CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
LABORATÓRIO DE ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES II - Turma 01

PRÁTICA 1 - HIERARQUIA DE MEMÓRIA

Natan Rodrigues Rocha / 20203013242
Gabriel Andrade Quezada / 20183021240

Belo Horizonte
08 de Abril de 2024

1. INTRODUÇÃO

Neste projeto, utilizamos a plataforma Quartus II para implementar um módulo de memória RAM. O módulo foi criado utilizando a biblioteca LPM. As entradas do sistema incluem:

1. **Endereço**: Determina a posição da memória que será acessada.
2. **Vetor de Dados**: Contém os dados a serem armazenados ou lidos.
3. **Chave de Escrita**: Habilita ou desabilita a escrita na memória.
4. **Botão de Clock do Sistema**: Controla o ciclo de clock.

Na placa DE2, utilizamos os displays para mostrar informações relevantes, como o endereço, o dado inserido e o dado lido da memória. Para testar o funcionamento do sistema, empregamos a ferramenta ModelSim.

2. OBJETIVO

Esta prática tem a finalidade de exercitar os conceitos relacionados à hierarquia de memória e relembrar como se utiliza a placa FPGA

3. PARTE 1

3.1. DESENVOLVIMENTO

Afim de executar a parte 1, elaboramos o módulo memória onde possui as seguintes entradas e saídas:

Entradas:
SW: vetor de 18 bits representando os interruptores.

Saídas:
HEX0 a HEX7: displays de 7 segmentos.
LEDG: LEDs verdes indicadores.
LEDR: LEDs vermelhos indicadores.

O módulo memória é composto pelas seguintes partes:

- **Geração do Sinal de Clock**: Utiliza o bit mais significativo dos interruptores (SW[17]) para gerar o sinal de clock. Este sinal é exibido em um par de LEDs verdes (LEDG[1] e LEDG[0]) para indicar o estado do clock.

- **Controle de Escrita na Memória**: O bit SW[16] é usado como habilitador de escrita (wren). Um LED vermelho (LEDR[0]) é acionado quando a escrita é desabilitada, e um LED verde (LEDG[7]) é acionado quando a escrita é habilitada.

- **Dados e Endereço**: Os dados a serem escritos na memória são definidos pelos interruptores SW[13:6], e o endereço da posição de memória é determinado pelos interruptores SW[4:0].

- **Instância do Módulo de Memória RAM**: Utiliza o módulo ramlpm, que é uma instância da biblioteca LPM, para implementar a memória RAM. Este módulo recebe o endereço, o sinal de clock, os dados e o habilitador de escrita como entradas, e fornece os dados armazenados na memória como saída.

- **Displays de 7 Segmentos**: São utilizados para exibir os dados lidos da memória (HEX0 e HEX1), o endereço da posição de memória (HEX4 e HEX5), e os dados escritos na memória (HEX6 e HEX7).

3.2. SIMULAÇÃO

Durante a simulação da primeira etapa do projeto, o foco foi na escrita em duas posições de memória distintas, seguida da leitura dessas posições. As posições selecionadas foram determinadas aleatoriamente, sendo atribuídas à memória as posições 1 e 4, com valores correspondentes de 7 e 4, respectivamente.

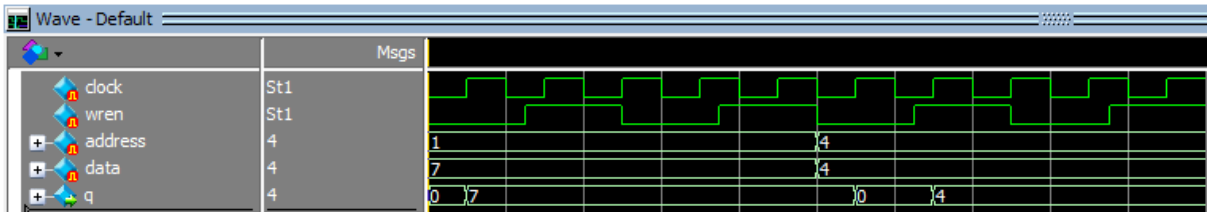

Figura 01 - Simulação da parte 1 no ModelSim

Ao escolhermos a posição 1 e o valor 7, observamos que quando o sinal de escrita é ativado (assumindo o valor 1), o valor 7 é armazenado na posição 1 da memória após um ciclo de clock. Da mesma forma, ao selecionar a posição 4 e o valor 4, o processo se repete, resultando no armazenamento do valor 4 na posição 4 da memória após um ciclo de clock.

4. PARTE 2

4.1. DESENVOLVIMENTO

Para o desenvolvimento da segunda parte do projeto, inicializamos a memória utilizando um arquivo MIF (Memory Initialization File). Este arquivo contém os dados que queremos armazenar na memória em seus respectivos endereços, permitindo a leitura e escrita de dados na memória, com exibição dos dados nos displays de 7 segmentos.

Segue o arquivo MIF abaixo:

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0	1	2	3	4	5	6	7	.....
8	0	0	0	0	0	0	0	0	.....
16	0	0	0	0	0	0	0	0	.....
24	0	0	0	0	0	0	0	0	.....

Figura 02 - Memoria.mif

4.2. SIMULAÇÃO

Na segunda parte do projeto, conduzimos uma simulação que envolve tanto operações de leitura quanto de escrita na memória. Esta memória foi inicializada por meio de um arquivo MIF.

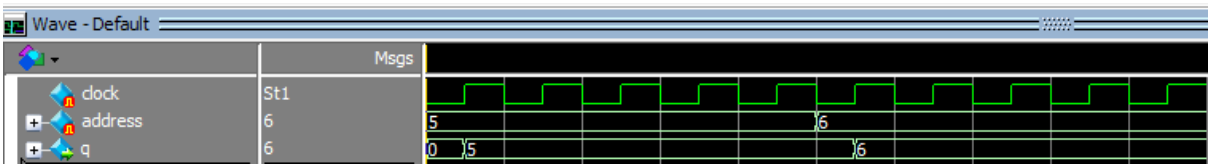

Figura 03 - Simulação da parte 2 no ModelSim

Ao realizar a leitura das posições 5 e 6 da memória, confirmamos que os valores previamente salvos nessas posições pelo arquivo MIF estão sendo lidos corretamente. Isso indica que a inicialização da memória através do arquivo MIF foi bem-sucedida e os dados estão sendo armazenados e recuperados conforme esperado durante a simulação. Essa validação reforça a integridade e a precisão do sistema implementado, garantindo a confiabilidade das operações de leitura e escrita na memória.

5. PARTE 3

5.1. DESENVOLVIMENTO

Na implementação da terceira parte, a arquitetura da cache foi definida como associativa em um conjunto de duas vias, o que significa que cada conjunto na

cache tem duas linhas de armazenamento, permitindo que cada bloco de memória possa ser mapeado para qualquer uma das duas linhas dentro desse conjunto. Por outro lado, a memória foi configurada como diretamente mapeada, onde cada bloco de memória na RAM é mapeado para exatamente uma linha específica na cache. Essa abordagem simplifica o processo de busca na cache, pois permite uma correspondência direta entre blocos de memória e linhas na cache, facilitando o acesso e a gestão dos dados entre os dois módulos.

.

O módulo Cache possui entradas para o clock, sinal de escrita (**wren**), sinal de reset (**reset**), endereço (**address**), dados (**data**), e saídas para os dados lidos (**q**), sinal de hit (**hit**), estado do processamento (**state**), entre outros.

A lógica do módulo Cache é dividida em três principais estados de processamento: HIT, MISS e Atualização LRU. No estado de HIT, verifica-se se o dado está presente na cache. No estado de MISS, ocorre a busca do dado na memória RAM e sua inserção na cache. E no estado de Atualização LRU, é atualizada a idade dos blocos de cache de acordo com a política LRU.

Além disso, são realizadas operações de leitura e escrita nos blocos de cache, atualização do bit de controle Dirty, e controle do estado da cache.

A cache implementada no sistema segue uma estrutura organizacional específica para armazenar e gerenciar os dados de forma eficiente. A organização da cache é crucial para otimizar o desempenho do sistema, reduzindo os tempos de acesso à memória principal e melhorando a taxa de acertos da cache.

Estrutura de Cada Entrada na Cache
Cada entrada na cache possui os seguintes campos:

- Dados Armazenados ([7:0]): Esses 8 bits representam os dados armazenados na cache. A cache é projetada para armazenar parte dos dados presentes na memória principal, permitindo acesso mais rápido a esses dados.

- Tag do Endereço ([12:8]): A tag do endereço consiste em 5 bits e é utilizada para identificar exclusivamente o bloco de memória principal associado ao bloco presente na cache. Essa tag é comparada com o endereço de acesso para determinar se o bloco está presente na cache.

- Bit de Controle LRU ([14:13]): Este campo, composto por 2 bits, controla a política de substituição adotada pela cache. A política de LRU (Least Recently Used) é implementada para determinar qual bloco deve ser substituído quando a cache está cheia e uma nova entrada precisa ser

armazenada. Os valores variam de 00 (indicando o bloco mais antigo) a 11 (indicando o bloco mais recente).

- Bit de Controle Dirty ([15]): O bit de controle Dirty é utilizado para indicar se os dados presentes no bloco de cache foram modificados desde que foram trazidos da memória principal. Isso é importante para estratégias de escrita de volta (write-back), onde os dados são escritos na memória principal apenas quando o bloco de cache é substituído.

- Bit de Controle Valid ([16]): Este bit indica se o bloco de cache contém dados válidos. Quando um bloco é trazido da memória principal para a cache, este bit é ativado para indicar que os dados na cache estão atualizados e podem ser utilizados. Se estiver desativado, os dados no bloco de cache não são válidos e devem ser buscados na memória principal.

O módulo memoria gerencia a comunicação entre a cache e a memória RAM. Ele recebe como entrada os sinais provenientes da cache, como endereço (**SW[15:11]**), sinal de escrita (**SW[17]**), dados a serem escritos na RAM (**SW[7:0]**), e ativação do clock (**KEY[0]**).

A memória RAM é acessada quando ocorre um MISS na cache. Os dados lidos da memória RAM são então armazenados na cache para acessos futuros. O módulo memoria também é responsável por atualizar os displays e LEDs de acordo com o estado do sistema.

Para o desenvolvimento da terceira parte do projeto, inicializamos a memória utilizando um arquivo MIF.

Segue o arquivo .MIF abaixo:

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0	0	0	0	0	0	0	0	.....
8	0	0	0	0	0	0	0	0	.....
16	0	0	0	0	8	6	4	6	.....
24	8	8	6	3	8	9	7	9	.....

Figura 04 - Memoria.mif

A tabela abaixo mostra os valores iniciais atribuídos a cada entrada da cache durante a fase de inicialização. Esses valores representam o estado inicial da cache antes de qualquer acesso ou operação de escrita.

BLOCO	VALID	DIRTY	LRU	TAG	DADO
1	1	0	0	20	6
	0	0		22	2
2	1	0	1	25	6
	1	0		21	4
3	0	0	3	10	4
	1	0		23	2
4	1	0	2	4	4
	0	0		3	2

Tabela 01 - Inicialização da Cache

5.2. SIMULAÇÃO

Teste read HIT: Verificar se a posição acessada está mapeada na cache e, em caso afirmativo, realizar a leitura dos dados dessa posição. Em seguida, verificar se há atualização no LRU.

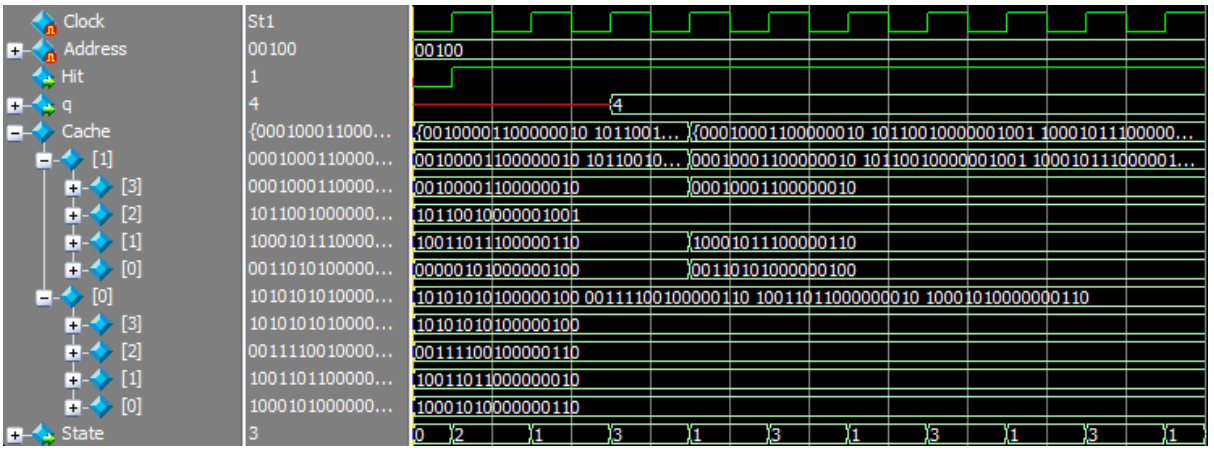

Figura 05 - Simulação do teste read HIT no ModelSim

Na simulação acessamos o endereço de Tag 4(em decimal ou 0100 em binário), que está presente na posição 6 cache como mostrado acima. Portanto, ao acessar esse endereço podemos perceber que ocorreu um hit na cache,

logo o sinal de “hit” fica com o valor 1, e o valor salvo no endereço de Tag 4 é retornado, no caso esse valor é 4 (decimal) também. Por fim, após o acesso a cache os bits de LRU (bits 14:13) são atualizados, e podemos perceber então que o LRU da posição da cache onde está armazenado o endereço de Tag 4 era 2 (decimal) antes do acesso, e passa a ser 3 (decimal) depois do acesso. Entretanto, por algum lixo de memória, o estado que era para sair do 0 para 1, visto que foi um HIT, vai para o 2 antes de ir para o 1. É perceptível, porém, que após o estado 1, ele vai para o estado 3, atualizando o LRU e mantendo um loop após vários clocks em sequência, pois o HIT se manteve. Sendo assim, mesmo com essa inconveniência no sistema, o comportamento foi correto e esperado.

Teste read MISS: Verificar se houve uma falha (miss) na cache ao acessar a posição desejada. Em seguida, testar se a posição de memória desejada foi acessada corretamente na RAM e em qRAM. Além disso, verificar se a cache foi atualizada corretamente, incluindo a atualização dos bits LRU, Valid, Dirty, Tag e Dado. Por fim, analisar se ocorreu um acerto (hit) na cache.

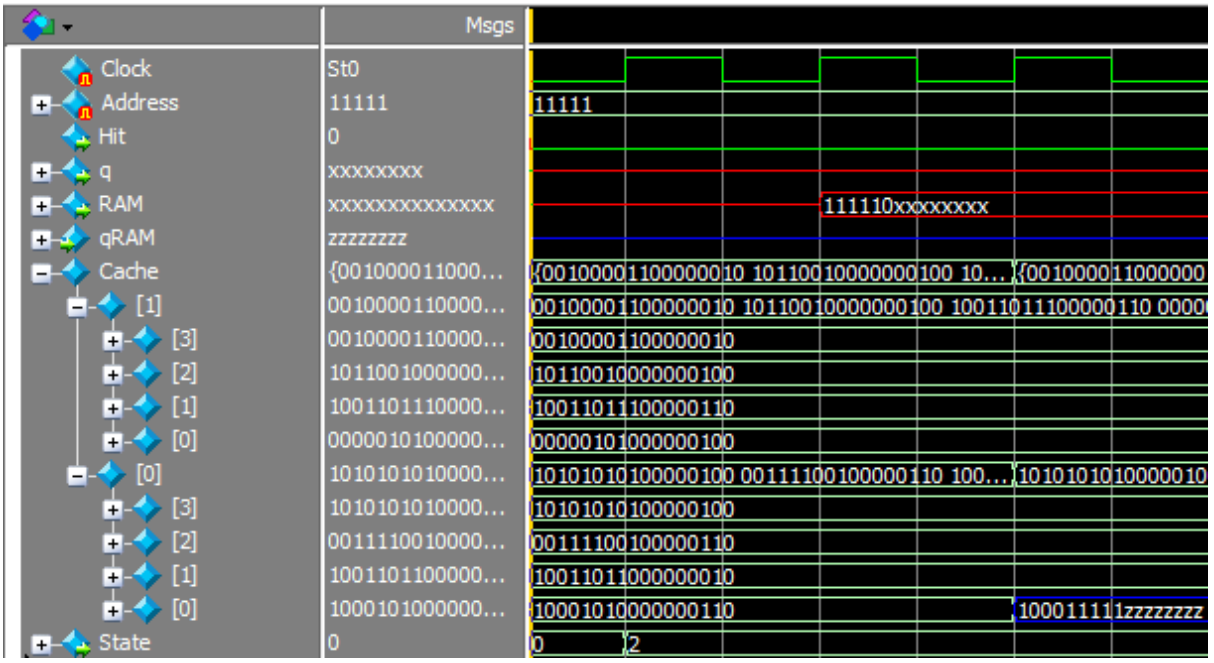

Figura 06 - Simulação do teste read MISS no ModelSim

Nesse teste, decidimos acessar o endereço de TAG 31, como visto na figura 06, que é um endereço que não está na cache. Portanto, num primeiro momento é indicado um miss e o sinal de “hit” recebe valor 0, em seguida a variável RAM recebe o endereço de Tag 31 para que ele possa ser acessado na memória principal e trazido para a cache. Além disso, o estado sai do 0 para o 2, indicando realmente o miss. Podemos analisar que o bloco presente na posição da cache que possui os bits de LRU com valor 0 (decimal) é retirado e o bloco referente ao endereço de Tag 31 é inserido no lugar. Após isso, é indicado um hit na cache e o

valor presente no endereço de Tag 31 é indicado na saída da cache, demonstrando o funcionamento correto e esperado do sistema.

Teste write HIT: Verificar se a posição acessada está mapeada na cache e, se estiver, realizar a escrita dos dados fornecidos na entrada na posição correspondente. Após a escrita, verificar se houve atualização nos bits Dirty e LRU da cache.

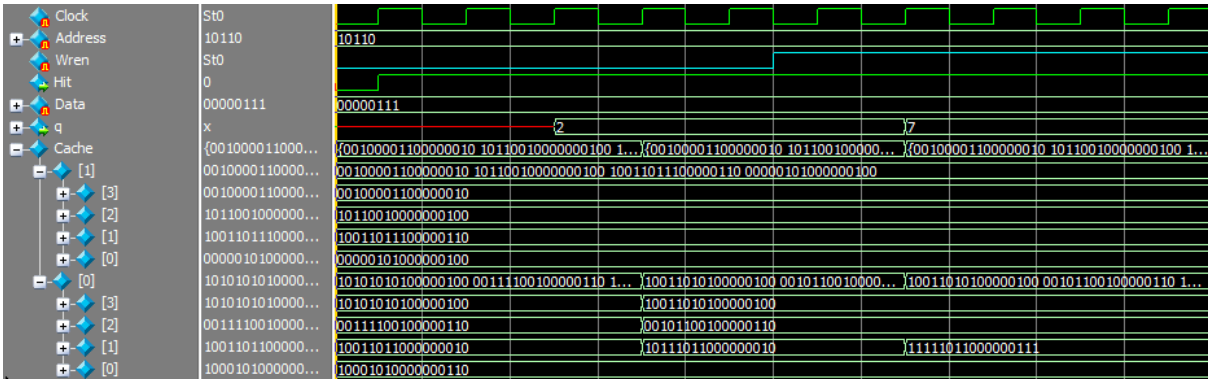

Figura 07 - Simulação do teste write HIT no ModelSim

Na simulação da figura 07, foi acessado o endereço de Tag 22, que está armazenado na posição 1 da cache, como indicado anteriormente. Inicialmente, realizou-se uma leitura dessa posição e, em seguida, uma escrita na mesma posição. A primeira parte da simulação envolve apenas a leitura da posição, conforme descrito no teste anterior. Na segunda parte, ocorre a escrita na posição. Ao acessar esse endereço, observa-se um acerto (hit) na cache, indicado pelo valor 1 do sinal "hit", e o valor armazenado no endereço de Tag 22 é retornado. Posteriormente, o sinal de escrita é ativado, representado por "wren" recebendo o valor 1, e um dado é fornecido como entrada - neste caso, o valor escolhido foi 7 (decimal) para escrever na posição selecionada. Com o sinal de escrita ativo e o dado fornecido como entrada, observa-se que o valor de "q", que indica a saída da cache, muda de 2 para 7, indicando que a escrita foi realizada com sucesso na posição 1 da cache. Por fim, nota-se que o bit Dirty (bit 15) da posição onde ocorreu a escrita teve seu valor alterado de 0 para 1. Entretanto, percebemos que ainda que ocorresse tudo bem na simulação, na lógica da implementação do writeback, a verificação do Valid/Dirty juntamente com o bloco estava com problemas e como não conseguimos fixar, decidimos deixar assim mesmo.

Teste write MISS: Verificar se houve um miss na cache ao acessar a posição desejada. Em seguida, verificar se os dados fornecidos para serem salvos na memória (RAM) estão corretos. Além disso, testar se a posição de memória desejada foi acessada corretamente na RAM e em qRAM. Posteriormente,

verificar se a cache foi atualizada corretamente, incluindo os bits LRU, Valid, Dirty, Tag e Dado. Por fim, analisar se ocorreu um acerto (hit) na cache.

Como já estávamos com dificuldade de implementar a lógica do writeback após o write hit, nesse teste para write miss, enfrentamos mais problemas, visto que nossa simulação estava funcionando como um read miss, ele num primeiro momento é indicado um miss e o sinal de "hit" recebe valor 0, o estado sai de 0 para 2 confirmando o MISS, mas fica nisso num loop. Segundo o que testamos na FPGA, até conseguíamos escrever algo por cima, mas ele não salvava nenhum valor e nos retornava o dado que já estava presente anteriormente em qualquer tag, mesmo sendo inválido ou estando sujo(Dirty). Concluímos que como esse caso é o mais complexo de todos, nossa falha na leitura das duas vias por meio de blocos contribuiu para que a simulação se assemelhasse a um read miss.

6. CONCLUSÃO

A partir da análise das simulações realizadas nas diferentes etapas do projeto, podemos concluir que houve uma abordagem sistemática e abrangente na verificação e validação do funcionamento do sistema de cache implementado.

Na primeira etapa do projeto, foi realizada uma simulação focada na escrita e leitura de posições de memória específicas. Através dessas operações, foi possível confirmar que o sistema estava armazenando e recuperando corretamente os dados nas posições indicadas. Além disso, a utilização de valores aleatórios para as posições e dados demonstrou a capacidade do sistema de manipular diferentes entradas de forma eficaz.

Na segunda parte, a validação da inicialização da memória por meio de um arquivo MIF foi fundamental para garantir que os dados estivessem sendo armazenados e recuperados conforme o esperado durante a simulação. Isso contribuiu para a confiabilidade das operações de leitura e escrita na memória, reforçando a integridade do sistema implementado.

Na terceira parte, os testes de read HIT e read MISS demonstraram a capacidade da cache de identificar corretamente se um endereço está presente na cache ou não, e de realizar as operações de leitura correspondentes de acordo com o resultado obtido. Apesar de algumas limitações identificadas durante a simulação, como a inconveniência na atualização do estado LRU em alguns casos, o comportamento geral do sistema foi considerado correto e esperado.

Já nos testes de write HIT e write MISS, embora tenha sido possível verificar o acerto ou falha na cache e realizar operações de escrita, algumas dificuldades

foram encontradas, especialmente no tratamento dos casos de write MISS. Problemas na lógica de implementação do writeback após um write HIT e a complexidade do processo de write MISS contribuíram para desafios adicionais na simulação.

Gostaríamos de destacar alguns pontos específicos que surgiram durante a realização do projeto:

- Achamos que o salto da Parte 1 e 2 para a 3 foi considerável, portanto demoramos a entender o problema, a desenhar no papel o sistema da cache de duas vias com o percorrimento de blocos e a construir o programa/lógica no verilog. O que consequentemente acarretou uma apresentação incompleta na FPGA e a uma falta de solução dos problemas para a simulação.

- Nossa implementação inicial de percorrer e manusear as duas vias estavam com a lógica incorreta, ao ponto que tivemos que ter que retrabalhar a lógica dos blocos e sua implementação, seguindo dicas da professora e desenhos no laboratório.

- A implementação da checagem dos blocos com um “indexCache = address[2:1] << 1;” antes da checagem das duas vias nunca saíram do papel, visto a dificuldade de implementar essa lógica em verilog. Com testes dando errado e falta de tempo, optamos por deixar nosso trabalho incompleto, como visto na apresentação na FPGA, ainda que com algumas melhorias, apresentadas na simulação no ModelSim.

Apesar dos desafios encontrados durante as diferentes etapas do projeto e das dificuldades na implementação prática da cache de duas vias, é evidente que houve um esforço significativo na verificação e validação do sistema. A análise minuciosa das simulações permitiu identificar pontos de melhoria e compreender melhor os conceitos subjacentes à arquitetura de cache. Embora o resultado final possa não ter sido totalmente funcional, a experiência adquirida ao lidar com esses desafios certamente proporcionará uma base sólida para enfrentar futuros projetos de programação em Verilog.