceitos usados para a proposta do cMIPS multicore. A Seção 2.1 descreve a implementação do core básico do sistema. A Seção 2.2 descreve os problemas de coerência ao considerar a implementação de um multiprocessador com caches privadas e a Seção 2.3 descreve o protocolo usado para manter as caches coerentes.

2.1 cMIPS

O cMIPS é um modelo em VHDL do processador de cinco estágios que se aproxima da descrição apresentada em [7]. A ideia de implementar o processador surgiu da necessidade de prática nas disciplinas de arquitetura de computadores do departamento de informática da UFPR, na qual o livro [7] é usado como texto base da disciplina [3]. O modelo contém a implementação de todas as instruções do conjunto MIPS32r2.

Durante o estágio *Instruction Fetch* (IF) as instruções são buscadas na memória ROM ou cache. Estas interfaces podem sinalizar ao núcleo a necessidade de ciclos adicionais para concluir a busca da instrução. No estágio *Instruction Decode and Register Fetch* (RF) acontece a decodificação da instrução lida e leitura dos registradores. O processador conta com travas e adiantamentos necessários resolver instruções de desvios neste estágio. O estágio *Execution* (EX) realiza os cálculos de endereço ou operações aritméticas e contém todas travas e adiantamentos necessários para resolver possíveis riscos de dados. A leitura ou escrita da memória de dados acontece no estágio *Memory* (MM), e neste estágio ocorre o acesso a uma cache de dados coerente. Por fim, o registrador de destino é escrito durante estágio *Write Back* (WB).

O modelo também conta com um coprocessador de controle que implementa seis interrupções de hardware, duas interrupções de software e interrupções não-mascaráveis. O modelo também conta com uma unidade de gerenciamento de memória (MMU) para permitir realocação da memória principal.

O suporte para acessos atômicos é feito através das instruções *Load-Linked* (LL) e *Store-conditional* (SC). A instrução LL lê um valor da memória, escreve o endereço da referência no registrador $LLAddr$ e atribui o valor 1 para o registrador $LLBit$. A instrução SC escreve um valor na memória e retorna indicação de sucesso se o valor registrador $LLBit$ é um, caso contrário, retorna indicação de falha (0) e não modifica a memória. O modelo do cMIPS suporta invalidações de instruções SC na ocorrência de exceções.

Além da implementação do núcleo, o projeto contém um ambiente de simulação como mostrado na Figura 2.1. Este ambiente consiste de caches de instruções e de dados, memórias RAM e ROM e periféricos de entrada e saída. As memórias ROM e RAM possuem versões de simulação e síntese, e são inicializadas a partir de arquivos gerados por *scripts* de compilacão. A versão de simulação contém periféricos de entrada/saída de arquivos e saída padrão, unidade de ponto flutuante e um temporizador de interrupções. A versão de síntese suporta a escrita de dados em um *display* LCD, a leitura de dados através de um *keypad* e uma interface serial (UART).

Os periféricos realizam comunicação com processador através de um barramento compartilhado. No início do estágio de memória (MM), o endereço gerado pelo núcleo é propagado para a memória principal, unidades de entrada/saída e decodificadores de endereço (representado pela entidade *sel* no diagrama). O barramento é implementado por meio de multiplexadores que são selecionados pelo decodificador *sel*.

2.2 O problema da coerência de caches

A proposta deste trabalho é colocar vários núcleos cMIPS para trabalhar com o requisito de que eles tenham uma mesma visão da memória. Uma solução para este problema poderia consistir em processadores que acessam diretamente a memória principal, sem passar por cache alguma. Neste caso, a memória impõem uma ordem serial em todas operações de leitura e escrita à memória, e todos os processadores tem a mesma visão da memória. Por razões de desempenho é desejável colocar uma cache privada em cada processador fazendo com que múltiplas cópias do

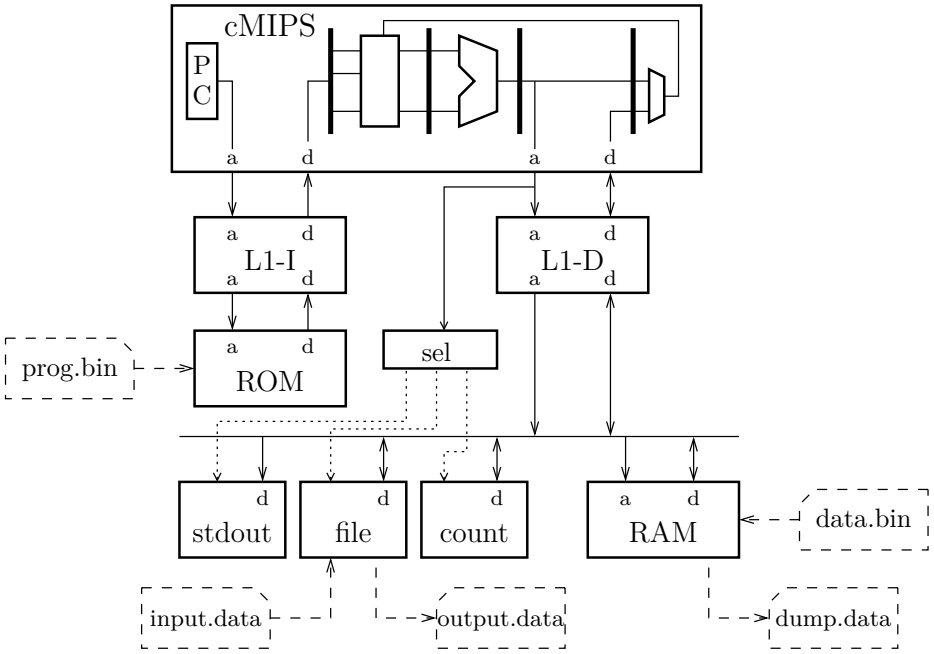

Figura 2.1: Barramento cMIPS

mesmo dado possam simultaneamente existir em diferentes caches. Vamos supor agora que um processador tem uma cópia local de um bloco e um outro processador do sistema deseja escrever em sua cópia local deste mesmo bloco. Sem nenhum cuidado adicional, os dois processadores podem ter visões diferentes da memória. Este problema é conhecido como o problema de coerência de cache e, de uma maneira informal, é desejável garantir que uma operação de leitura retorna o último valor escrito naquele endereço [2].

Formalmente um sistema de memória é coerente se a seguintes condições são atendidas [2]:

- Uma leitura por um processador P em uma posição X seguido de um gravação por P em X, sem a ocorrência de gravações em X por outro processador neste intervalo, sempre retorna o valor gravado por P.

- Uma leitura por P1 na posição X após uma gravação por P2 em X retorna o valor gravado se a leitura e a gravação estiverem separadas no tempo e não ocorrer nenhuma outra gravação em X entre os dois acessos.

- Gravações na mesma posição são serializadas: duas gravações na mesma posição por dois processadores quaisquer são vistas na mesma ordem por todos os processadores.

2.3 Protocolo MESI

Um mecanismo em hardware para manter o sistema de memória coerente foi proposto em [6] e é chamado de protocolo de coerência MESI. A solução é baseada no método de espionagem com invalidação e pressupõe uma cache privada para cada processador. Os controladores de cache observam todo o tráfego no barramento e, se uma cache verifica uma operação de escrita a um bloco na qual ela tem uma cópia, então invalida esta cópia local. De uma forma simplificada, na ocorrência de uma escrita em uma das caches, as demais cópias do sistema são invalidadas. As ações de um controlador são visíveis pelos demais controladores através da difusão em um barramento ou alguma outra forma de *broadcast*. A solução descrita em [6] pressupõe caches com políticas de escrita preguiçosa (*write-back*).

A implementação do protocolo baseia-se na associação de estados a cada bloco da cache, com o estado implementado por meio de dois bits: o primeiro indica se alguma cache do sistema possui cópia do mesmo bloco mas este não foi modificado (está consistente com a memória); o segundo indica se o bloco foi modificado localmente, se o valor do bloco está diferente do valor do bloco na memória principal [6]. A combinação destas duas condições leva ao seguinte conjunto de estados:

- Não compartilhado e não modificado (EXCLUSIVO): um bloco neste estado é a única cópia válida e está consistente com a memória principal;
- Não compartilhado e modificado (MODIFICADO): nenhuma outra cache detém cópia do bloco mas a cache realizou modificações locais no bloco, fazendo com que a memória principal mantenha um valor obsoleto;
- Compartilhado e não modificado (COMPARTILHADO): o bloco está consistente com a memória mas outras caches podem ter cópias do mesmo bloco; e
- Compartilhado e modificado (INVALIDO): no protocolo proposto em [6] este estado indica que o bloco está inválido na cache, e não pode ser usado.

Um bloco somente é escrito na memória principal em caso de substituição de um bloco no estado MODIFICADO. O estado COMPARTILHADO não implica em mais de um cópia válida no sistema, tendo em vista que um bloco COMPARTILHADO pode ter sido previamente substituído. Portanto, indica que em algum momento da execução duas cópias do mesmo bloco estavam presentes.

No protocolo MESI o controlador precisa ser capaz de responder às requisições do processador e do barramento. Uma forma de dividir essa tarefa é pensar na cache contendo dois controladores que cooperam entre si para manter a coerência do sistema. O "controlador do lado do processador" atende às requisições do processador e inicia uma transação no barramento, atribuindo o devido estado ao bloco requisitado pelo processador. Esta transação colocada no barramento é monitorada por todos controladores de espionagem que modificam, se necessário, o estado da sua cópia local do bloco [1]. Este controlador é chamado de "controlador de espionagem" ou "controlador do lado do barramento".

Usando caches com escrita preguiçosa, o valor mais recente de um dado pode estar em alguma das caches do sistema. Portanto, se uma cache deseja um ler um dado, ela deve difundir a requisição aos controladores que monitoram o barramento. Estes, por consequência, devem informar a cache faltosa se possuem uma cópia válida do bloco. Se alguma cache responde positivamente, então a cache faltosa deve ler os dados da cache remota e inibir a memória principal de fornecer os dados no barramento. Em suma, se alguma cache retém cópia do bloco, então ela deve colocar os dados no barramento, caso contrário, a memória principal responde as requisições do barramento [1].

Com estas definições, é possível apresentar o algoritmo de coerência proposto em [6]. O Algoritmo 1 mostra a sequência de passos para o controlador da cache que atende requisições do processador. O Algoritmo 2 é executado pelo controlador de espionagem. A operação "lê bloco" significa ler da memória principal ou de uma cache remota.

A principal vantagem deste protocolo é a otimização do caso em que um núcleo lê um dado e imediatamente depois o escreve. Considerando um protocolo com somente os estados MODIFICADO, COMPARTILHADO e INVALIDO, em que um bloco lido é colocado no estado COMPARTILHADO [8], uma escrita subsequente necessita de uma transação no barramento para invalidar outras cópias do bloco, mesmo se nenhuma cache contenha o bloco. Desse modo, no protocolo MESI, quando um bloco é lido e nenhuma outra cache tem cópia do bloco, então o estado EXCLUSIVO é atribuído ao mesmo. Como resultado, um subsequente escrita não provoca uma transação no barramento para invalidar outras cópias, evitando tráfego desnecessário no barramento [8, 6].

```
se requisição do processador é de leitura então
    se falta de leitura então
        se bloco.estado = MODIFICADO então
            write-back;
        fim
        difunde requisição do bloco no barramento e lê bloco;
        se bloco veio da memória principal então
            bloco.estado ← EXLUSIVO;
        senão
            bloco.estado ← COMPARTILHADO;
        fim
    senão
        lê da cache;
    fim
senão
    se falta de escrita então
        se bloco.estado = MODIFICADO então
            write-back;
        fim
        lê bloco;
        invalida outras cópias;
    senão
        se bloco.estado = COMPARTILHADO então
            invalida outras cópias;
        fim
    fim
    bloco.estado ← MODIFICADO;
    escreve na cache;
fim
```

Algoritmo 1: Algoritmo do controlador do lado do processador

```
se cache local tem bloco requisitado por cache remota então
    se leitura então
        se bloco.estado = MODIFICADO então
            write back;
        fim
        coloca bloco no barramento;
        bloco.estado ← COMPARTILHADO;
    senão
        // escrita ou invalidação
        bloco.estado ← INVALIDO;
    fim
fim
```

Algoritmo 2: Algoritmo do controlador de espionagem

Capítulo 3

Implementação

Um modelo da implementação do cMIPS multicore é mostrado na Figura 3.1. A arquitetura proposta consiste de um ou mais núcleos interligados através de um barramento compartilhado. O número de núcleos é configurável através de *generics* no código VHDL. Cada *core* cMIPS é conectado a uma memória de instruções (ROM) e a uma cache de dados. O controlador da cache de dados é sempre o iniciador de qualquer transação no barramento, que é atendida por uma cache remota, memória principal ou dispositivos de entrada e saída. Os periféricos de entrada e saída são mapeados como memória.

A permissão de utilização do barramento é controlada pela unidade "árbitro do barramento" através do algoritmo *round-robin*. A unidade "árbitro de espionagem" é responsável por decidir, dado o tipo da transação ocorrendo no barramento, que unidade escreve no barramento de dados.

A Seção 3.1 descreve as interfaces utilizadas no sistema bem como a atribuição dos endereços aos periféricos. A Seção 3.2 descreve o funcionamento do barramento e do processo de arbitragem. A Seção 3.3 descreve as unidades básicas da cache de dados e os tipos de transações que elas geram no barramento. A Seção 3.4 descreve o circuito de dados da cache de dados. A Seção 3.5 descreve o controle necessário ao circuito de dados da cache. A Seção 3.6 descreve os relógios empregados no sistema. Por fim, a Seção 3.7 descreve as modificações feitas no núcleo cMIPS para a implementação das primitivas de sincronização.

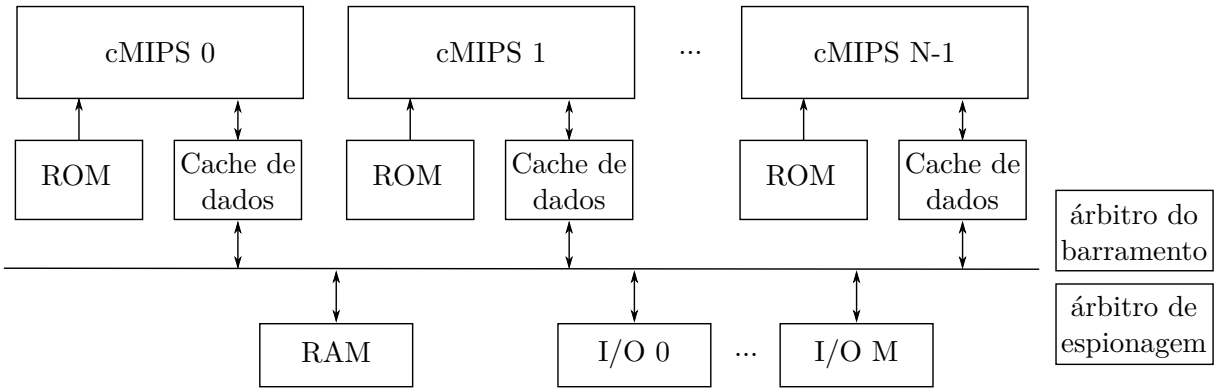

Figura 3.1: Arquitetura proposta do cMIPS multicore

3.1 Interface e controladores de periféricos

A interface da memória principal (RAM), memória de instruções (ROM), caches de dados e dispositivos de entrada e saída seguem o mesmo padrão de acesso. A temporização de acessos a essas unidades é controlada pelos sinais *sel* e *rdy*. Uma unidade que deseja transferir algum dado ativa o sinal *sel* para sinalizar que o endereço e dados de saída estão estáveis e válidos. Na próxima borda do relógio, a unidade de destino captura os dados. O sinal *rdy* deve ser ativado para que o processador espere pelo dispositivo endereçado. A Figura 3.2 mostra um acesso de escrita de quatro palavras utilizando a temporização de acesso implementada. Os sinais *sel* e *wr* são ativos em zero.

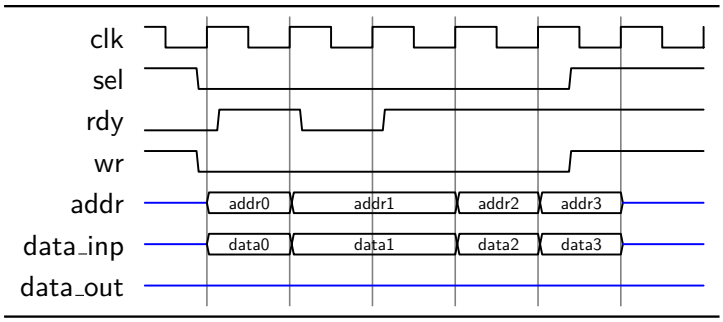

Figura 3.2: Diagrama de tempo de uma escrita de 4 palavras

A uniformidade nos acessos às unidades acopladas ao barramento é desejada pois o protocolo MESI permite transferência de blocos de cache para cache. Quando o controlador da cache está realizando uma operação de leitura no barramento, este pode estar lendo o bloco da memória principal ou de uma cache remota. Esse padrão simplifica o controle de acessos às interfaces ligadas ao barramento, visto que a origem do bloco é irrelevante para este objetivo. A informação da origem do bloco, no entanto, é essencial ao protocolo MESI quando consideramos o estado EXCLUSIVO.

A interface da cache de dados é composta pelos sinais descritos na Tabela 3.1 e pode ser subdividida em três interfaces: (i) interface com o processador, (ii) interface da cache com o barramento e (iii) interface de espionagem. A interface (i) recebe as referências de memória do núcleo cMIPS. Se existe um falta na cache, então a interface (ii) inicia uma transação no barramento. Essa transação é atendida por uma cache remota através da interface do tipo (iii), pela memória principal (iv) ou por um periférico (v).

As unidades (iv) e (v) respondem se possuem ou não o dado requisitado através de unidades de decodificação dos endereços do barramento. A seleção desses dispositivos é baseada no mapa de endereços mostrado na Tabela 3.2.

3.2 Barramento e árbitro

Os acessos à memória principal, às caches de dados e dispositivos de entrada e saída ocorrem através de um barramento compartilhado. Devido a impossibilidade do uso de *drivers tri-state* em FPGA's, o efeito de um barramento é atingindo com uso de multiplexadores e é, de fato, um *crossbar*. A cache de dados é a única iniciadora de requisições de leitura/escrita no barramento. O par requisição–resposta de um acesso às unidades do barramento é indivisível e o controlador da cache libera o barramento apenas quando conclui sua transação, e por isso é chamado de barramento atômico. O controle de acesso ao barramento de endereço, dados e comandos é feito por duas unidades chamadas "árbitro do barramento" e "árbitro de espionagem". A lógica de arbitragem do barramento demora um ciclo do relógio para ser concluída. A Figura 3.3 mostra a ligação entre os elementos do sistema.

O árbitro do barramento implementa a política de arbitragem *round-robin* para decidir qual unidade tem permissão de acesso ao barramento. Na borda de subida do relógio as caches sinalizam através do sinal `bus_req` uma requisição do barramento. O árbitro responde à requisição através do sinal `bus_gnt` e seleciona os multiplexadores de endereço, comando, e sinais controle *sel*, *wr* e *rdy*. O endereço e comando da requisição são propagados para as interfaces de espionagem das caches e decodificadores de endereço. Essas unidades reportam para o árbitro de espionagem se possuem o bloco requisitado.

O árbitro de espionagem é responsável por escolher qual unidade escreve no barramento de dados. Quando o controlador da cache realiza uma transação de escrita no barramento, então o próprio controlador escreve no barramento de dados. Em caso de uma operação de leitura, o

Nome	Descrição
rst	reset
clk	relógio da interface
phi2	relógio deslocado de 90 graus de clk
	interface com processador
cpu_sel_i	valores nas linhas de dados e endereço são válidos
cpu_rdy_o	dado requisitado à cache está pronto
cpu_wr_i	operação de escrita
cpu_addr_i	endereço da operação requisitada
cpu_data_i	dado a ser escrito
cpu_data_o	dado lido
cpu_xfer_i	granularidade da escrita (*byte, halfword, word*)
cpu_abort_ref_i	cache deve abortar referência corrente
	interface da cache
bus_sel_o	valores nas linhas de dados e endereço são válidos
bus_rdy_i	operação realizada no barramento está pronta
bus_wr_o	operação de escrita
bus_addr_o	endereço da operação requisitada
bus_data_i	dado lido
bus_xfer_o	granularidade da escrita (*byte, halfword, word*)
bus_cmd_o	tipo de operação
bus_req	pedido de acesso ao barramento
bus_rearrange	solicita mudança do barramento de dados
bus_gnt	confirmação do pedido de acesso
bus_data_o	dado de escrita
snoop_block_s	o dado requisitado vem de uma cache
snoop_block_m	alguma cache tem uma cópia do bloco no estado MODIFICADO
	interface de espionagem
bus_sel_i	valor nas linhas de dados e endereço é valido
bus_wr_i	operação de escrita
bus_rdy_o	pedido de operação está pronto
bus_addr_i	endereço vindo do barramento
bus_cmd_i	tipo de transação no barramento
snoop_req	uma interface vai começar um acesso ao barramento
snoop_hit_report	esta cache tem uma cópia do bloco requistado pelo barramento
snoop_modified_report	esta cache tem uma cópia do bloco no estado MODIFICADO

Tabela 3.1: Descrição da interface da cache de dados

Faixa de endereços (hex)	Dispositivo
00000000 - 00004000	ROM
00010000 - 00018000	RAM
0F000000 - 0F002000	I/O

Tabela 3.2: Mapa de endereçamento do sistema

bloco requisitado pelo núcleo pode estar na memória principal ou em uma cache remota. As caches de dados tem prioridade sobre a memória de dados e, se alguma cache possui uma cópia do bloco, então uma transação cache para cache deve ocorrer. Do contrário, a memória principal fornece os dados [6]. Na implementação aqui proposta, a prioridade é baseada no índice de cada unidade, na qual uma unidade u_i tem mais prioridade que u_j se i < j. Um erro no barramento ocorre se nenhuma unidade responde a uma requisição.

Na Figura 3.3, a variável $N-1$ representa o número de cores do sistema e $M-1$ o número de periféricos de E/S, e $N+M+1$ interfaces atendem requisições do barramento. As interfaces do tipo (iii) possuem maior prioridade e ocupam as posições $0..N-1$, a interface (iv) ocupa a posição N e as interfaces (v) as posições $N+1..M$. O sinal `dev_sel(k)` fica ativo se k é o índice da unidade que atende à requisição da cache. O valor do sinal *sel* de uma interface i é obtido pela conjunção de `dev_sel(i)` e `bus_sel`.

A política de *write-back* introduz outro detalhe de implementação do árbitro de espionagem: durante a escrita do bloco modificado na memória, a cache de dados é quem escreve no barramento de dados, e após término da escrita, a fonte do novo bloco precisa ser determinada. Neste contexto, após o *write-back*, o árbitro de espionagem verifica quais unidades têm o dado desejado e troca a seleção do multiplexador do barramento de dados para a fonte do bloco. A cache sinaliza fim do *write-back* através do sinal `bus_rearrange`.

No protocolo MESI original [6], se uma cache possui um bloco no estado MODIFICADO e uma outra cache requisita esse mesmo bloco, então a cache fonte do bloco atualiza a memória principal e coloca o estado do bloco em COMPARTILHADO [6], e então o controlador da cache faltosa obtém o bloco da memória principal ou de outra cache. Neste trabalho, a memória principal não é atualizada neste caso, mas ocorre uma transação de cache para cache, fazendo o estado do bloco na cache fonte ficar INVALIDO e a cache faltosa receber o bloco no estado MODIFICADO. Essa modificação foi sugerida em [4] com o objetivo de simplificar a lógica de controle dos controladores de cache e do barramento, pois a temporização de acesso dos controladores da cache é diferente da temporização do controlador da memória principal.

Por esse motivo, a implementação usa o sinal `snoop_modified_report(i)` indicando que a cache i contém o bloco requisitado no estado MODIFICADO. O árbitro de espionagem informa às caches se o dado reside em alguma das caches (necessário para o saber se estado do bloco é EXCLUSIVO ou COMPARTILHADO) e se alguma das caches tem o bloco no estado modificado (necessário para transações de cache para cache).

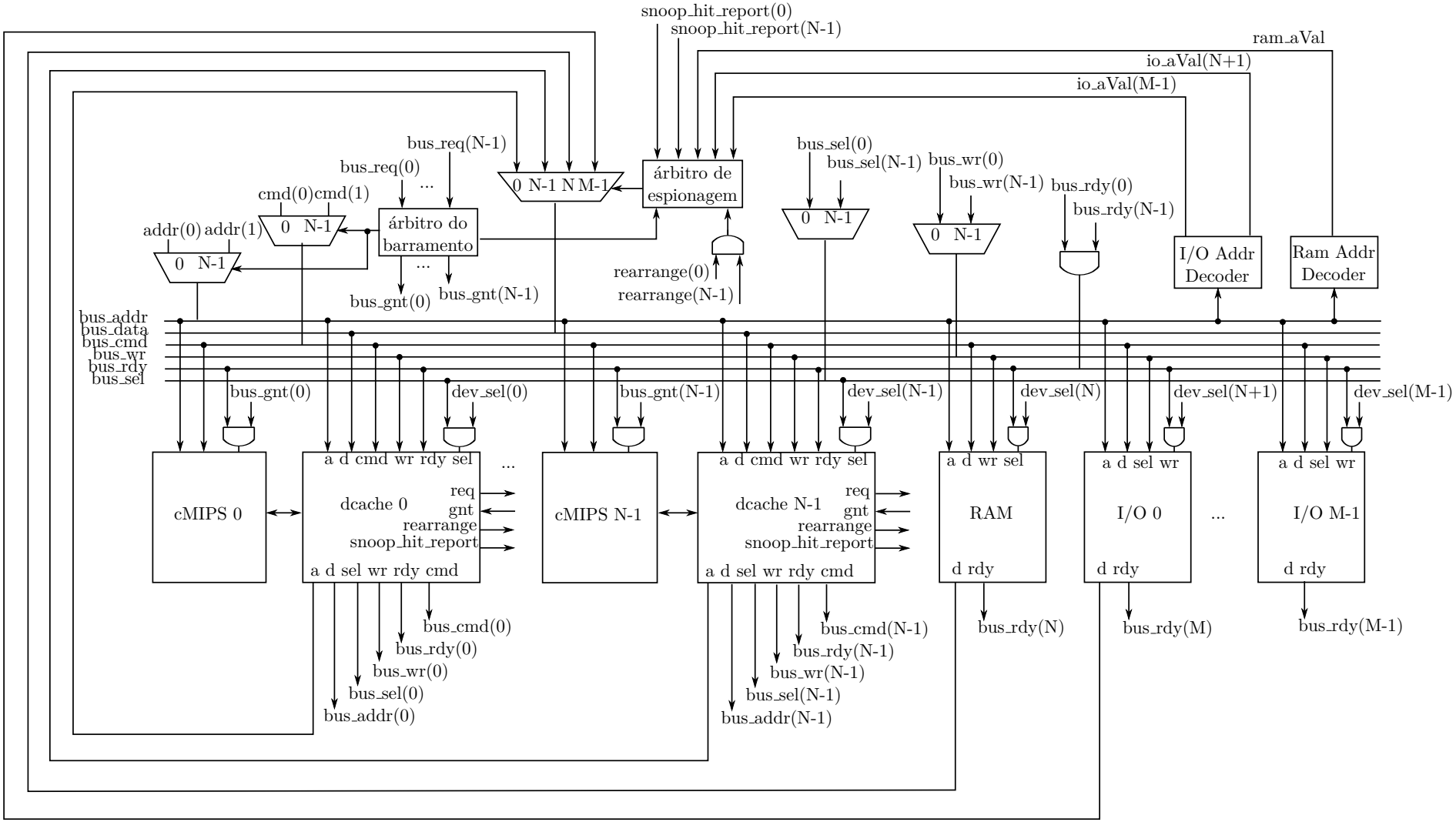

Figura 3.3: Estrutura do barramento

3.3 Cache de dados e protocolo de coerência

As caches de dados são diretamente mapeadas e implementam o protocolo MESI usando a política de escrita preguiçosa (*write-back*). A capacidade e tamanho do bloco podem ser configurados através de parâmetros *generics* no código VHDL. As memórias de dados, etiquetas e estado foram implementadas com o cuidado de que o sintetizador faça o uso de memórias RAM internas denominadas *Block RAMs*.

A implementação de caches coerentes introduz a necessidade de atender à requisições de outras caches, além das requisições do processador. Nesta implementação, são usados dois controladores: um controlador responsável por responder aos estímulos do processador, chamado de "controlador do processador", e o controlador responsável por atender aos estímulos do barramento, chamado de "controlador de espionagem". Cada controlador precisa acessar as memórias de dados e estado/etiquetas. Para minimizar o impacto no desempenho que caches coerentes trazem, as memórias de dados e etiquetas possuem duas portas, `A` e `B`, para permitir que os controladores acessem as etiquetas e dados ao mesmo tempo, desde que a blocos diferentes, como mostra a Figura 3.4.

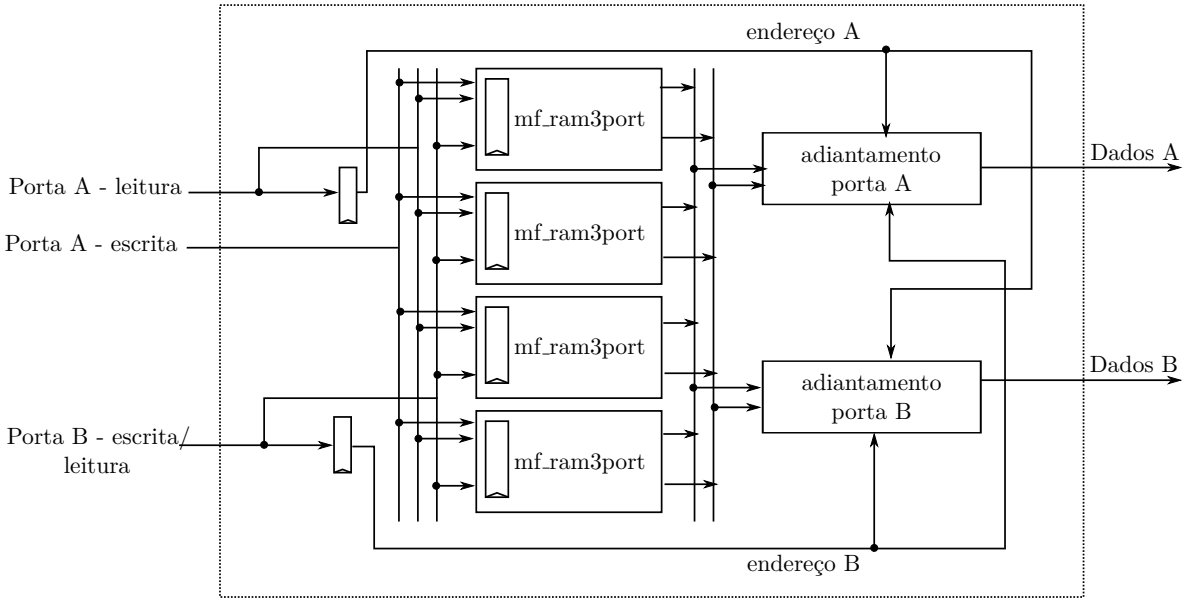

Figura 3.4: Organização da memória de etiquetas e dados

Devido a política de *write-back* não é possível escrever em um bloco enquanto a verificação de acerto é realizada, pois em caso de falta o bloco sujo deve ser copiado para memória principal. Um acerto de escrita requer dois ciclos para ser realizado - um para verificar acerto e outro para

escrita na memória de dados [7]. Neste trabalho, o processador é liberado para continuar sua execução no primeiro ciclo, e o tempo de acerto da cache é de 1 ciclo.

Em um determinado ciclo do relógio, três acessos simultâneos às memórias de etiquetas e/ou dados podem ocorrer. Vamos supor a seguinte situação: o núcleo emite a instrução SW no ciclo t e no ciclo $t+1$ emite um LW. Suponha também que no instante $t+1$ o controlador de espionagem atende uma requisição do barramento. Neste caso, no instante $t+1$, a memória de dados está realizando a escrita anterior, lendo a palavra requisitada pela processador e realizando a leitura de um bloco requisitado pelo barramento. Para contornar este problema, a porta `A` é subdividida em uma porta de leitura e outra de escrita. Como consequência, fez-se necessário um circuito de adiamento de dados para leitura. No exemplo anterior, se as instruções SW e LW referenciam o mesmo dado então o LW deve retornar o valor escrito no instante t.

Se a cache de dados não pode fornecer o dado no ciclo corrente, a interface de cache deve usar o barramento para realizar uma transação que permita servir o processador com o dado requisitado. Para tanto, quatro tipos de transação podem ocorrer no barramento [1]:

- Leitura ($BusRd$): Transação realizada quando há uma falta de leitura na cache;
- Leitura exclusiva ($BusRdX$): Transação realizada quando há uma falta de escrita na cache. Neste caso, lê o bloco da memória principal ou cache remota, e invalida todas as outras cópias;
- Escrita ($BusWb$): Essa transação ocorre quanto o controlador precisa substituir um bloco no estado MODIFICADO da cache (*writeback*);
- Upgrade ($BusUpgr$): Esta transação ocorre quando o processador precisa escrever em um bloco no estado COMPARTILHADO. Neste caso, invalida todas as outras cópias e não lê nenhum dado.

A Tabela 3.3 mostra as transações necessárias em função do tipo de operação, etiquetas e o estado atual do bloco. Quando ocorre um acerto na cache nenhuma transação é necessária.

3.4 Circuito de dados da cache

O caminho de dados da cache de dados é mostrado na Figura 3.5. Este caminho pode ser divido em seis componentes com objetivos distintos: o primeiro deles é o circuito que verifica acerto na cache; o segundo e terceiro realizam leitura e escrita de um bloco no barramento; o

Operação	Etiquetas iguais	Estado do bloco	BusRd	BusRdX	BusWB	BusUpgr
Leitura	Sim	MODIFICADO	0	0	0	0
Leitura	Sim	EXCLUSIVO	0	0	0	0
Leitura	Sim	COMPARTILHADO	0	0	0	0
Leitura	Sim	INVALIDO	1	0	0	0
Leitura	Não	MODIFICADO	1	0	1	0
Leitura	Não	EXCLUSIVO	1	0	0	0
Leitura	Não	COMPARTILHADO	1	0	0	0
Leitura	Não	INVALIDO	1	0	0	0
Escrita	Sim	MODIFICADO	0	0	0	0
Escrita	Sim	EXCLUSIVO	0	0	0	0
Escrita	Sim	COMPARTILHADO	0	0	0	1
Escrita	Sim	INVALIDO	0	1	0	0
Escrita	Não	MODIFICADO	0	1	1	0
Escrita	Não	EXCLUSIVO	0	1	0	0
Escrita	Não	COMPARTILHADO	0	1	0	0
Escrita	Não	INVALIDO	0	1	0	0

Tabela 3.3: Decodificação dos sinais de transação em função da operação requisitada e etiqueta/estado do bloco

quarto lê ou escreve um dado em um periférico; o quinto atende requisições de uma cache remota; e o sexto efetua a inicialização da cache.

A memória de dados é fisicamente organizada em palavras e é indexada pelo índice e palavra do endereço vindo do processador. A memória de etiquetas é indexada pelo índice do endereço emitido pelo processador. Como é visto na Seção 3.6, nesta implementação, o endereço vindo do núcleo é do estágio de execução (EX). Na borda do relógio, esse endereço é capturado diretamente pelos registradores da memória de etiquetas e dados. O registrador com saída `cpu_addr_old` também captura o endereço requisitado.

Na Figura 3.5, o circuito `Novo estado cpu` gera o próximo estado de um bloco para uma operação realizada pelo processador. Este circuito é baseado na Tabela 3.4 e depende do tipo da operação do processador e fonte do bloco. O circuito `Novo estado snoop` gera o próximo estado de um bloco para uma transação observada no barramento no qual a cache local tem cópia válida do bloco. Este circuito é baseado na Tabela 3.5 e depende da transação realizada no barramento e o estado atual do bloco.

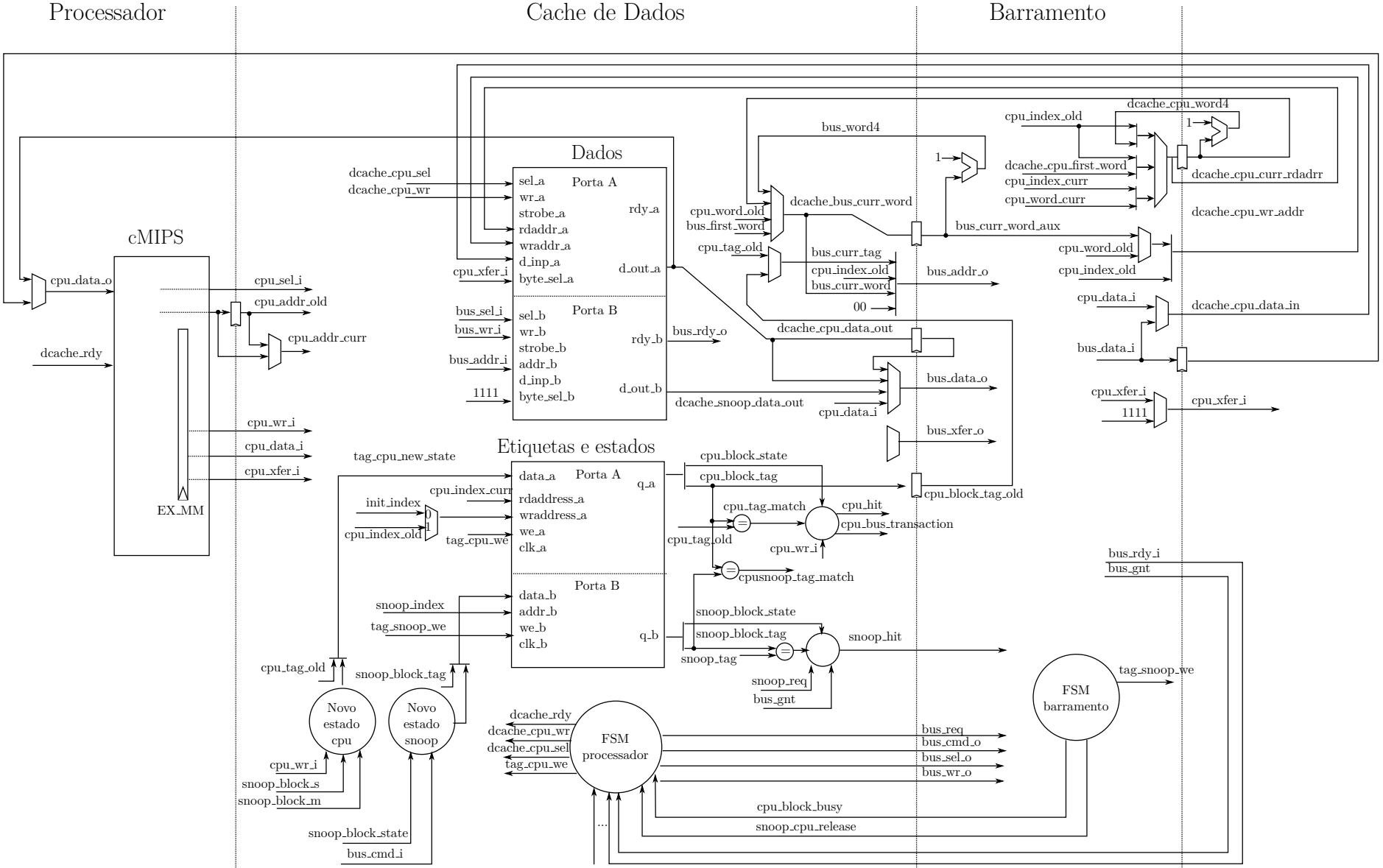

Figura 3.5: Caminho de dados com as duas unidades de controle

Operação	Origem do bloco	Modificado na cache fornecedora	Próximo estado
Escrita	Não importa	Não importa	MODIFICADO
Leitura	RAM	Não importa	EXCLUSIVO
Leitura	Cache	Não	COMPARTILHADO
Leitura	Cache	Sim	MODIFICADO

Tabela 3.4: Decodificação de estados dos blocos devido às requisições do processador

Tipo da transação	Estado atual do bloco	Próximo estado
BusRd	MODIFICADO	INVALIDO
BusRd	EXCLUSIVO	COMPARTILHADO
BusRd	COMPARTILHADO	COMPARTILHADO
BusRdX	Não importa	INVALIDO
BusUpgr	Não importa	INVALIDO

Tabela 3.5: Decodificação de estados dos blocos devido às requisições do barramento

3.4.1 Etapa de verificação

A etapa de verificação determina se a cache de dados pode fornecer, no ciclo corrente, o dado requisitado pelo núcleo cMIPS. Nesta etapa é analisado se existe um acerto na cache, se o bloco está sendo modificado por uma cache remota e se o dado apontado pelo endereço pode residir na cache.

Para verificar um acerto na cache, a etiqueta do endereço vindo do núcleo é comparada com a etiqueta vinda da memória de etiquetas. Se as etiquetas são iguais então o sinal `cpu_tag_match` fica ativo. Conforme mostra a Tabela 3.3, existe um acerto se a operação é de leitura de um bloco no estado MODIFICADO, EXCLUSIVO ou COMPARTILHADO, ou é uma operação uma escrita em um bloco no estado EXCLUSIVO ou MODIFICADO. Na Figura 3.5, o sinal `cpu_hit` indica um acerto, e em caso de falta, o sinal `cpu_bus_transaction` indica as transações necessárias para completar a instrução de transferência de dados.

Concorrentemente com a lógica combinacional da verificação da etiqueta e estados, outro circuito verifica se o bloco requisitado pelo núcleo está sendo modificado por outra cache - o controlador de espionagem está atendendo uma requisição para o mesmo bloco de uma cache remota. Neste caso, se o núcleo requisita um bloco que está sendo modificado por uma cache remota, então o controlador da cache de dados passa para um estado de espera até transação remota terminar. O sinal `cpusnoop_tag_match` fica ativo se a etiqueta do endereço do barramento é igual a etiqueta do bloco requisitado pela CPU. O sinal `snoop_block_lock` indica que a cache está servindo uma cache remota. Portanto se `cpusnoop_tag_match = 1` e `snoop_block_lock = 1` então a cache de dados fica esperando até a requisição da cache remota completar.

Visto que toda transação ocorrida no barramento é iniciada pelos controladores de cache, também é necessário verificar se o dado do endereço requisitado pode ficar na cache. O sinal `cpu_addr_cacheable` indica tal condição. Se o endereço indica um acesso a uma unidade de entrada/saída, então o controlador solicita acesso ao barramento para escrever o dado e posteriormente libera o núcleo cMIPS. A Seção 3.4.4 descreve os detalhes desta etapa.

Se o endereço requisitado satisfaz todas as condições acima, então o acesso à cache é concluído. No caso de uma escrita, a atualização da memória de dados e etiqueta é realizada no ciclo seguinte. Nesta situação, o endereço de escrita na memória de dados e etiqueta vem de `cpu_addr_old`; os dados a serem escritos vem do processador através do sinal `cpu_data_i`; e o novo estado é determinado com base na Tabela 3.4 e é indicado no sinal `tag_cpu_new_state`.

3.4.2 Tratamento de faltas

No caso de falta na cache, o controlador solicita o barramento no ciclo seguinte ao ciclo de verificação. A etiqueta, o índice e o *offset* do byte do endereço colocado no barramento são constantes durante o preenchimento da cache e são derivados do sinal `cpu_addr_old`, o qual contém o endereço da requisição do processador. A palavra do endereço, no entanto, não é constante durante o processo de leitura e é determinada pelo multiplexador com saída em `dcache_bus_curr_word`.

Quando o barramento está disponível, o controlador coloca o endereço da primeira palavra do bloco requisitado no barramento através do sinal `bus_first_word`, que representa o *offset* da primeira palavra de um bloco qualquer do sistema. Em seguida, o dado é buscado pela memória principal (ou cache remota) e o endereço é incrementado de modo que o sinal `bus_word4` aponte para a próxima palavra do bloco. Quando a memória completa seu acesso e sinaliza através do sinal `bus_rdy`, os dados do barramento são colocados na memória de dados da cache na próxima borda do relógio. Para isto, o sinal `dcache_cpu_wr` é ativado; o sinal `dcache_cpu_data_in` é selecionado para conter `bus_data_i`; e a palavra do próximo endereço no barramento vem de `bus_word4`. Este procedimento é repetido até a leitura da última palavra do bloco, e neste ponto a memória de etiquetas é atualizada com o sinal `tag_cpu_new_state` que contém a nova etiqueta e estado.

3.4.3 Tratamento de *writebacks*

Caso a falta na cache seja em uma linha já utilizada, então deve-se escrever o bloco sujo na memória principal e depois carregar o bloco requisitado pelo processador. O controlador da cache faz requisição do barramento e espera liberação. Após a permissão de acesso, o controlador lê a primeira palavra do bloco da memória de dados através da concatenação de `cpu_index_old` e `dcache_cpu_first_word`. No ciclo seguinte, escreve a palavra recém lida no barramento no endereço composto do endereço usado para acessar a cache (`dcache_cpu_curr_rdword_aux`) e da etiqueta do bloco antigo (`cpu_block_tag_old`). Enquanto realiza a escrita desta palavra, o controlador lê a próxima palavra a ser colocada no barramento.

A palavra lida da memória de dados (`dcache_cpu_data_out`) é capturada num registrador para encurtar o tempo de uma transação no barramento. Sem este, após o barramento sinalizar uma conclusão da escrita de uma palavra, seria necessário um ciclo adicional para o ler a próxima palavra para, posteriormente, escrever na memória principal. Desse modo, no ciclo imediatamente posterior ao término de uma escrita, a memória pode iniciar a escrita da próxima palavra. Assim, se a escrita na memória principal teve latência maior que 1 ciclo então o sinal `bus_data_o` recebe o sinal `dcache_cpu_data_out`, caso contrário recebe `bus_dcache_data_out_aux`.

Após o término do *writeback* o controlador da cache inicia a leitura do bloco requisitado pelo processador, como descrito na seção anterior.

3.4.4 Tratamento de endereços que não devem passar pela cache

Referências que pertencem à faixa de endereços de periféricos de entrada e saída não examinam e nem causam mudança no estado da cache. Elas apenas colocam o endereço e dados no barramento e esperam o periférico de destino concluir a transação. Como mostrado na Tabela 3.2, os endereços que pertencem os periféricos de entrada e saída possuem os bits 24 a 27 ligados. A implementação consiste em verificar se estes bits estão ligados e ativar o sinal `cpu_addr_cacheable`.

Os multiplexadores que compõem o sinal de `bus_addr_o` são selecionados para que o sinal seja idêntico a `cpu_addr_old`; o sinal `bus_data_o` recebe `cpu_data_i`; e o sinal `bus_wr_o` recebe o valor de `cpu_wr_i`. No caso de uma operação de leitura, ao término da transação o controlador da cache coloca os dados vindos do barramento (`bus_data_i`) na entrada do processador.

3.4.5 Requisições de caches remotas

A porta `B` da memória de dados da cache é usada para atender requisições de caches remotas. Como visto na Seção 3.1, a interface de comunicação de uma cache com a memória principal ou cache remota é a mesma. Durante uma transferência de dados de cache para cache, a fornecedora de dados se comporta como a memória principal. Para tanto, os sinais de controle, endereço e dados do barramento são ligados diretamente à porta `B` da memória de dados. O multiplexador que gera a saída de `bus_data_o` é selecionado para a saída da porta `B` da memória de dados através do sinal `dcache_snoop_data_out`.

3.4.6 Inicialização da cache

A inicialização da cache consiste em atribuir o estado INVALIDO para todos os blocos da cache. Um contador indexa a memória de etiquetas e o sinal `tag_cpu_new_statetag` contém a concatenação da constante que representa o estado INVALIDO com uma etiqueta qualquer.

3.5 Controle da cache de dados

O controle da cache de dados é implementado através de duas máquinas de estados. A primeira é encarregada de responder aos estímulos do processador gerando os sinais de controle para modificar o estado interno da cache e gerar transações no barramento. A segunda é o controlador responsável por atender aos estímulos do barramento e gera os sinais de controle para modificar a memória de etiquetas.

3.5.1 Controlador do lado da CPU

A máquina de estados que reage aos estímulos vindos do processador é mostrada na Figura 3.6. Os retângulos com bordas tracejadas representam os estados que fazem interface com as unidades anexadas ao barramento e geram os sinais de controle mostrados na Seção 3.1.

A máquina de estados inicia no estado $IDLE$. O sinal `cpu_sel_i`, se ativo, indica que existe uma instrução de acesso à memória no estágio de execução (EX) e a máquina transiciona para o estado $CHECK$, responsável por realizar as verificações descritas na Seção 3.4.1. Se o endereço requisitado não deve passar pela cache, então a máquina transiciona para o estado BUS_REQ_IO. No estado BUS_REQ_IO é emitido uma requisição do barramento fazendo

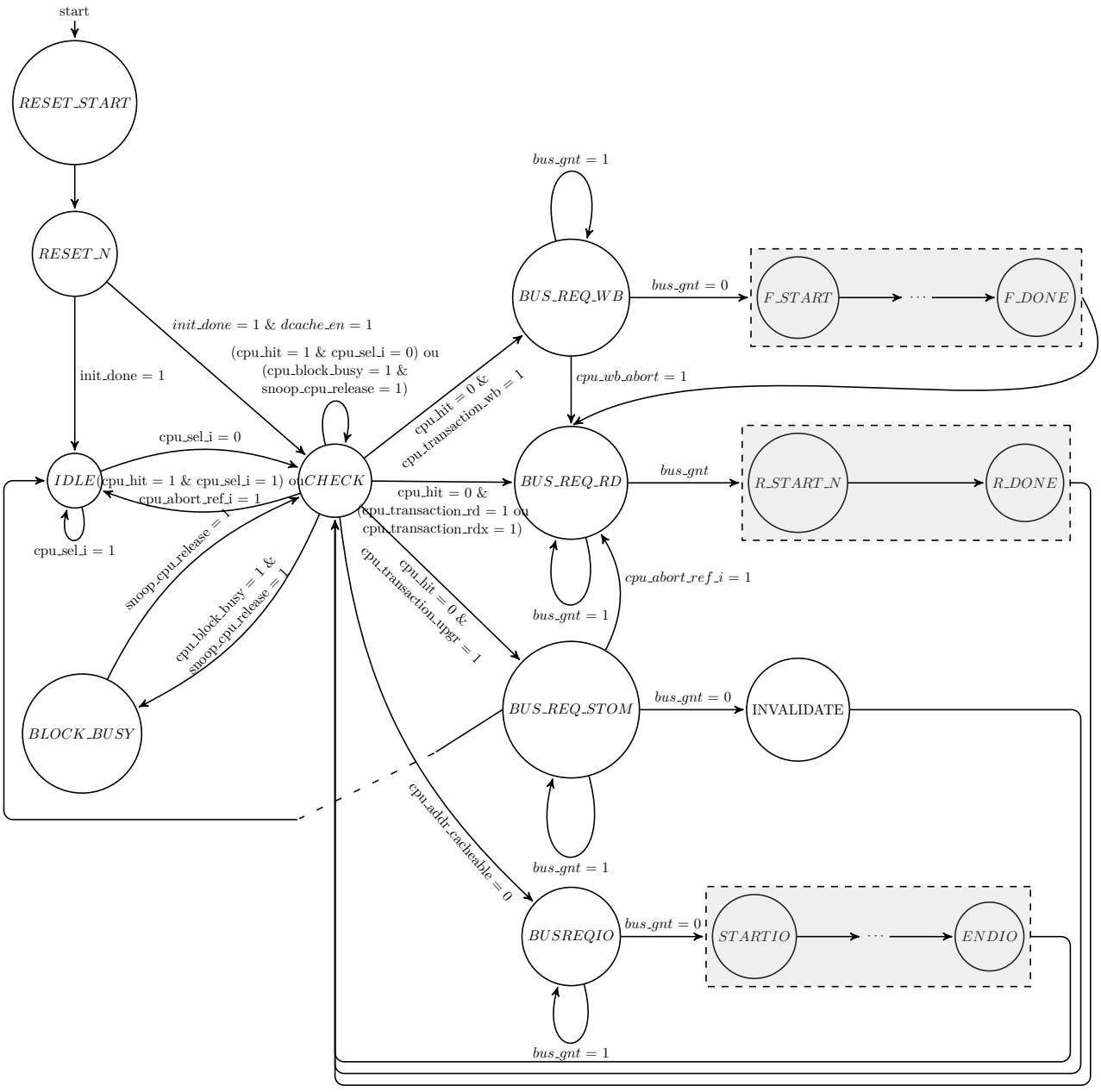

Figura 3.6: Máquina de estados do controlador da cache de dados do lado do processador

`bus_req = 0`. Quando o árbitro concede o barramento, então a máquina entra em uma sequência de estados que gera os sinais da interface de entrada/saída para transferência de dados.

No estado $CHECK$, se o bloco solicitado está sendo modificado por uma cache remota, então ocorre uma transição para o estado $BLOCK\_BUSY$. A máquina permanece em $BLOCK\_BUSY$ enquanto o sinal `snoop_cpu_release` não é ativado. Este sinal é ativado quando a cache remota termina sua transação, fazendo com que a máquina do processador volte para o estado $CHECK$, para tentar acessar o bloco da cache novamente.

As demais transições saindo do estado $CHECK$ ocorrem quando existe uma falta na cache e são baseadas na Tabela 3.3. Se a transação necessária é do tipo $BusWb$ então máquina muda para o estado BUS_REQ_WB; se a transação é do tipo $BusRd$ ou $BusRdX$, então máquina passa para o estado BUS_REQ_RD para ler o bloco da memória principal ou cache remota; caso contrário, é necessário invalidar outras cópias (transação $BusUpgr$) e neste caso passa para estado $BUS\_REQ\_S\_TO\_M$. Uma vez que a requisição do barramento é atendida, a máquina realiza a transação necessária para tratar a falta e volta para o estado $CHECK$. Note que depois de realizar um *write-back* do bloco, a máquina passa para o estado BUS_REQ_RD com o objetivo de fazer o árbitro de espionagem decidir qual unidade vai responder à requisição do bloco. Isto não faz o controlador liberar o barramento, pois então a transação no barramento não seria atômica.

O Algoritmo 1, apresentado no Capítulo 2, considera que as ações acontecem de forma atômica. A implementação do algoritmo consiste de uma sequência de etapas discretas: primeiro o controlador verifica se as etiquetas são iguais, depois tenta adquirir o barramento e, quando adquire, realiza a transferência dos dados. Essas ações discretas introduziram algumas complicações na implementação do protocolo MESI. Considere, por exemplo, que dois processadores compartilhem em suas caches, $C0$ e $C1$, um bloco qualquer e desejam escrever neste bloco ao mesmo tempo. Neste caso, vamos supor que a máquina do lado do processador estava no estado $IDLE$, passou para $CHECK$ e está em $BUS\_REQ\_S\_TO\_M$, esperando a liberação do barramento para invalidar as outras cópias. Suponha ainda que $C0$ ganha o barramento, invalida as outras cópias do bloco e volta para o estado $CHECK$ liberando o barramento. Quando $C1$ adquire o barramento, a sua cópia do bloco está no estado INVALIDO e a cópia de $C0$ em MODIFICADO. Portanto, quando o barramento é concedido a $C1$, este deve ler o bloco de $C0$ para posteriormente modificá-lo.

Essa complicação é resolvida fazendo com que controlador, no estado $BUS\_REQ\_S\_TO\_M$, fique monitorando o barramento e, caso observe uma transação do tipo $BusRdX$ ou $BusUpgr$ no barramento, então a máquina do lado do processador passa para o estado BUS_REQ_RD e continua solicitando o barramento. No momento em que sua requisição é atendida, o controlador inicia uma transação de leitura exclusiva ($BusRdX$) para buscar o valor mais recente do bloco, desse modo mantendo o sistema de memória coerente.

Outro cuidado necessário é a transição de BUS_REQ_WB para BUS_REQ_RD. Do protocolo MESI, se um bloco está no estado MODIFICADO, então ele é a única cópia válida

do bloco no sistema. Se a cache está em BUS_REQ_WB esperando liberação do barramento para fazer *write-back* do bloco modificado e verifica uma transação do tipo $BusRd$ ou $BusRdX$ ocorrendo, então significa que outra cache ganhou acesso ao barramento mas a máquina de estados de espionagem, da sua própria cache, está atendendo à requisição corrente do barramento. Por consequência, quando a cache recebe o barramento, não é necessário fazer *write-back* do bloco, pois uma cache remota obteve o bloco, e a cache local apenas realiza a operação de leitura do novo bloco.

Os estados $RESET\_START$ e $RESET\_N$ invalidam todos os blocos da cache. Caso o processador requisite um dado durante esta etapa, ao término da inicialização, a máquina passa para o estado $CHECK$, caso contrário passa para o estado $IDLE$.

3.5.2 Controle do lado do barramento

O controle dos estímulos vindos do barramento é implementado com uma máquina de estados finitos, como mostrada na Figura 3.7. Esta máquina é responsável por controlar o sinal de escrita da memória de etiquetas e sinalizar ao controlador do lado do processador que um bloco está sendo modificado. Um acerto de espionagem ocorre quando a cache possui uma cópia válida do bloco sendo requisitado no barramento e a requisição vem de uma cache remota. Neste caso `snoop_hit = 1` indica um acerto de espionagem.

A máquina permanece no estado $CHECK$ enquanto não existe um acerto de espionagem, ou o tipo da transação no barramento é $BusWb$ - um *write-back* não modifica o estado dos blocos de uma cache remota. No caso de acerto de espionagem e uma requisição de invalidação é observada, a máquina atualiza o estado do bloco para INVALIDO e retorna para o estado $CHECK$. No caso de leitura, transiciona para o estado $WAIT\_SEL$ e atualiza o estado do bloco conforme a Tabela 3.5. Este estado é necessário pois caso o bloco esteja COMPARTILHADO, então outras caches podem responder positivamente, mas apenas uma cache serve à requisição. Se o árbitro de espionagem seleciona outra cache para responder à requisição do barramento, então a máquina volta para o estado $CHECK$. O sinal que faz a requisição de um bloco fica ativo apenas durante a fase de arbitragem para fazer com que a máquina permaneça no estado $CHECK$ durante uma transação de um bloco compartilhado sendo servido por outra cache.

Se a cache local deve responder, então a máquina do barramento passa para o estado $FLUSH$ e permanece nele até que a transação seja concluída. Se a transação é do tipo leitura exclusiva então o bloco passa para o estado INVALIDO; se bloco está MODIFICADO então

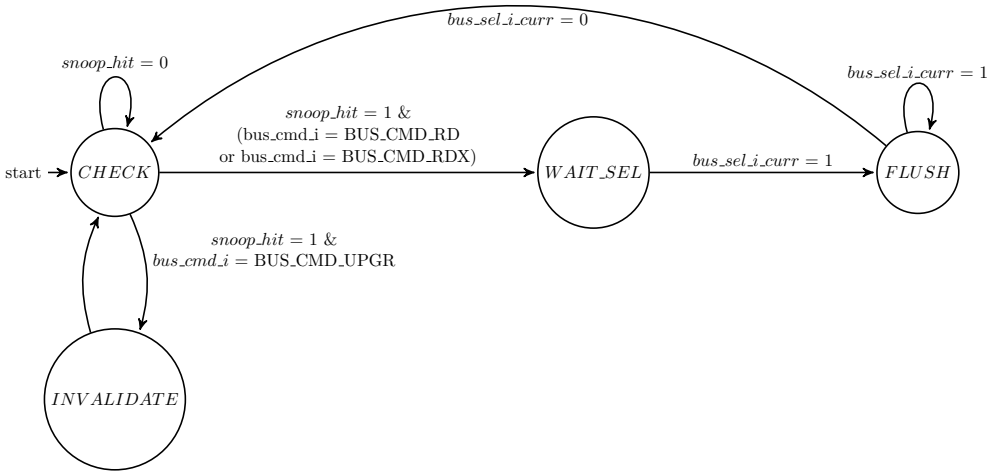

Figura 3.7: Máquina de estados do controlador da cache de dados do lado do barramento

fica INVALIDO; se EXCLUSIVO então fica COMPARTILHADO; e se COMPARTILHADO permanece no mesmo estado.

3.6 Metodologia de temporização

Os registradores de pipeline do núcleo cMIPS, a memória de dados da cache de dados, a porta `A` da memória de etiquetas, a máquina de estados do controlador da cache do lado do processador e do barramento trabalham com o mesmo relógio `clk`. A porta `B` da memória de etiquetas trabalha em um relógio chamado `phi2`.

Como visto na Seção 3.2, a arbitragem do barramento é realizada em um ciclo do relógio. Na borda de subida de `clk`, as unidades sinalizam um acesso ao árbitro do barramento. Seja $t_{arbitro}$ o tempo necessário para decidir qual a próxima unidade a usar o barramento e t_{prop_mux}, o tempo de propagação do multiplexador que seleciona o endereço do barramento. Depois de aproximadamente $t_{arbitro} + t_{prop\_mux}$ unidades de tempo, o endereço está estável nas entradas das interfaces de espionagem e decodificadores de endereço. A partir desse momento as caches consultam a memória de etiquetas para decidir se contêm o bloco requisitado no barramento. A porta `B` (utilizada pelo controle de espionagem) trabalha em um relógio com borda de subida deslocada de, aproximadamente, $t_{arbitro} + t_{prop\_mux}$ unidades de tempo com relação a `clk`.

Outro ponto a considerar refere-se à máquina de estados do controlador do lado do processador. Esta máquina permanece no estado $IDLE$ enquanto não existe uma referência à memória no estágio MM do pipeline do cMIPS. Quando um operando de memória entra no estágio MM, a máquina de estados do processador deve estar no estado $CHECK$ para determinar um

acerto ou falta. Como visto na Seção 3.5.1, durante o estado $CHECK$, a memória de etiquetas é lida e comparada com a etiqueta fornecida pelo processador. Além disso, a etiqueta lida é comparada com a etiqueta do barramento para verificar se o bloco requisitado está sendo modificado pelo controlador de espionagem. Esta lógica combinacional exige que a memória de etiquetas seja lida o quanto antes para sinalizar ao núcleo cMIPS, um acerto ou falta, antes da próxima borda do relógio. Tendo isso em vista, a leitura das etiquetas é ativada pelo mesmo relógio que sincroniza os registradores do pipeline do cMIPS.

As memórias de dados, etiquetas e estados foram implementadas com blocos internos BRAM de memória do FPGA. Implementações típicas dessas memórias colocam um registrador nas portas de entrada, e para evitar que o endereço seja capturado pelo registrador de pipeline e um ciclo depois pelos registradores das BRAM, o endereço vindo do núcleo é do estágio de execução (EX). Na borda do relógio, esse endereço é capturado diretamente pelos registradores da memória de etiquetas e dados da cache.

3.7 Primitivas de sincronização

O conjunto de instruções do MIPS fornece as instruções *Load-Linked* (LL) e *Store-conditional* (SC) para implementar uma operação de *read-modify-write* (RMW) atômico. Como mostrado na Seção 2.1, a implementação do núcleo cMIPS possui suporte as instruções SC e LL em um ambiente no qual somente exceções fazem um SC falhar.

No contexto de multiprocessadores é necessário considerar acessos realizados por outros núcleos do sistema. Neste trabalho, o núcleo espiona o barramento de endereço e comandos para invalidar, quando necessário, o bit `LLBit`. Se existe uma transação no barramento cujo endereço corrente é igual ao endereço em `LLAddr` e o tipo da transação é de BusRdX ou BusUpgr, então a instrução SC deve falhar.

Duas situações podem ocorrer neste caso. Na primeira delas o bit `LLBit` foi invalidado antes da instrução SC chegar ao estágio MM. Isto faz com que processador não ative a cache de dados durante MM, mas apenas atribui uma falha ao retorno do SC para o estágio WB. A outra situação ocorre quando o SC entra no estágio MM e o `LLBit` está ativo, fazendo com que a cache inicie uma transação para atender ao pedido do processador.

Se durante o estado $CHECK$ não ocorre uma modificação no bloco, por parte de uma cache remota, e existe um acerto na cache então o SC completa. Caso contrário, se ocorreu uma

modificação, então a cache aborta a referência; se ocorreu uma falta na cache então o controlador solicita o barramento no ciclo seguinte. Quando o controlador está esperando liberação do barramento e uma modificação no mesmo bloco ocorre, então o controlador também aborta a referência em memória e libera o processador para continuar sua execução.

O sinal `cpu_abort_ref_i` foi adicionado a interface da cache de dados com o núcleo cMIPS e é controlado pelo núcleo. Este sinal fica ativo quando ocorre uma transação do tipo BusRdX ou BusUpgr para endereço contido em `LLAddr` e existe SC no estágio de MM. O processador determina que a cache aborte a transação corrente, e por consequência, a cache notifica ao processador que pode continuar sua execução.

A execução de um LL por um processador não faz um SC para o mesmo bloco falhar em outro processador. A granularidade da invalidação é de uma palavra - uma modificação em uma palavra diferente do mesmo bloco não causa um SC falhar em outro processador. Nesta implementação, se ocorrer substituição do bloco apontado pelo endereço `LLAddr` então o resultado de um SC é indefinido.

Capítulo 4

Simulações e testes funcionais

As simulações realizadas permitem avaliar a funcionalidade da implementação do cMIPS multicore. Os códigos de teste foram compilados usando as ferramentas *mips-gcc* e *binutils* através de *scripts* de compilação do cMIPS. Dado um código em *assembly* ou código C, os *scripts* de compilação produzem os arquivos *prog.bin* e *data.bin* que são utilizados para inicialização dos modelos em VHDL da memória RAM e ROM respectivamente.

Na Seção 4.1 uma bateria de testes é descrita e objetiva verificar a manutenção da coerência do sistema de memória. Na Seção 4.2 são apresentados os resultados da síntese em lógica programável.

4.1 Testes funcionais de coerência

Os testes descritos nas próximas seções avaliam somente o protocolo de coerência e não as operações básicas da cache de dados. Os testes permitem avaliar se a implementação proposta se equivale ao protocolo MESI especificado na Seção 2.3. O tamanho do bloco foi configurado para duas palavras e as unidades de memória possuem latência zero nos acessos para leitura e escrita.

4.1.1 Primeira leitura de um bloco

O objetivo deste teste é avaliar as transações causadas por faltas de leitura na cache, as quais causam transições do estado INVALIDO para os estados EXCLUSIVO, COMPARTILHADO ou MODIFICADO. O sistema é composto de três núcleos e o código *assembly* executado em

cada núcleo é mostrado na Figura 4.1. A Figura 4.2 mostra o diagrama de tempo a partir da primeira referência a memória.

Inicialmente, os três processadores requisitam um dado fazendo com que ocorra uma falta de leitura em cada cache. $C1$ e $C2$ requisitam o mesmo bloco, enquanto $C0$ solicita um bloco distinto. A prioridade inicial do algoritmo *round-robin* é a cache $C0$, logo $C1$ e $C2$ devem esperar liberação do barramento. No ciclo t_4 a cache $C0$ escreve a última palavra do bloco e atualiza o estado para EXCLUSIVO. $C1$ termina sua transação em t_8 e também coloca o bloco no estado EXCLUSIVO. Durante leitura de $C1$, o processador ligado a cache $C0$ faz uma escrita no bloco recém lido e atualiza sua etiqueta para MODIFICADO, sem causar uma transação no barramento.

No ciclo t_9 a cache $C2$ ganha permissão para acessar o barramento e requisita um bloco do qual $C1$ tem uma cópia no estado EXCLUSIVO. $C1$ responde à requisição de $C2$ e ao término da transação os controladores das duas caches contêm o bloco no estado COMPARTILHADO. Por fim, no ciclo t_{13}, a cache $C1$ solicita um bloco que está no estado MODIFICADO em $C0$. Portanto, $C0$ responde às requisições de $C1$ e coloca seu bloco no estado INVALIDO, enquanto que em $C1$, o bloco é armazenado no estado MODIFICADO.

```
addi $3,$0,-10          addi $3,$0,-10          addi $3,$0,-10
nop                     nop                     nop
lw   $4, 0($15)         lw   $4, 8($15)         lw   $4, 8($15)
sw   $3, 0($15)         lw   $3, 0($15)         nop
nop                     nop                     nop
wait                    wait                    wait
```

(a) (b) (c)

Figura 4.1: Trecho do código que avalia leituras

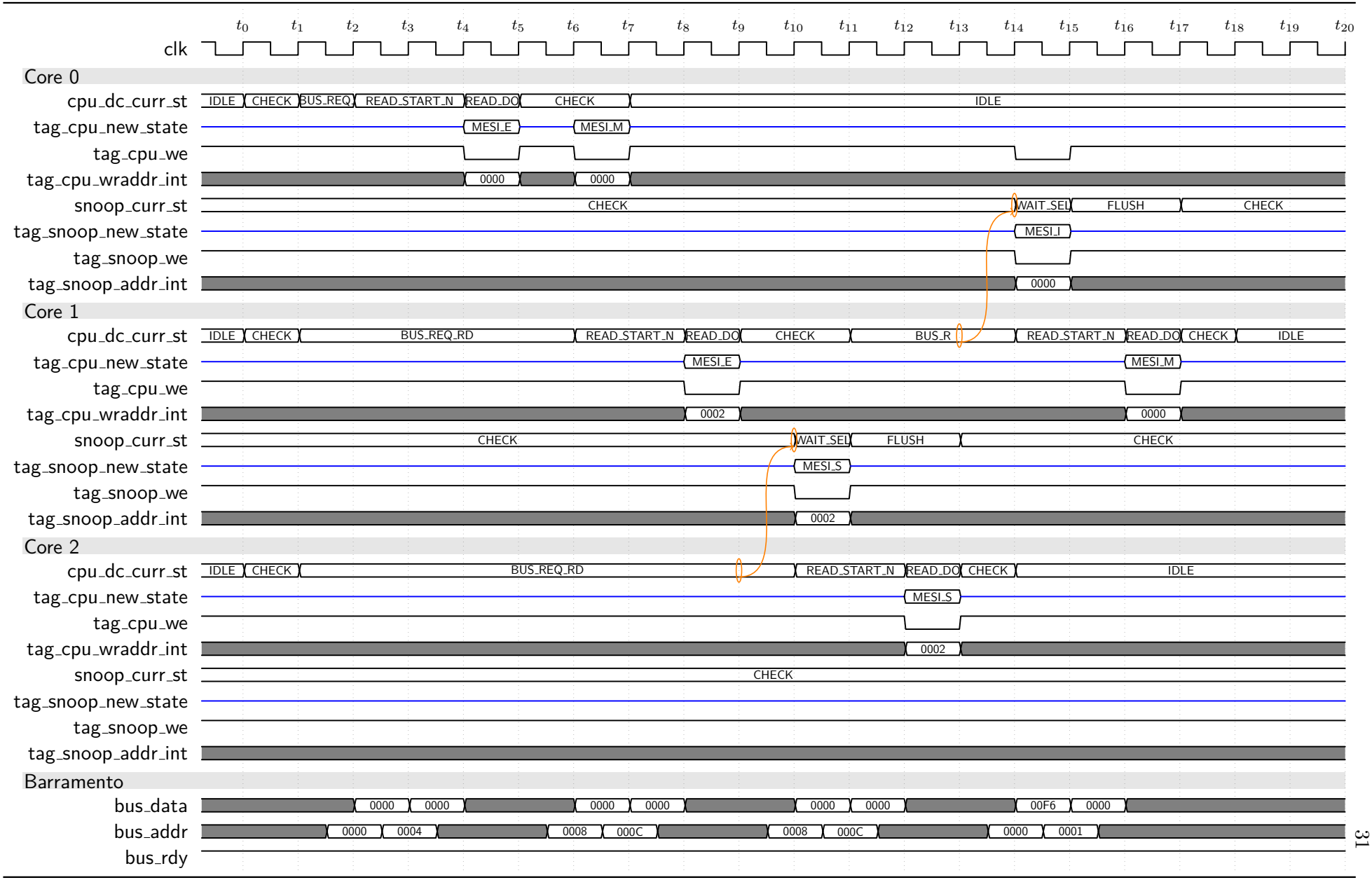

Figura 4.2: Diagrama de tempo da execução do código mostrado na Figura 4.1

4.1.2 Invalidação de outras cópias

O objetivo deste teste é avaliar as transações que causam invalidação das cópias de um bloco sendo referenciado. A transação $BusRdX$ indica a leitura de um bloco no barramento e invalida todas suas cópias, enquanto $BusUpgr$ causa apenas invalidações e não retorna os dados do bloco requisitado. O teste foi descrito em *assembly* e emprega três núcleos, cada um executando um dos trechos de códigos mostrado na Figura 4.3. A Figura 4.4 mostra o diagrama de tempo a partir da primeira referência à memória.

Primeiramente, ocorre uma falta de leitura em cada cache do sistema; as caches $C0$ e $C1$ referenciam o mesmo bloco $b0$, e $C2$ o bloco $b1$. A prioridade inicial do árbitro é a cache $C0$, logo $C1$ e $C2$ devem esperar liberação do barramento. No ciclo t_4 a cache $C0$ escreve a última palavra do bloco e atualiza o estado do bloco $b0$ para EXCLUSIVO. Entre os ciclos t_6 e t_8 a cache $C1$ realiza a cópia do bloco recém lido em $C0$, fazendo o estado ficar COMPARTILHADO nas duas cópias do bloco $b0$. Durante a leitura de $C1$, a cache $C0$ tenta realizar uma escrita no bloco recém lido $b0$, o qual está sendo transferido para $C1$. A máquina de estado do lado do processador de $C0$ passa para o estado $BLOCK\_BUSY$ para esperar a transação de $C1$ concluir.

Em seguida, a prioridade do barramento está com $C3$, que lê o bloco $b1$ entre os ciclos t_{10} e t_{12}, enquanto $C0$ e $C1$ esperam pela liberação do barramento. Quando $C3$ libera o barramento, $C0$ consegue invalidar as cópias do bloco $b0$ e escreve o novo valor. Posteriormente, $C1$ lê o bloco $b1$ de $C2$. No ciclo t_{19}, $C1$ e $C2$ possuem o bloco no estado COMPARTILHADO. Por fim, $C0$ deseja escrever em $b1$ causando uma falta de escrita e gerando uma transação de $BusRdX$. É importante notar que $C1$ e $C2$ têm o bloco requisitado, mas $C1$ é quem serve à requisição; $C1$ e $C2$ invalidam o bloco, mas apenas $C1$ serve o bloco requisitado.

```
addi $3,$0,-10          addi $3,$0,-10          addi $3,$0,-10
nop                     nop                     nop
lw   $4, 0($15)         lw   $4, 0($15)         lw   $4, 8($15)
sw   $3, 0($15)         lw   $3, 8($15)         nop
sw   $3, 8($15)         nop                     nop
nop                     nop                     nop
wait                    wait                    wait
```

(a) (b) (c)

Figura 4.3: Trecho do código que avalia invalidações

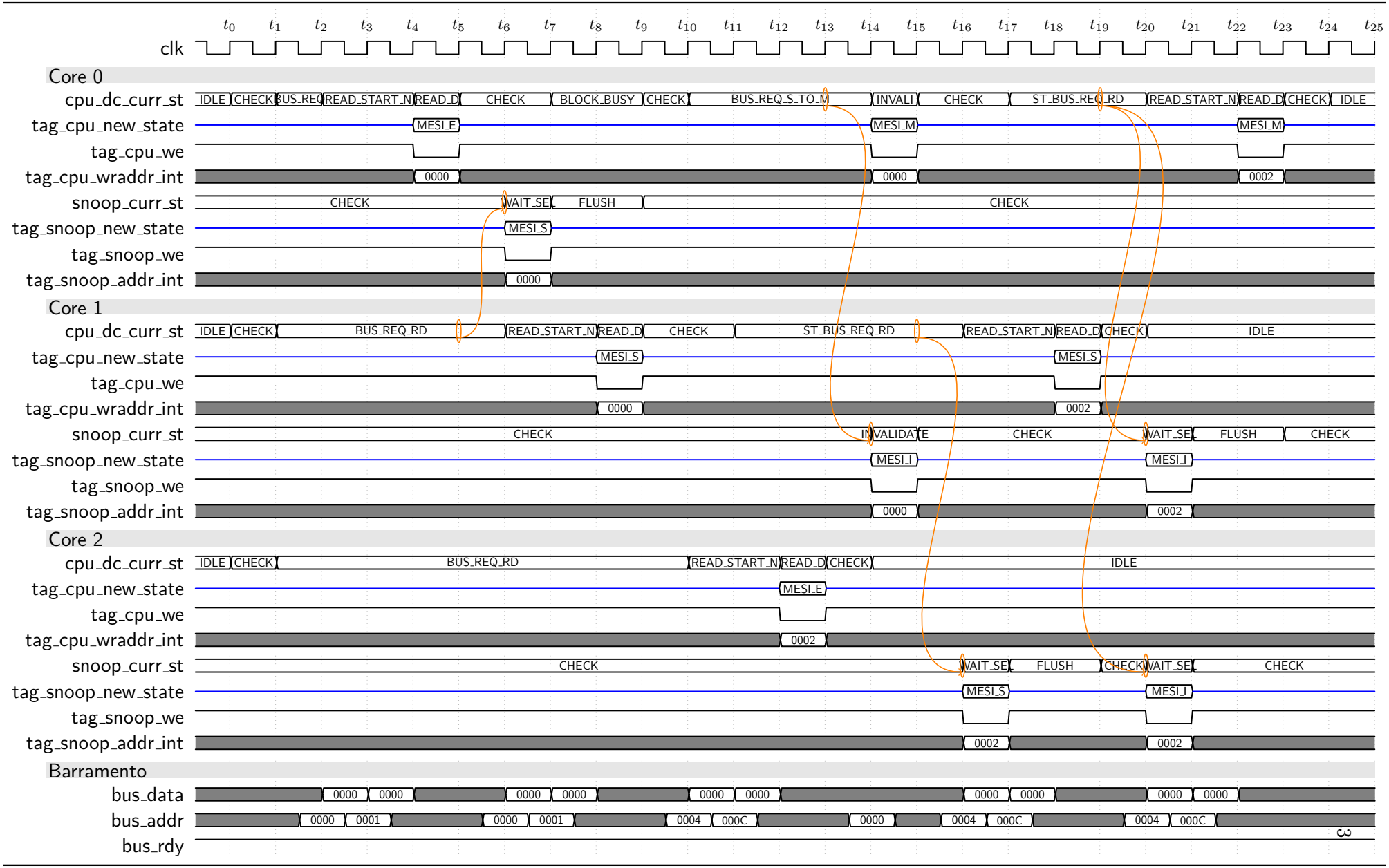

Figura 4.4: Diagrama de tempo da execução do código mostrado na Figura 4.3

4.1.3 Teste das operações discretas

O objetivo deste teste é avaliar o efeito das operações discretas nas ações do controlador da cache de dados. O teste foi descrito em *assembly* e é mostrado na Figura 4.5. A Figura 4.6 mostra o diagrama de tempo a partir da primeira referência à memória. Inicialmente, ocorre uma falta de leitura nas caches $C0$ e $C1$ para o mesmo bloco. A prioridade inicial do árbitro é a cache $C0$, logo $C1$ deve esperar liberação do barramento. No ciclo t_4 a cache $C0$ escreve a última palavra do bloco e atualiza o estado para EXCLUSIVO. Em seguida, $C1$ ganha o barramento e copia o bloco da cache $C0$ para a memória de dados local. Neste ponto, as duas caches contêm o mesmo bloco no estado COMPARTILHADO.

Em seguida, as duas caches tentam escrever no bloco, $C0$ em t_9 e $C1$ em t_{10}. Como $C0$ ganha o barramento, $C1$ transiciona do estado $BUS\_REQ\_S\_TO\_M$ para o estado BUS_REQ_RD, visto que sua cópia está invalida e precisa ler o novo valor do bloco antes de modificá-lo. No ciclo em t_{15}, a cache $C1$ termina a leitura do bloco modificado de $C0$. Neste ponto, a cópia de $C0$ está no estado INVALIDO e de $C1$ no estado MODIFICADO.

Na segunda parte do teste, $C0$ realiza novamente a leitura do bloco, agora armazenado em $C1$, enquanto espera o barramento para substituir o bloco modificado - por causa da falta de escrita. Enquanto $C1$ espera pelo barramento, o controlador do barramento serve a requisição de $C0$. Quando $C1$ ganha o barramento no ciclo t_{27}, ele não deve mais fazer *flush* do bloco, mas apenas a leitura do novo bloco.

```
addi $3,$0,-10
nop
lw   $4,  0($15)
sw   $3,  0($15)
nop
...
nop
lw   $4,  0($15)
nop
wait
```

(a)

```
addi $3,$0,-9
nop
lw   $4,  0($15)
sw   $3,  4($15)
nop
...
nop
lw   $4,  2048($15)
nop
wait
```

(b)

Figura 4.5: Trecho do código que avalia as operações discretas das ações do controlador da cache de dados

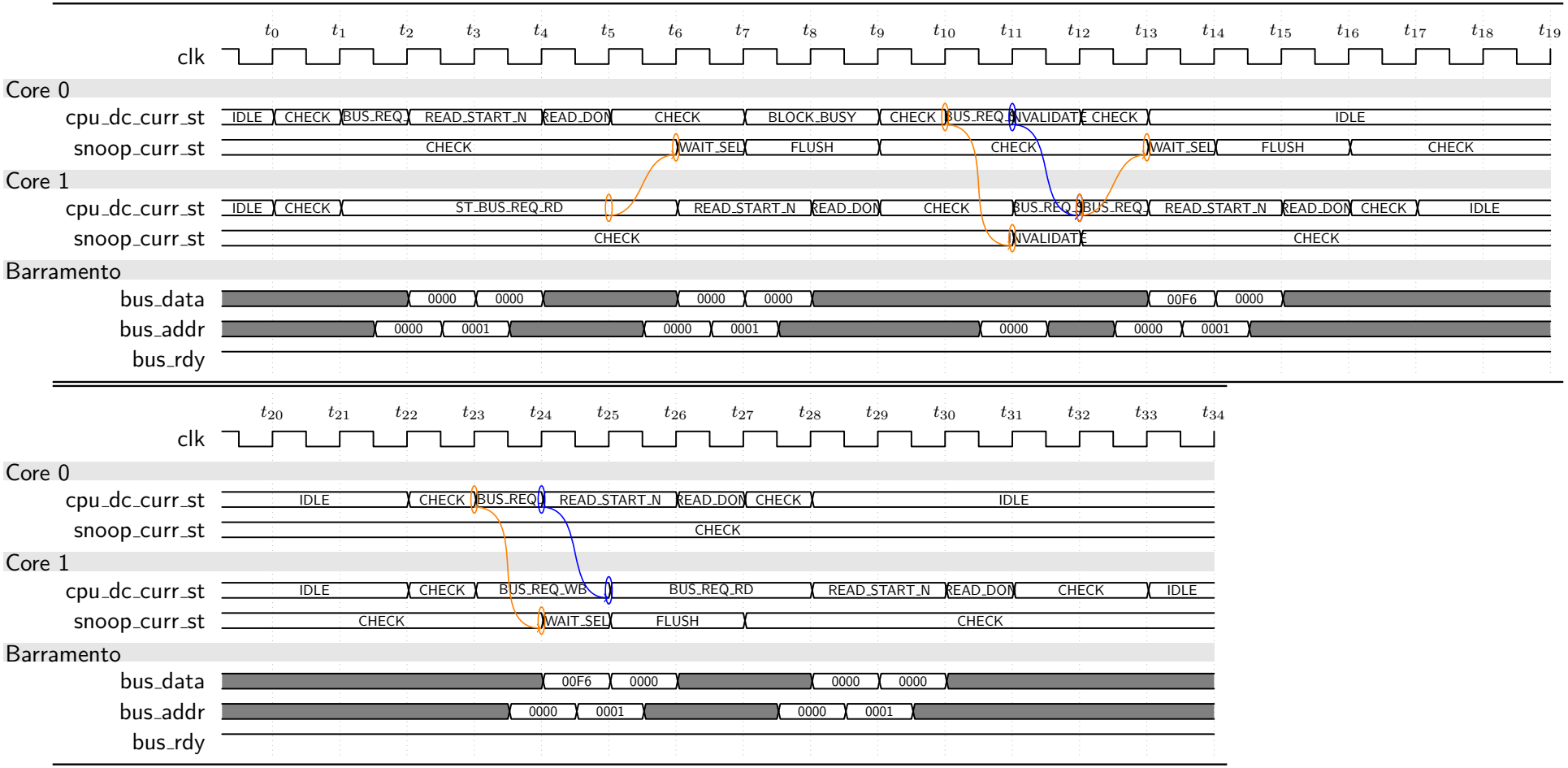

Figura 4.6: Diagrama de tempo da execução do código mostrado na Figura 4.5

4.1.4 Primitivas de sincronização

O objetivo deste teste é verificar a corretude da implementação das primitivas de sincronização. O teste foi descrito em *assembly* e a parte importante do teste é mostrada na Figura 4.7. A ideia do teste é realizar uma operação de read-modify-write (RMW) atômico de uma palavra.

```
ll    $3, 0($15)        ll    $3, 0($15)
sc    $3, 0($15)        sc    $3, 0($15)
```

(a) (b)

Figura 4.7: Trecho de código que testa as primitivas de sincronização

Os testes supõem o seguinte cenário: dois processadores, com caches $C0$ e $C1$, realizam um LL para um mesmo bloco e em seguida os dois tentam fazer um SC no bloco recém lido. Vamos supor que $C0$ tem prioridade de acesso ao barramento. Devemos analisar as seguintes situações:

(i) O processador de $C1$ verifica a invalidação de $C0$ antes do SC entrar no estágio MM;

(ii) O processador de $C1$ verifica a invalidação de $C0$ enquanto o SC está no estágio MM;

(iii) O processador de $C1$ verifica a invalidação de $C0$ depois do SC passar pelo estágio MM;

O caso (i) é resolvido dentro do núcleo cMIPS e ocorre quando o SC de $C1$ está no estágio EX, ou qualquer ciclo anterior, e observa uma invalidação de $C0$. A Figura 4.8 mostra o caso extremo quando o SC de $C1$ é invalidado na fase EX. No ciclo t_1 a cache $C0$ reporta um falta na cache, no ciclo t_2 solicita o barramento, em t_3 invalida as cópias e em t_4 escreve no bloco. Note que o SC de $C1$ deveria entrar no estágio de MM em t_4, como indicado pelo sinal `MM_is_SC`, entretanto, a invalidação por $C0$ no ciclo anterior faz com que cache $C1$ não seja ativada. O caso (iii) é equivalente a (i) mas o SC de $C1$ tem sucesso e o de $C0$ falha.

A situação (ii) deve ser analisada com um pouco mais de atenção e é dividida nos casos (a), (b) e (c). O caso (a) é mostrado na Figura 4.11 e acontece quando $C0$ e $C1$ executam a instrução SC ao mesmo tempo. No ciclo t_1 as duas caches verificam as etiquetas e constatam que o bloco está compartilhado. No ciclo t_2 requisitam o barramento e $C0$ ganha o acesso. Em t_3, a cache $C0$ invalida sua cópia e força o processador $C1$ invalidar o bit `LLBit` e abortar a referência corrente em $C1$. Em t_4, $C0$ escreve no bloco e $C1$ volta para o estado $IDLE$ liberando o processador para continuar sua execução, entretanto retornando a falha do SC.

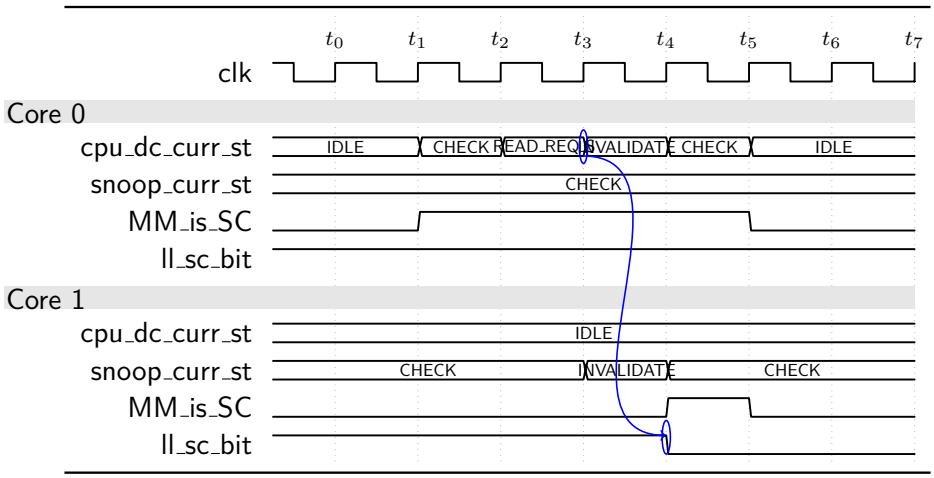

Figura 4.8: Diagrama de tempo da execução do caso (i)

No caso (b), mostrado na Figura 4.10, a cache $C1$ verifica a etiqueta um ciclo depois da verificação de $C0$. No ciclo t_1, $C0$ verifica as etiquetas e constata que o bloco está compartilhado. Em t_2, $C0$ requisita e ganha o barramento e $C1$ realiza verificação das etiquetas. Em t_3, a cache $C0$ modifica sua cópia e causa o processador acoplado a $C1$ invalidar o bit `LLBit`, enquanto $C1$ espera liberação do barramento. Em t_4, $C0$ escreve no bloco e $C1$ volta para o estado $IDLE$ liberando o processador para continuar sua execução, entretanto retornando a falha do SC.

No caso (c), mostrado na Figura 4.9, a cache $C1$ verifica a etiqueta dois ciclos depois da verificação de $C0$. No ciclo t_1, $C0$ verifica as etiquetas e constata que o bloco está compartilhado. Em t_2, $C0$ requisita e ganha o barramento. Em t_3, a cache $C0$ modifica sua cópia enquanto $C1$ está realizando verificação das etiquetas. Em t_4, $C0$ escreve no bloco e $C1$ volta para o estado ocioso.

Se $C1$ verifica a etiqueta três ou mais ciclos depois da verificação de $C0$ então ocorre a situação (i).

Somente durante os estados CHECK e BUS_REQ_S_TO_M, da máquina de estados do lado do processador, o sinal `cpu_abort_ref_i` precisa ser avaliado. Quando um SC é emitido na cache, o bloco deve estar nos estados MODIFICADO, EXCLUSIVO ou COMPARTILHADO, supondo que o LL correspondente tenha sido efetuado e o bloco apontado por `LLAddr` não tenha sido substituído. Um SC em um bloco no estado MODIFICADO ou EXCLUSIVO é resolvido durante o estado CHECK; e um SC em um bloco no estado COMPARTILHADO faz o controle, do lado do processador, percorrer os estados CHECK, BUS_REQ_S_TO_M e INVALIDATE para invalidar outras cópias.

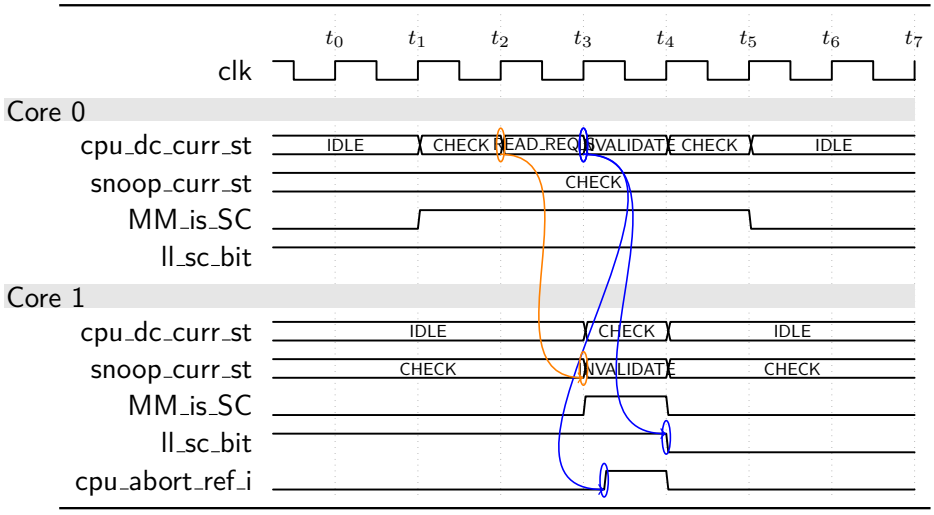

Figura 4.9: Diagrama de tempo da execução do caso (ii)-(a)

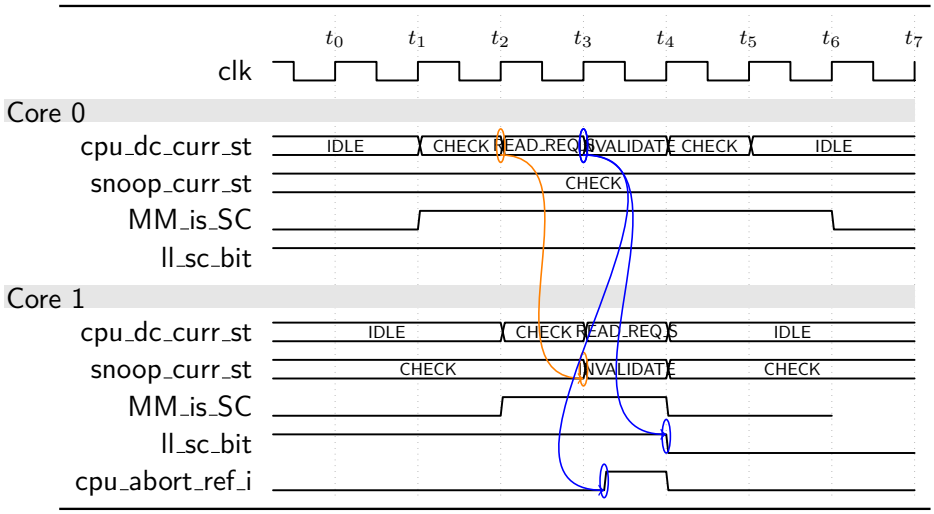

Figura 4.10: Diagrama de tempo da execução do caso (ii)-(b)

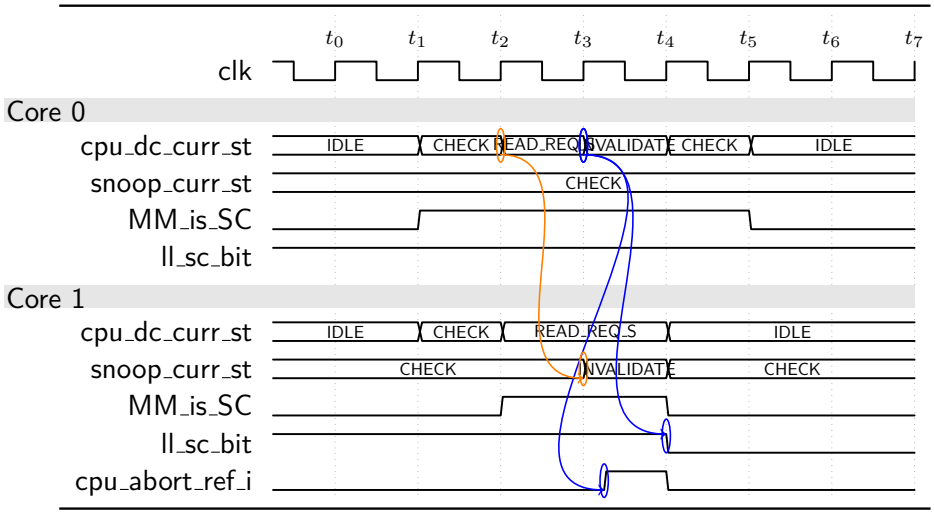

Figura 4.11: Diagrama de tempo da execução do caso (ii)-(c)

4.2 Síntese em lógica programável

Os resultados preliminares da síntese do multiprocessador num FPGA Altera Cyclone IV GX(EP4CGX150DF31C7) são mostrados na Tabela 4.1. A ferramenta *Quartus* II foi usada para a síntese. O multiprocessador foi configurado com memória de instruções e dados de 32 Kbytes (256 Kbits). A cache de dados foi configurada para conter 128 blocos de 4 palavras cada. Cada memória de estado/etiquetas armazena 128 palavras de 23 bits, totalizando 2944 bits. Cada cache de dados armazena 512 palavras, totalizando 2 Kbytes (16 Kbits). As memórias de dados e etiquetas de três portas foram implementadas usando dois blocos de RAM com duas portas cada, visto que o FPGA escolhido não suporta blocos de RAM com três portas. Para um núcleo esta configuração totaliza aproximadamente 69 Kbytes de memória no FPGA.

Núcleos	LUT's	registradores	bits de memória
1	6.386	1.410	564.992
2	12.147	2.710	867.840
4	25.137	5174	1.473.536
8	49.977	10.100	2.684.928

Tabela 4.1: Resultados da síntese no FPGA Altera Cyclone IV

Capítulo 5

Conclusão e Trabalhos Futuros

Este trabalho descreve a implementação do modelo em VHDL de um multiprocessador com memória compartilhada. O núcleo base do sistema é um processador MIPS de cinco estágios ligado a uma cache de dados privada, mantida coerente através do protocolo MESI. Os acessos à memória principal, às caches de dados e dispositivos de entrada e saída ocorrem através de um barramento compartilhado.

Um novo barramento é proposto para a implementação multiprocessada com o cMIPS. O barramento é atômico e é controlado por duas unidades: árbitro do barramento e árbitro de espionagem. A primeira controla a permissão de acessos usando a política *round-robin*; a segunda controla o barramento de dados, dando preferência a transações de cache para cache. Um trabalho futuro pode considerar a implementação de um barramento não atômico, no qual uma transação pode ser separada nas etapas de requisição e resposta. Esta modificação pode permitir uma maior utilização do barramento mas aumenta a complexidade da implementação.

Poucas modificações foram realizadas no núcleo cMIPS para acomodar a implementação *multicore*. A primeira delas foi uma consequência do uso de registradores na porta de entradas das memórias implementadas, o que levou a remoção de alguns sinais do registrador de pipeline EX/MM. A segunda modificação foi realizada no coprocessador 0 do cMIPS, na implementação das instruções LL/SC, para acomodar modificações realizadas por processadores remotos. O processador é responsável por monitorar o barramento para, possivelmente, invalidar um SC e abortar uma transação na cache de dados.

As caches de dados são diretamente mapeadas e implementam o procolo MESI usando a política de escrita preguiçosa (*write-back*). Capacidade e tamanho do bloco podem ser configuradas através de parâmetros *generics* no código VHDL. As memórias de dados, etiquetas

e estado foram implementadas com o cuidado de que o sintetizador faça o uso de memórias RAM internas denominadas *Block RAMs*.

O controle da cache de dados foi implementado por meio de duas máquinas de estados finitos responsáveis por atender aos estímulos do processador e do barramento. O controle contém todas as travas e adiamento necessários para manter o sistema de memória coerente sem comprometer o desempenho do sistema. Existe espaço para algumas otimizações no controle das caches de dados para reduzir os custos dos eventos associados ao protocolo de coerência.

A implementação do protocolo MESI foi adaptada para permitir a transferência de um bloco MODIFICADO de cache para cache. Se a cache fonte de um bloco estiver no estado MODIFICADO, então a cache que recebe o bloco cache o armazenará no estado MODIFICADO, inibindo a necessidade de atualização da memória principal. A modificação permitiu simplificar o controle da cache mas não foi investigado o impacto no desempenho que ela provoca. Se consideramos espera ocupada, é possível que um bloco, no estado MODIFICADO, fique sendo transferido de cache para cache a cada leitura de um processador, causando tráfego desnecessário no barramento. Se o bloco fosse colocado na memória principal, então os processares realizando espera ocupada manteriam o bloco no estado COMPARTILHADO e não causariam transações no barramento durante a espera.

A maior dificuldade encontrada na realização do trabalho foi a grande complexidade dos testes necessários para garantir a corretude funcional da implementação. Em alguns casos não foi possível implementar uma rotina automatizada de testes, e a verificação ocorre através da análise dos diagramas de tempo: um processo manual, cansativo e propício a erros. Outra dificuldade encontrada foi a demora das simulações em sistemas com mais de dois núcleos.

A implementação do multiprocessador apresenta um correto funcionamento nas simulações realizadas. Para ser avaliada de forma mais completa é necessário reproduzir o modelo em um FPGA. Pretende-se desenvolver controladores de periféricos de entrada/saída e um controlador de SDRAM para sintetizar o sistema multiprocessado em lógica programável. O resultado esperado é uma aplicação funcional em um FPGA.

Dentre outras perspectivas de trabalhos futuros está a análise de desempenho da implementação multiprocessada. O objetivo desta análise é avaliar o impacto da implementação de um barramento não atômico, outras configurações da cache de dados, e modificações no circuito de controle exigido pelo protocolo MESI.

O multiprocessador pode ser usado na prática de docência em disciplinas de Arquitetura de Computadores e disciplinas correlatas. O tópico de processamento paralelo, quando aborda de coerência entre caches e sincronismo entre processos demanda um grande esforço intelectual para compreensão. Neste sentido, a prática com um multiprocessador pode acelerar o entendimento destes tópicos. O modelo também abre espaço para modificações do hardware e implementações de software, como por exemplo, a implementação ou porte de um sistema operacional para o ambiente multiprocessado.

O código VHDL resultante deste trabalho está disponibilizado como código livre em https://gitlab.c3sl.ufpr.br/rams11/cMIPS-multicore.

Referências Bibliográficas

[1] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[2] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[3] Roberto A. Hexsel and Renato Carmo. cMIPS – uma ferramenta pedagógica para o estudo de arquitetura. *Workshop sobre Educação em Arquitetura de Computadores*, 12(3):348–354, January 2013.

[4] Jorge Tortato Junior. Projeto e implementação de multiprocessador embarcado em dispositivos lógicos programáveis. Master's thesis, 2009.

[5] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[6] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[7] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2007.

[8] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.