

ESCOLA POLITÉCNICA - USP

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E
SISTEMAS DIGITAIS

PCS3422: ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES II

Hierarquia de Memória com Cache Multinível

Autores:

Douglas RAMOS

NUSP: 9853300

Henrique HATTORI

NUSP: 9837330

Igor ORTEGA

NUSP: 9763071

Lucas KOGACHI

NUSP: 9016960

Rafael HIGA

NUSP: 9836878

Professor:

Bruno ALBERTINI

Turma: 01 Grupo: A



25 de novembro de 2019

Sumário

1	Introdução	3
2	Estudos Sobre Otimização de Cache	4
2.1	Quantidade de níveis em uma hierarquia de memória	5
2.2	Associatividade	7
2.3	Política para armazenamento de blocos para os níveis de cache	8
2.4	Tamanho dos Blocos	14
2.5	Cache Replacement Policy	15
2.6	Write-Buffer	17
2.7	Técnicas avançadas de otimização de cache	17
2.7.1	Diminuição do hit time - Caches L1 mais simples e pequenos	19
2.7.2	Aumento da cache bandwidth - Multibanked Cache	21
2.7.3	Critical word first and early start	22
2.7.4	Merging Write-buffers	22
2.7.5	Hardware Prefetching	23
2.7.6	Nonblocking Cache	24
3	Conclusão da Pesquisa e Parâmetros do Projeto	26
4	Implementação do Projeto	27
4.1	Hierarquia Básica	27
4.1.1	Cache L1 - Instruções	27
4.1.2	Cache L1 - Dados	31
4.1.3	Memória L2	36

4.2	Hierarquia Otimizada	37
4.2.1	Caches L1	38
4.2.2	Cache L2	38
4.2.3	Memória L3	43
4.3	Victim Buffer	44
4.3.1	Descrição	44
4.3.2	Interface	44
4.3.3	Implementação	44
4.3.4	Unidade de Controle	45
4.4	Módulo Tester	48
4.4.1	Descrição	48
4.4.2	Interface	48
4.4.3	How To Use	50
4.4.4	Exemplo de funcionamento do Tester	51
5	Testes Executados	54
5.1	Teste 1: Hierarquia Básica	54
5.2	Teste 2: Hierarquia Otimizada	55
5.3	Teste 3: Hierarquia Otimizada	57
5.4	Teste 4: Hierarquia Básica	58
6	Conclusão	60
Referências		60

1 Introdução

O presente relatório documenta o desenvolvimento do projeto de tema “otimização de hierarquia de memória”, que objetivou o estudo de técnicas de otimização de cache e da definição da quantidade adequada de níveis de hierarquia de memória para esse projeto; e a implementação em VHDL da hierarquia de memória, verificando-se o efeito das técnicas de otimização implementadas.

Hierarquias de memória são encontradas em quase todas as máquinas modernas e tem o intuito de melhorar o desempenho dos computadores. Em suma, trata-se de uma distinção entre o armazenamento das instruções e dos dados de maior probabilidade de uso e os de menor probabilidade, guardando de forma temporária as informações mais relevantes no cache. Isso possibilita a utilização de aparelhos de melhor performance, de forma que, quando essas informações forem acessadas, ela o sejam da maneira mais eficiente.

A primeira hierarquia, chamada de Hierarquia básica consiste de dois níveis. O primeiro nível, mais próximo da CPU, consiste em duas memórias caches, uma destinada para fetch de instruções e outra para leitura ou escrita de dados. O segundo nível é, com efeito, a memória principal, que é inicializada com valores a partir de um arquivo .dat.

A segunda hierarquia, a Hierarquia Otimizada, possui três níveis, introduzindo um cache nível 2 (cacheL2). Esta, além implementar um novo nível, conta ainda com outros tipos de otimizações para os caches.

Inicialmente, um estudo acerca das diversas possibilidades de otimização de caches em hierarquia de memória foi feito, do qual algumas das técnicas foram escolhidas e, enfim, daí se obteve uma descrição VHDL e simulações comprobatórias do efeito das otimizações de cache implementadas.

Aplicando-se associatividade aos caches; definindo-se uma hierarquia de memória de dois níveis de cache - de maneira devidamente embasada, conforme a seção 3; e se implementando a política de exclusão de blocos entre os caches L1 e L2, gerou-se uma “hierarquia avançada”, detalhada na seção 4, sendo assim possível sua comparação com a hierarquia básica.

A seção 2 do presente relatório descreve a pesquisa de técnicas de otimização realizada - das vantagens e desvantagens de diversas técnicas e da fundamentação das escolhas para o dado projeto.

Já a seção 3 documenta a implementação do projeto - os parâmetros escolhidos para os caches, as decisões de projeto tomadas, e os respectivos testes elaborados em prol da validação do projeto.

A seção 4 detalha, passo a passo, component a component, o devido uso e significado

de cada módulo, interface e sinal presentes no projeto, bem como a ordem de execução de simulações e a configuração de interfaces necessária para a execução dos testes realizados, enquanto a seção 5 conta com os testes realizados.

Enfim, na seção 6 são traçadas as considerações a respeito dos resultados obtidos e as respectivas conclusões às quais o grupo atingiu.

2 Estudos Sobre Otimização de Cache

A otimização do desempenho de uma hierarquia de memória se dá sob diversas formas distintas; e também sob a forma da tomada de decisões de projeto que se fazem inerentes ao mesmo, mas que, de acordo com as especificações do projeto¹ (i.e., tamanho do cache, quantidade de níveis de memória, latência desejada para a resolução de requisições de endereços por parte do processador, miss rate, etc), tornam uma determinada especificação mais ou menos adequada.

Alguns parâmetros são relevantes dentre a problemática da otimização do desempenho de uma hierarquia de memória, sobretudo:

- **Miss Rate:** taxa de misses que ocorre na execução dos programas em um dado dispositivo de memória (um cache, ou até a hierarquia de caches como um todo, pensando de uma forma global);
- **Miss Penalty:** o tempo o qual (devido a um miss de acesso a endereço em um determinado nível L da memória) é despendido para a transferência desse bloco de um nível inferior L+1 ao nível L, possibilitando assim a leitura do endereço requisitado.
- **Memory Stall Cycles:** Número de misses x miss penalty. Consiste no número de ciclos de clock perdidos devido a um stall. Para uma análise mais complexa, a fórmula pode ser estendida para:

$$MemoryStallCycle = IC \times ReadsPerInstruction \times MissPenalty \quad (1)$$

contabilizando assim, separadamente, os misses de escrita e de leitura.

- **Average Access Time:** O tempo de acesso de um endereço em um nível de memória é definido por:

$$AverageMemoryAccessTime = HitTime + MissRate \times MissPenalty \quad (2)$$

¹a exemplo da arquitetura ARM básica projetada no início do semestre, os estágios ID e MEM contam com acesso aos caches L1, consistindo no estágio de pipeline cuja atividade era a mais longa, definindo assim a frequência do ciclo de clock utilizado no processador pipeline, sendo assim vantajoso que tal cache L1 possuísse um baixo hit time.

Caso venha a haver uma hierarquia de dois níveis de cache, basta considerar o miss penalty do cache L2:

$$HitTime_{L1} + MissRate_{L1} \times (HitTime_{L2} + MissRate_{L2} \times MissPenalty_{L2}) \quad (3)$$

- **Consumo de energia:** está atrelado sobretudo ao uso de hardwares mais sofisticados. Frequentemente, técnicas de otimização se deparam com o tradeoff entre o uso de um software mais complexo, que permita a obtenção, por exemplo, de menores miss rates, e o aumento do consumo de energia.

Para o presente projeto, como será descrito mais detalhadamente no capítulo de projeto deste relatório, dentre essas medidas de desempenho apresentadas, utilizou-se como métricas sobretudo o miss rate, o miss penalty e o average access time.

As otimizações de hierarquia de memória, em geral, visam por otimizar um dos parâmetros determinados acima, porém, podem acarretar na piora dos índices de outro fator de desempenho.

As subseções seguintes destacam os resultados das pesquisas sobre técnicas de associatividade exercidas pelo grupo. Intenta-se por apresentar um panorama resumido da problemática, destacando os conceitos e o entendimento por parte do grupo sobre os mesmos, sem explicações muito complexas e presumindo a ciência por parte de termos básicos envolvendo o tema, como bloco, index, tag, tipos de miss (3 C's), etc.

2.1 Quantidade de níveis em uma hierarquia de memória

Em uma hierarquia de memória, existem níveis mais próximos ao processador e níveis mais próximos à memória RAM. Ao cache próximo ao processador, é interessante que os acessos à memória sejam rápidos, ao passo que ao cache próximo à memória RAM, quanto menos acessos à mesma sejam feitos, menor é o tempo gasto no cumprimento de miss penalties. Essa dualidade de comportamento corrobora com a presença de dois níveis de cache em uma hierarquia de memória.

Outra questão pertinente à dada problemática é a de se mais níveis se fazem necessários. Em primeira instância, é evidente que quanto mais níveis forem implementados, maior é o espaço de memória na hierarquia dos caches e menor é o miss rate global no sentido de minimizar os acessos à memória RAM. Por outro lado, a adoção de mais níveis também implica em custos financeiros para o projeto, além do fato de que os ganhos em diminuição dos miss rates de acesso à memória tendem a ficar pequenos de forma a se tornarem desproporcionais com os gastos despendidos com o tal novo nível de cache.

Em [3] e [4], destaca-se que a utilização de um cache L3 (e atualmente até mesmo caches L4 são usados) é característica sobretudo de aplicações em processadores com múltiplos

núcleos. O cache L3 costuma ser um cache de tamanho significativamente grande (por exemplo, 8 MB em um processador core i7), compartilhado com os diversos núcleos com os quais se comunica.

Em multiprocessadores, é interessante haver níveis de memória “globais”, para simplificar a lógica do cache. Evita-se tanto o acesso simultâneo à memória RAM como a replicação de dados em hierarquias de cache distintas (por estarem isoladas, os núcleos acabam “não se conversando”), lidando melhor com a coerência dos blocos.

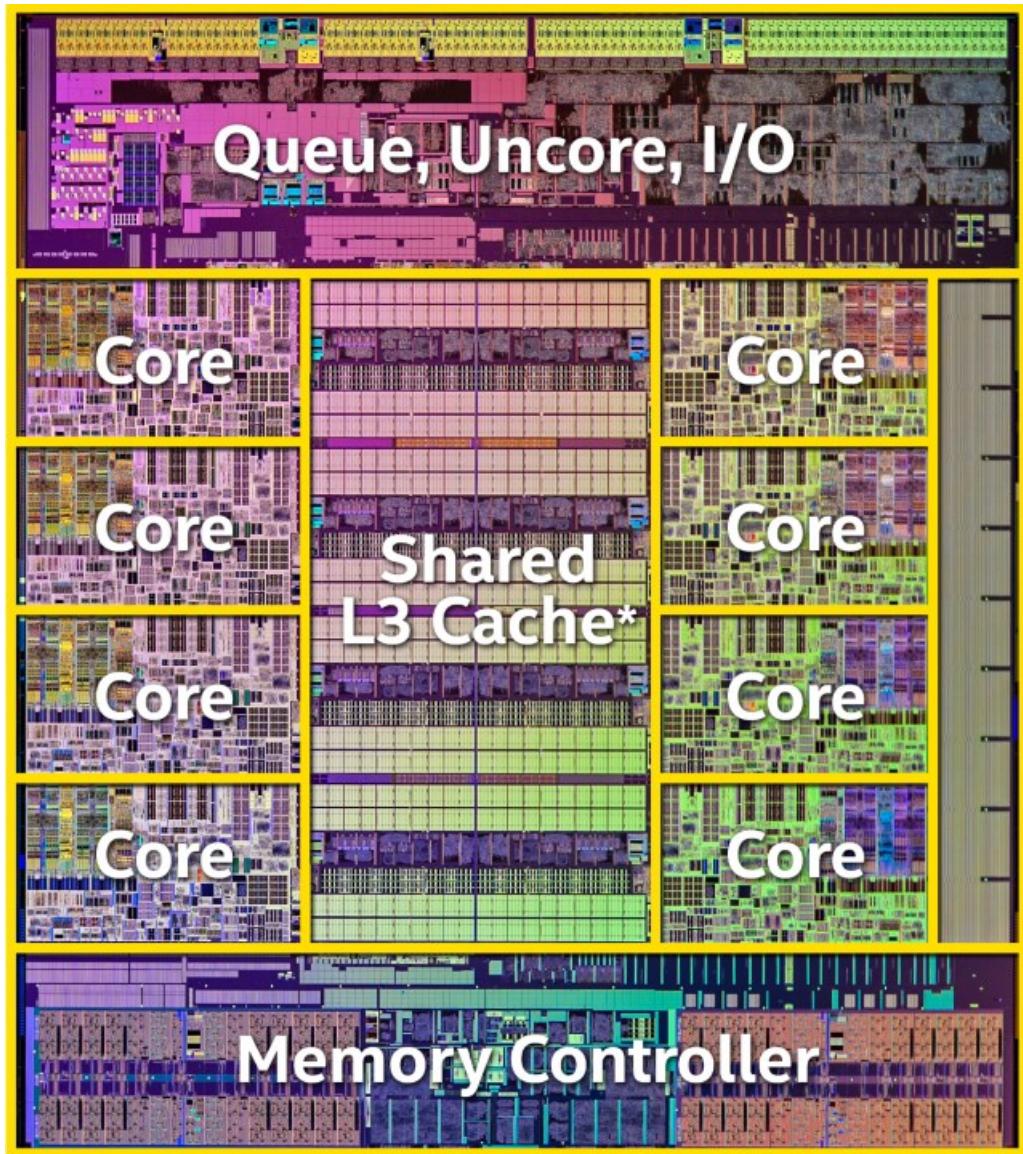


Figura 1: Processador dotado de um cache L3, compartilhado entre seus diversos núcleos. Extraído de [4].

Ainda discutindo sobre o tamanho de caches, segue também um pequeno comparativo entre três modelos de processadores, na figura 2, e algumas características dos três níveis que possuem. É interessante notar os já mencionados 8 MB a que chega o tamanho do L3, justo, uma vez que comporta dados de diversos núcleos; e também os tamanhos de caches de níveis superiores: o cache L2 tem cerca de 1 a 2 MB, no melhor dos casos cerca de 4 vezes menos capacidade de armazenamento que um cache L3. Sobre o cache L1, nem chega a possuir 1 MB.

Cache	Bulldozer	Piledriver	Steamroller
Level 1 code	64 kB, 2-way, 64 B line size, shared between two cores.	64 kB, 2-way, 64 B line size, shared between two cores.	96 kB, 3-way, 64 B line size, shared between two cores.
Level 1 data	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.	16 kB, 4-way, 64 B line size, per core. Latency 3-4 clocks.
Level 2	1 - 2 MB, 16-way, 64 B line size, shared between two cores. Latency 21 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 20 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 12 clock.	2 MB, 16-way, 64 B line size, shared between two cores. Latency 19 clocks. Read throughput 1 per 4 clock. Write throughput 1 per 6 clock.
Level 3	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	0 - 8 MB, 64-way, 64 B line size, shared between all cores. Latency 87 clock. Read throughput 1 per 15 clock. Write throughput 1 per 21 clock.	None

Table 14.4. Cache sizes on AMD Bulldozer, Piledriver and Steamroller

Figura 2: Comparativo entre algumas arquiteturas da AMD que utilizam de cache L3. Nota-se a diferença de tamanho com os caches L2 e L1. Tabela extraída de [3].

2.2 Associatividade

A associatividade consiste, resumidamente, na divisão do espaço de memória de um cache em conjuntos, com capacidade para armazenamento de um (direct-mapped cache) ou mais blocos (associativo por conjuntos ou totalmente associativo), que são alocados nos conjuntos conforme o campo “index” de seu endereço.

Nesse contexto, um cache pode ser:

- **Diretamente mapeado (direct-mapped cache):** conjuntos do tamanho de um bloco. Possui menos custo de hardware para promover o fetching do bloco;

- **Associativo por conjuntos:** cache dividido em n conjuntos, os quais podem ser capazes de armazenar mais de um bloco. Devido ao armazenamento de mais de um bloco, os misses de conflito diminuem, apesar do fato de um hardware adicional para fetching de um bloco dentre os elementos de um mesmo conjunto ser necessário. Apesar de maior complexidade de hardware, pode vir a ser benéfica a diminuição dos miss rates, a despeito do aumento da complexidade de hardware;
- **Totalmente associativo:** não há divisão do cache em conjuntos (ou, de outra forma, o cache inteiro é um conjunto), de forma a praticamente extinguir misses de conflito, que em tese somente aconteceriam caso o cache já estivesse cheio.

Em resumo, o aumento da associatividade é benéfico para a diminuição do miss rate, porém, a maior complexidade de hardware para o fetching do bloco desejado dentro de um conjunto aumenta o tempo de busca do bloco, consistindo em um tradeoff para o aumento ou não da associatividade.

Em [2], o estudo constando na tabela 3 é realizado:

Nas condições descritas, ou seja, com caches de diferentes tamanhos e associatividades (a despeito das rotinas de teste executadas, interessando aqui a forma de proporcionalidade entre associatividade e miss rate; e tamanho de cache e miss rate), nota-se que um cache de associatividade por 8 conjuntos tende a possuir um conflict miss-rate nulo ou praticamente nulo, de forma tal que seu comportamento pode ser aproximado ao de um cache totalmente associativo.

Ou seja, o aumento da associatividade não necessariamente precisa chegar ao extremo caso de cache totalmente associativo para que os misses de conflito sejam reduzidos a zero.

Como já dito, para o cache L1 é adequado priorizar a otimização do hit time, enquanto que para o cache L2, o ideal é que se otimize o miss-rate, a fim de evitar o miss penalty atrelado a uma busca no nível inferior da hierarquia de memória, que é significativamente maior que o miss penalty do próprio L2.

Nesse contexto, a associatividade do L1 não precisa ser muito alta, ao contrário do L2, em que a minimização do miss-rate é prioridade, valendo a troca do aumento do hit time em favor de uma maior associatividade e portanto um menor miss rate.

2.3 Política para armazenamento de blocos para os níveis de cache

Em um cenário envolvendo dois níveis de cache (suponha-se, por exemplo, L1 e L2), é importante uma política que descreva a forma como se dá a alocação de blocos e utilização

Cache size (KB)	Degree associative	Total miss rate	Miss rate components (relative percent) (sum = 100% of total miss rate)					
			Compulsory	Capacity	Conflict			
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size N has about the same miss rate as a two-way set-associative cache of size $N/2$ up through 128 K. Caches larger than 128 KB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

Figura 3: Comparativo entre miss rates conforme associatividade e tamanho do cache.

do espaço presente nos dois caches. Nesse contexto, três políticas importantes existem: a inclusão, exclusão, e não-inclusão.

- **Inclusão:** Dados dois caches L1 e L2, por exemplo, supõe-se que todos os blocos que estão presentes no L1 devem também estar presentes no L2;
- **Exclusão:** Nesse caso, blocos localizados no L1 não se encontram no L2, e vice-versa;
- **Não-inclusão:** o meio-termo entre inclusão e exclusão: um bloco pode ou não estar localizado em L2 e L1 simultaneamente.

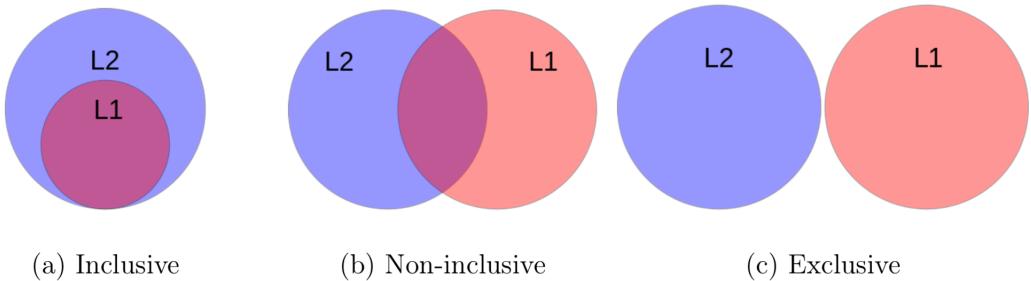


Figura 4: Representação gráfica das políticas de inclusão, exclusão e não-inclusão. Extraído de [1].

Primeiramente, a respeito da não-inclusão, trata-se de um meio termo que permite com que um bloco esteja ou não replicado nos dois níveis de cache porque permite cópia de um cache para outro mas não força a manutenção dessa cópia, como no caso da inclusão.

Por exemplo: caso em uma hierarquia de memória com dois níveis de cache, haja um miss que exija busca do bloco na memória, o bloco é gravado em L2 e depois em L1, estando localizado em ambos os caches. Porém, posteriormente, caso haja uma sobreescrita no tal bloco em L2, não é exigido que se exclua tal bloco em L1, como no caso da inclusão.

A não-inclusão, dessa forma, acaba se comportando de forma tal que tem a inclusão e a exclusão como casos extremos, que raramente ocorrem mas são possíveis.

Já a respeito da inclusão, esta tende a ser interessante frente à exclusão no quesito da simplicidade de implementação. Além disso, a pesar de as pesquisas realizadas pelo grupo não terem se aprofundado muito no fato, tal simplicidade é interessante em aplicações com multiprocessadores, por permitir lidar com mais facilidade com os problemas de coerência, que se tratam da problemática de blocos estarem localizados em diversos caches distintos, sem que escritas em tais blocos não fiquem dessincronizadas com os demais registros do mesmo bloco em outros caches.

Quanto à exclusão, trata-se de uma forma de otimização do espaço de armazenamento do cache. Ao contrário da inclusão, que exige que todos os blocos do cache L1 estejam replicados no cache L2, a exclusão ganha em espaço total de armazenamento em uma hierarquia de cache o tamanho do cache L1, em comparação com a política de inclusão.

Explicando de forma mais direta, conforme [1], sendo E o espaço de armazenamento total de uma hierarquia de caches, e c_n o espaço de armazenamento do nível n da hierarquia de cache:

Política de Inclusão:

$$E = c_n \quad (4)$$

Política de Exclusão:

$$E = \sum c_i \quad (5)$$

Através do aumento da capacidade de armazenamento da hierarquia de memória, menor se torna o miss rate do último nível de cache para a memória, e assim, menor é o miss penalty pago por acessos à memória.

Em geral, caches pequenos tendem a se beneficiar mais do uso da política de exclusão, e conforme o tamanho dos caches na hierarquia de memória são maiores, a diferença de efeito em relação a inclusão torna-se mínima, de forma a se preferir o uso da inclusão.

Em [9], foi justamente feito um estudo a respeito do uso da política de exclusão ou de inclusão e sobre qual deles apresentou melhor resultado. Uma das medições, por exemplo, consta na figura 5, em seguida explicada de forma bastante resumida e concisa:

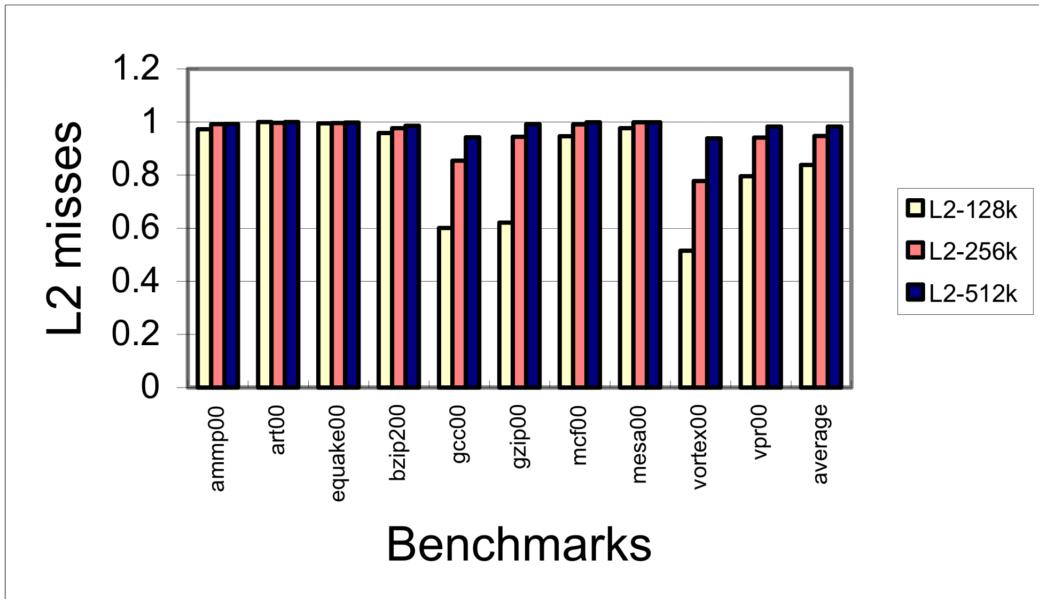


Figura 5: Comparativo de L2 misses com variação do tamanho do cache L2. Extraído de [9].

Para caches L1 com um tamanho fixo e L2 de tamanho variado, o gráfico mostra a razão entre a quantidade de misses obtidos através da adoção da política de exclusão e através da adoção da política de inclusão, para diferentes benchmarks. Nota-se que, para caches de 128kB, o uso da exclusão chegou a obter melhorias significativas, o que já não ocorre para caches L2 maiores, como para o caso de 512KB representado no gráfico em questão.

Ressalta-se que o tempo de execução também foi avantajado, graças aos miss penalties

para a memória evitados, mesmo que, em relação à inclusão, o hardware seja mais complexo, cobrando um maior tempo de processamento. Apesar disso, portanto, tem-se um cenário em que a política de exclusion é válida.

Apesar de haver outras estatísticas importantes, as duas acima apresentadas configuram nos fatores principais para a decisão a se tomar no presente projeto.

Em resumo, pode-se sintetizar os aspectos destacados na tabela 6.

	Inclusive	Non-inclusive	Exclusive
Data replication	↑↑	↑	None
Benefits	Simple coherence, no copy back necessary	Simple to implement	Highest effective capacity
Drawback	Wastes cache space, back invalidations	Data replication on a miss, complex coherence	More complex on an LLC hit
Replacement policies	Core-aware problem	Simple	Heuristic problem (no recency, frequency info)

Figura 6: Propriedades das Políticas de Inclusão. Extraído de [1]. Obs: LLC significa "Last Level Cache".

Concluindo-se o assunto, destaca-se a discussão a respeito da escolha da política mais adequada para o projeto em questão. Uma vez que a hierarquia de memória implementada pelo grupo é simples, com caches deveras pequenos, optou-se pela implementação da política de exclusion, a despeito de sua maior complexidade de hardware, no intuito de garantir otimização do pouco espaço de armazenamento disponível na hierarquia de cache.

Não se escolheu a política de inclusion justamente por trazer vantagem em cenário oposto ao da política de exclusion, em que há boa disponibilidade de espaço de armazenamento; além do fato de que não se está implementando no presente projeto uma aplicação multicore.

Quanto à aplicação da não-inclusão, esta tem o exclusion (e portanto as respectivas vantagens do mesmo, da qual se está buscando) somente como um caso extremo, de forma que ainda se preferiu o uso do exclusion.

Enfim, é importante destacar uma possível arquitetura para o uso da política de exclusion, que fora utilizada para o projeto, como mostrado na 7.

Na figura 8, destaca-se um diagrama de estados correspondente ao funcionamento da

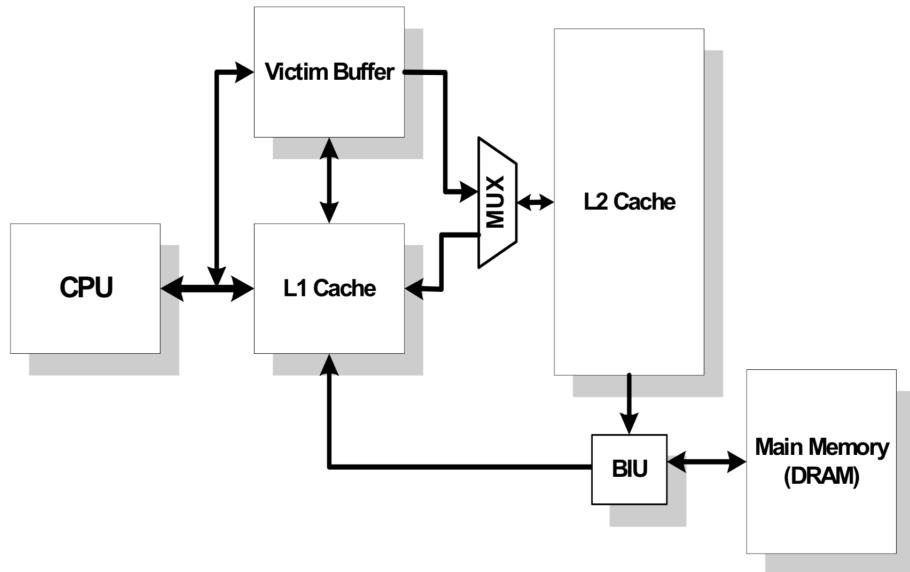


Figura 7: Arquitetura de cache com política de exclusão. Extraído de [9]

hierarquia de caches na presença da política de exclusão. Na fig. 7, consta a arquitetura usada na implementação, que conta com a presença do Victim Buffer - um buffer para o qual são “despejados” (evicted, em inglês) os blocos que sobre os quais são escritos blocos oriundos do cache L2. Esses blocos são armazenados no cache L2 após serem lidos do victim buffer, promovendo de certa forma um “swapping” de blocos entre os dois níveis de cache.

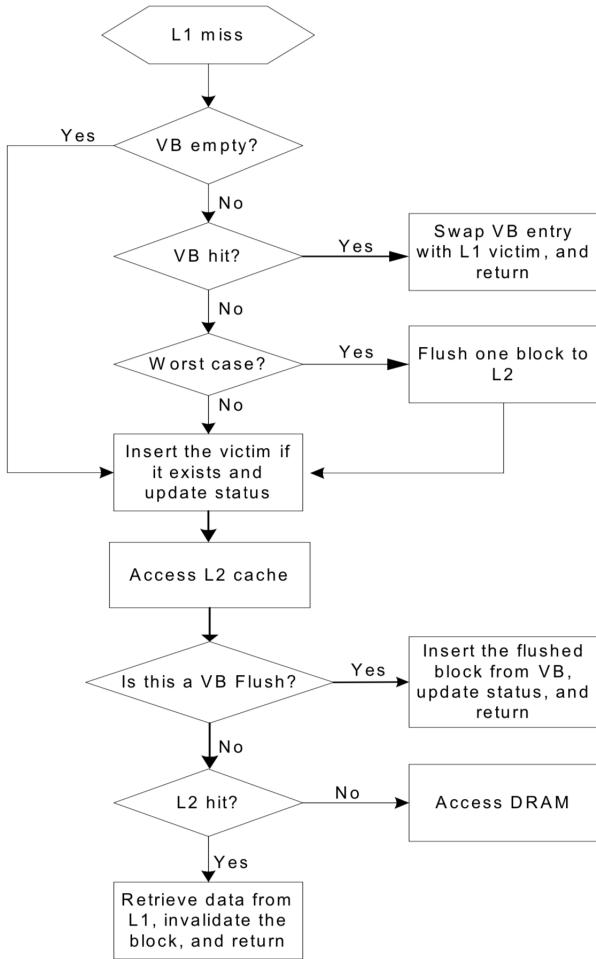


Figura 8: Diagrama de blocos para o cache com política de exclusão. Extraído de [9]

Ressalta-se que o victim buffer se trata de uma otimização a fim de que o cache L1 não precise interromper sua execução em espera do carregamento do bloco que despejou ser carregado no L2.

2.4 Tamanho dos Blocos

O tamanho dos blocos a serem transportados ao longo dos níveis de cache é uma decisão de projeto importante, que possui implicações diretas no desempenho do programa.

Por um lado, blocos grandes trazem menor miss compulsório, visto que blocos grandes trazem mais endereços consigo, de modo que um único miss compulsório aumenta no cache L1 a quantidade de endereços nunca antes referenciados, e assim diminuindo os misses compulsórios.

Por outro lado, o aumento do tamanho dos blocos faz com que aumentem os misses de capacidade e de conflito. A partir de um determinado tamanho de bloco, a otimização provinda do aumento de tamanho de bloco não é mais vantajosa, ou seja, os blocos se tornam grandes a ponto de a diminuição de misses compulsórios não mais compensar o aumento dos misses de conflito e de capacidade. O gráfico da figura 9, extraído de [2], ilustra esse fenômeno.

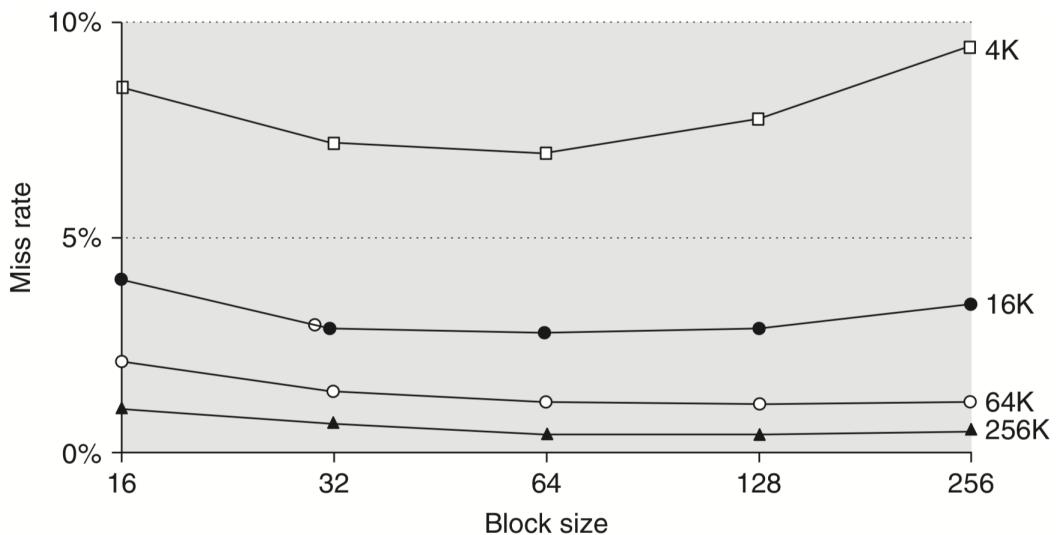


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 [Gee et al. 1993].

Figura 9: Block Size x Miss rate. extraído de [2].

2.5 Cache Replacement Policy

Trata-se da definição da situação em que, dado um cache com um conjunto cheio e a necessidade de load de um bloco oriundo do nível inferior na hierarquia de memória, decidir qual dos blocos do conjunto deve ser “despejado”.

Dentre inúmeros algoritmos possíveis, é de especial destaque a política LRU (Least Recently Used).

Muitos computadores atuais implementam o LRU. Porém, é de difícil implementação, de forma que existem variações a fim de facilitar tal implementação.

Em especial, para um cenário simples, de cache de associatividade 2, bastaria que se implementasse um bit de verificação, no qual dentre os dois elementos do conjunto, o último que tenha sido utilizado (em alguma operação de read/write, no caso) esteja definido como 1 e o outro, como 0. Assim, facilmente é definido o bloco a ser “despejado”.

Já no caso de associatividades maiores, o uso de bits se torna mais complexo. Para associatividade n , existem 2^n possibilidades, ou estados, de recência dos blocos em um conjunto, ou seja, 2^n situações possíveis que se deve analisar no caso da política LRU.

Uma alternativa, que diminui o custo da operação, é o uso do algoritmo pseudo-LRU, que consiste na implementação de uma árvore binária.

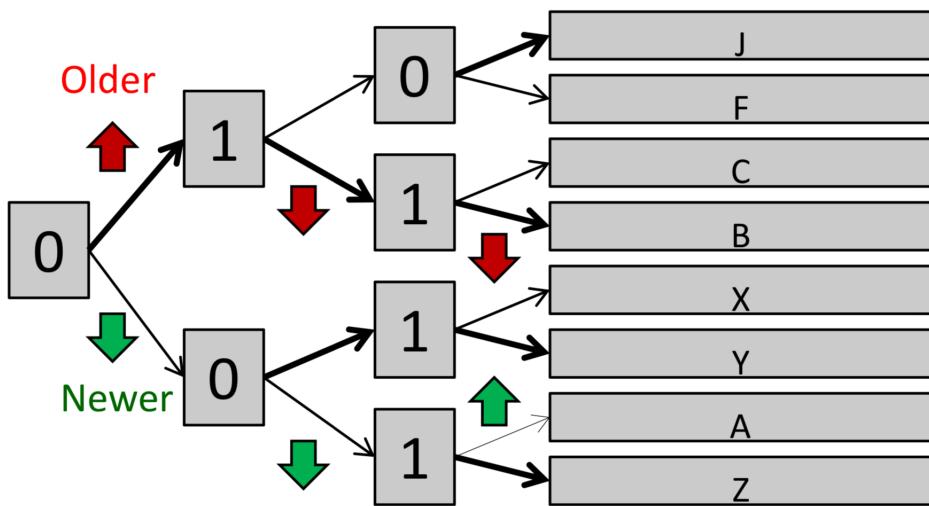


Figura 10: Arvore binária. Extraído de [5]

Cada ramo da árvore binária corresponde a um bit. Se a referência a um bloco no cache é feita “à esquerda” do bit, este assume um valor, e assume o valor inverso caso contrário.

Assim, conforme a figura 10, “seguindo-se” os bits de valor 1, correspondentes a acessos mais antigos, é possível chegar a um bloco de uso não tão recente. Pode não ser o mais antigo, mas para propósitos práticos, tal implementação já se torna boa o suficiente. A figura acima ilustra que, seguindo os nós cujo bit ficou estipulado como ‘1’, consegue-se chegar a um bloco antigo.

Enfim, em caso de implementação de uma cache replacement policy, o uso de LRU, e mais especificamente, um pseudo-LRU para associatividades grandes, é de interesse prático.

2.6 Write-Buffer

A utilização de um write-buffer tem como motivação permitir com que, após uma operação de write, não seja necessário o dispêndio de tempo no processo de atualização do bloco modificado em um nível da hierarquia de memória, sendo esse bloco reescrito nos demais níveis inferiores.

Através da utilização de um write-buffer, um bloco escrito é simplesmente depositado no write-buffer, e o cache de onde esse bloco é oriundo não necessita desperdiçar tempo de execução na transferência do dito bloco a níveis inferiores da hierarquia de memória. O bloco no write buffer será eventualmente transferido ao nível inferior da hierarquia de memória conforme a velocidade desse nível inferior ao qual se conecta.

Existem dois casos de write-buffers: write-back e write-through.

- Write-Back: permite o processador escrever dados diretamente no cache, assim ficando livre para executar outras operações; consome menos largura de banda e ocorre à velocidade do cache, mas pode gerar problemas como consistência de dados devido à latência e ocorrer escritas em blocos de endereços "dirty" da memória.
- Write-Through: o sistema, ao escrever para uma zona de memória, que está contida no cache, escreve tanto para a linha específica do cache quanto para a zona de memória, gerando uma consistência interna ao contrário do anterior, sendo ainda mais fácil de implementar. Em compensação utiliza de uma banda maior e possui um desempenho inferior ao do Write-Back.

2.7 Técnicas avançadas de otimização de cache

Algumas técnicas de otimização são sugeridas por [2], como “técnicas de otimização avançadas”. São elas:

- Caches de Nível alto (L1) mais simples e pequenos;
- Way-Prediction;
- Nonblocking caches;
- Multibanked cache;
- Critical Word First and Early Start;
- Merging Write Buffer;

- Hardware Prefetching;
- Uso de Compilador;
- Pipelined cache;
- Compilador controlando prefetching.

Tais técnicas de otimização sobretudo consistem ou em modificações em componentes da hierarquia de memória ou da inclusão de novos componentes, com novos recursos, à mesma. Podem ser classificadas conforme o parâmetro de desempenho da hierarquia de memória que visam otimizar:

- **Diminuição do hit time:** Caches de Nível alto (L1) mais simples e pequenos e Way-Prediction;
- **Aumento da bandwidth do cache:** Pipelined Cache, Multibanked cache, Non-blocking cache;
- **Diminuição do miss penalty:** Critical Word First and Early Start, Merging Write Buffer;
- **Diminuição do miss rate:** Otimização por compilador;
- Diminuição do miss rate ou do miss penalty através de paralelismo: Hardware Prefetching, Compiler prefetching.

Dentro do escopo de projeto sugerido, nota-se que nem todas as técnicas de otimização de cache são adequadas. O uso de compilador, caracterizado pelas técnicas de uso de compilador e de compiler prefetching citadas, foge do escopo do projeto e da disciplina, portanto sua implementação foi descartada. Também foi descartado o uso da técnica de Pipelined Cache - uma vez que já fora definida para esse projeto uma arquitetura pipeline com o uso de cache limitado a somente um estágio do pipeline.

A seguir, descrevem-se as características de cada otimização escolhida, suas vantagens, desvantagens e a motivação de escolha frente aos outros métodos preteridos. Em geral, buscou-se a escolha de métodos que possuíssem um bom equilíbrio entre bons resultados de otimização e simplicidade de implementação (a motivo de gestão do projeto - em termos de tempo e da consequente possibilidade de execução de mais testes, obtendo assim resultados mais consistentes).

2.7.1 Diminuição do hit time - Caches L1 mais simples e pequenos

Trivial, consiste no cuidado com a projeto dos caches em tamanho e complexidade. Os caches L1 estão em contato direto com a arquitetura pipeline, e dessa forma, são intensamente acessados pelo processador. Dessa forma, é interessante que, nesse nível, o hit time seja um parâmetro de acesso à memória favorecido frente aos demais, como já falado em seções anteriores. Cita-se como uma vantagem o fato de que caches pequenos e simples, cujo fetching de instruções/dados seja rápido, implicam em um baixo hit time, e logo, um melhor desempenho no acesso a memória. Esse método também traz consigo desvantagens, uma vez que o esforço na obtenção de menores hit times acaba ocasionando maiores miss rates. Se trata de uma questão de projeto achar a medida adequada para o problema. Algumas análises são interessantes para a presente discussão:

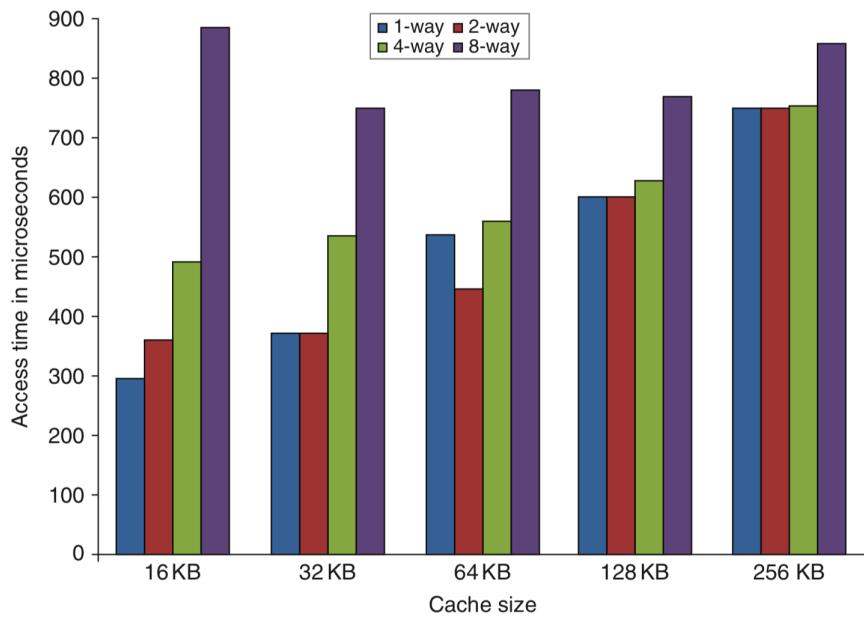


Figure 2.3 Access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan, Thoziyoor, and Jouppi [2005]. The data assume a 40 nm feature size (which is between the technology used in Intel's fastest and second fastest versions of the i7 and the same as the technology used in the fastest ARM embedded processors), a single bank, and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Since such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space rather than precision analysis of the trade-offs.

Figura 11: Tamanho do cache x tempo de acesso. Extraído de [2].

Intuitivamente, é possível pensar que o aumento no tamanho de um cache implica em menor hit time. Porém, conforme o gráfico da figura 11, nota-se que isso não necessariamente se trata de uma verdade, ao exemplo do cache associativo de 8 conjuntos, em que o aumento de tamanho do cache pode vir a implicar em aumento do tempo de acesso.

A tecnologia usada, o hardware de indexing e fetching dos blocos, etc, podem não apresentar uma relação diretamente proporcional com o hit time.

Outro aspecto importante sobre essa técnica de otimização de cache é o consumo de energia, representado para um caso específico no exemplo da figura 12:

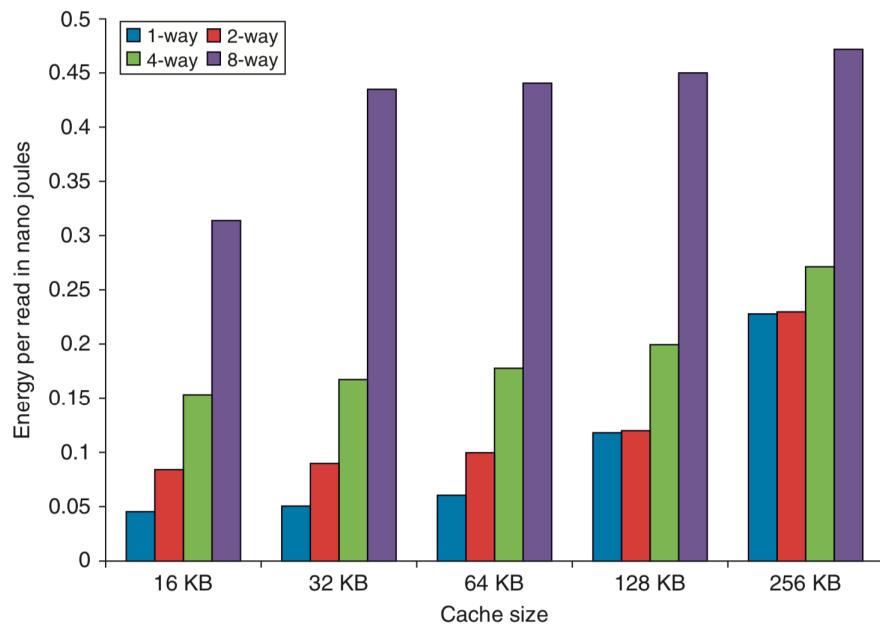


Figure 2.4 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.

Figura 12: Tamanho do cache x Consumo de energia por operação de read. Extraído de [2].

Em resumo: um cache L1 tem de ser pequeno e possuir menor associatividade, diminuindo assim o hit time. Processadores modernos até ousam adotar maior associatividade, ou até mesmo caches L1 um pouco maiores, a fim de diminuir o miss rate. Há casos, inclusive, em que o cache utiliza dois ciclos de clock do pipeline para fazer o acesso, o que permitiria com que fosse possível ter hit times maiores, sem prejudicar o período de clock.

2.7.2 Aumento da cache bandwidth - Multibanked Cache

Consiste no uso de caches compostos por múltiplos bancos de memória. Dessa forma, é possível haver múltiplos acessos simultâneos a memória que utilize dessa estratégia de otimização de cache, aumentando a bandwidth do mesmo. Esse método também reduz consumo de energia, além de baixa complexidade de hardware, em comparação com outros métodos (basta utilizar mais bancos e um hardware para executar o fetching de mais blocos, mas agora também em diferentes bancos, apesar de a estrutura desse hardware de buscar blocos não se alterar significativamente).

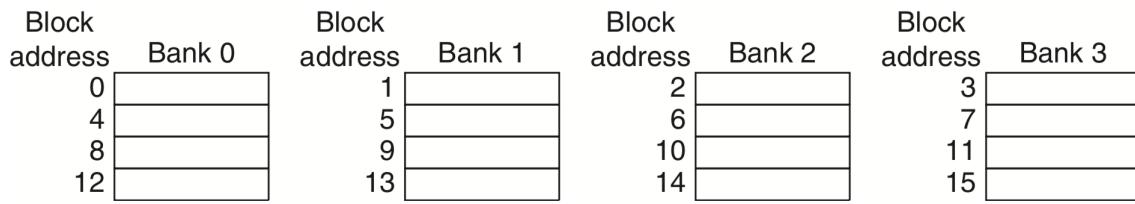


Figure 2.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Figura 13: Ilustração de um banked cache, utilizando sequential interleaving. A numeração dos blocos não é feita consecutivamente dentro de um mesmo banco. Figura extraída de [2].

Na figura 13, tem-se uma ilustração de um cache que seja composto de 4 bancos de memória com associatividade 4. O endereçamento é feito com “sequential interleaving” - blocos de numeração consecutiva presentes em bancos de memória diferentes. Acessos, assim, se espalham mais em bancos diferentes (princípio da localidade espacial faria com que blocos vizinhos tivessem maior probabilidade de uso. Mas nessa configuração, os acessos a blocos vizinhos acabam sendo em bancos distintos também, reduzindo eventual “gargalo” ao se acessar blocos consecutivos).

Na referência [2], até mesmo destaca um caso real de aplicação da técnica em questão. Processadores ARM e o intel Core I7 são exemplos: “*The Arm Cortex-A8 supports one to four banks in its L2 cache; the Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock), and the L2 has eight banks*” ([2], pag 86). Como é plausível concluir, níveis mais baixos na hierarquia de memória, justamente devido à sua possibilidade de possuir maior tamanho em termos de capacidade de armazenamento, suportam um maior número de memory banks, assim se aumentando a bandwidth de uma forma relativamente simples e que não expressivamente afeta o desempenho de outro parâmetro de otimização do cache.

2.7.3 Critical word first and early start

Trata-se de dois métodos de otimização que visam diminuir o tempo de obtenção de um endereço de memória por parte de um processador em um cenário em que ocorre um miss, evitando se esperar a obtenção do bloco inteiro ao qual pertence tal endereço, encaminhando ao processador diretamente a informação contida no endereço requisitado, assim que possível.

Em resumo:

- A técnica “critical word first” permite com que, na ocorrência de um miss, ao invés do acesso ao dado endereço ficar sujeito à transferência do respectivo bloco ao cache L1, o endereço demandado, somente, seja lido da memória através da implantação de hardware adicional para tal.
- Já na técnica “early start”, durante a transferência de um bloco para o cache L1, através de hardware adicional, monitora-se os endereços dentro do bloco transmitido ao L1. Assim que o endereço desejado for transferido ao L1, este já é encaminhado ao processador, antes mesmo do término da operação de transferência do respectivo bloco ao cache L1.

Essas duas técnicas são vantajosas, sobretudo, quando os blocos utilizados na hierarquia de memória possuem tamanhos grandes, visto que o adiantamento do endereço requisitado ao processador se torna mais significativo ao não se esperar o carregamento do bloco inteiro.

Entretanto, algumas questões se mostram problemáticas com esse método: caso após a execução desse método uma vez, outro endereço seja requisitado, dentro do mesmo bloco da requisição anterior, o cálculo do miss penalty pode se tornar uma tarefa complicada. Apesar disso, ainda vale o fato de que uma otimização através desse método ocorreu.

2.7.4 Merging Write-buffers

Quando uma escrita é feita, um bloco inteiro necessita ser atualizado nos níveis inferiores da hierarquia de memória.

Entretanto, se há uma operação de escrita em dois endereços pertencentes ao mesmo bloco, o correspondente bloco teria de ser atualizado duas vezes.

Com merging write-buffers, é possível que duas escritas no mesmo bloco ocorram em somente uma atualização de blocos. Para tanto, os write-buffers são monitorados de forma que se analisem os blocos no buffer e o endereço que foi alterado em cada um. Na eventualidade de se tratar de um mesmo bloco, as duas alterações de endereços são registradas

em um mesmo bloco, e assim as alterações nos níveis inferiores da hierarquia de memória são repassadas através de um único bloco, e não mais através de um bloco a cada escrita de endereço ocorrida, mesmo que de endereço pertencente ao mesmo bloco.

Em situações nas quais endereços consecutivos são escritos, essa técnica se mostra definitivamente vantajosa.

2.7.5 Hardware Prefetching

Utiliza-se da localidade espacial para adiantar a transferência de blocos. Sempre que ocorre um miss, ao invés de simplesmente executar o fetch no nível mais baixo da hierarquia e carregar o bloco correspondente, busca-se também o bloco seguinte, que é armazenado num instruction/data stream buffer (ou seja, são carregados mais de um bloco a cada miss, mas o adicional é armazenado em um buffer). Isso ajuda então a diminuir o miss rate, aproveitando-se da localidade espacial.

“A similar approach can be applied to data accesses [Jouppi 1990]. Palacharla and Kessler [1994] looked at a set of scientific programs and considered multiple stream buffers that could handle either instructions or data. They found that eight stream buffers could capture 50% to 70% of all misses from a processor with two 64 KB four-way set associative caches, one for instructions and the other for data.” ([2], pg. 91)

Nota-se, com o dado exemplo retirado de [2], que essas capturas de misses através de hardware prefetching são deveras significativas, visto que resolver cerca de 5% a 70% dos misses simplesmente transferindo blocos de um buffer permite uma grande economia de tempo ao se evitar o cumprimento de miss penalties.

Como uma desvantagem desse método, nota-se que necessita de mais bandwidth, para buscar do nível inferior um bloco a mais a cada miss (mesmo assim, isso é compensado caso o bloco no stream buffer venha a ser usado): *“When prefetching works well its impact on power is negligible. When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power”*. ([2], pag 92)

Segue na figura 14 outro exemplo citado por [2]. Cerca de 10% a 20% de melhora de performance é possível de se obter.

Em termos de aplicação real, destaca-se brevemente, inclusive, o fato de que um processador core i7 utiliza hardware prefetching, tanto para o nível L1 como para o nível L2.

Sobre a implementação do hardware prefetching, uma ideia simplificada é a implementação de uma fila FIFO, como mostram as figuras ?? e 15:

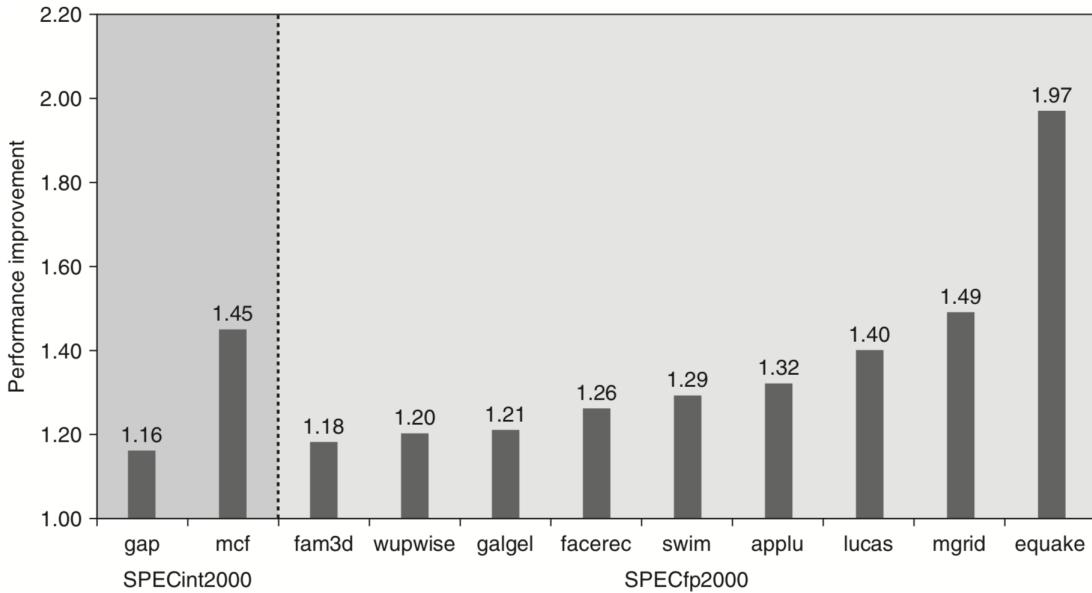


Figure 2.10 Speedup due to hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPEC benchmarks by less than 15% [Singhal 2004].

Figura 14: speedup obtido com hardware prefetching para diferentes benchmarks. Extraído de [2].

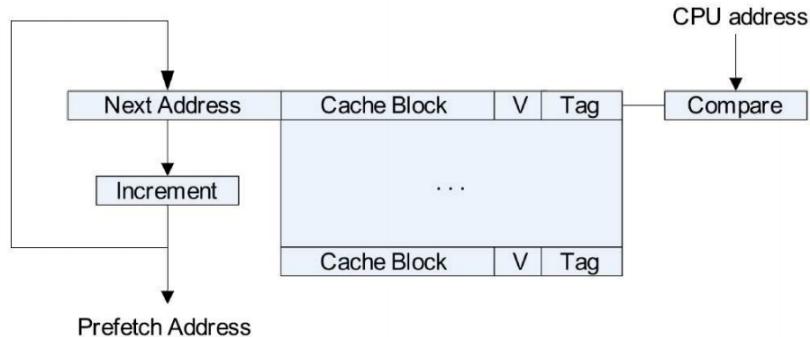


Figura 15: Stream Buffer. Extraído de [6].

2.7.6 Nonblocking Cache

Trata-se de uma técnica de otimização de hierarquia de memória que exige que o processador seja capaz de execução de instruções fora de ordem. Basicamente, quando ocorre um miss sem execução fora de ordem, o processador sofre stall e nenhuma instrução poderia ser buscada na hierarquia de memória até que o miss em questão seja resolvido.

Com a execução fora de ordem, no instante em que uma instrução sofre miss, existem outras instruções também disponíveis para serem executadas. A técnica de nonblocking cache se aproveita desse fato: durante a ocorrência de um miss, ao invés de esperar a resolução desse miss, permite a busca de outros endereços já disponíveis.

Existem duas situações possíveis nesse cenário, conforme [7] e [8]: miss under miss e hit under miss. O primeiro caso é mais complexo, consistindo na ocorrência de um miss já durante a ocorrência do primeiro miss em paralelo ao qual se aceitou processar a busca de outro endereço. Nesse caso, pode-se então aceitar o stall, ou não. Define-se um limite de quantos misses podem ocorrer simultaneamente até que se declare stall.

Já quando ocorre hit under miss, simplesmente a instrução é processada mesmo durante a ocorrência do primeiro miss.

A figura 16 ilustra brevemente a técnica de nonblocking caches. Quando ocorre um miss, a CPU não precisa ser interrompida. Interrompe somente quando o resultado desse miss é de fato necessário. Até esse momento, porém, a cpu pode continuar seu funcionamento.

Deve-se ressaltar também o fato de que essa técnica de otimização exige a possibilidade de o cache permitir mais de um acesso simultâneo: um para o processamento do miss e outro(s) para buscas em paralelo ao miss em ocorrência. Para se obter esse acesso simultâneo, a técnica de otimização de “multibanked caches” (seção 2.7.2) se faz essencial, justamente por permitir esse acesso simultâneo ao um cache.

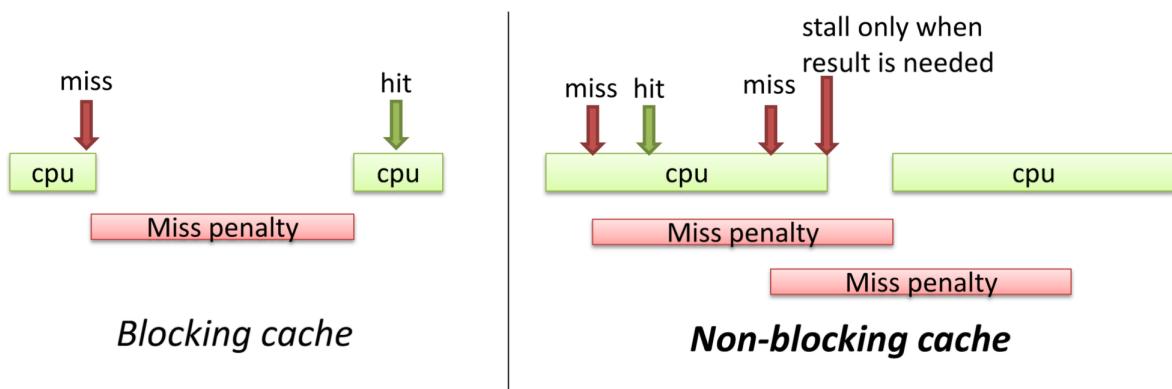


Figura 16: Diagrama comparativo entre Blocking Cache e Nonblocking Cache. Extraído de [8].

3 Conclusão da Pesquisa e Parâmetros do Projeto

Partindo do pressuposto de uma implementação simples e adequada a um projeto pequeno, com caches pequenos e de um determinado escopo de cronograma, implementou-se as seguintes otimizações:

- **Níveis na hierarquia de memória:** Foram implementados dois níveis de cache: L1 e L2. A presença de um L3 foi descartada, visto que a implementação do processador ARM em questão contava com a presença de somente um único núcleo;
- **Associatividade:** O cache L1 foi implementado com associatividade de 2 conjuntos, e o cache L2 foi implementado com associatividade de 8 conjuntos. Primou-se por não complicar demasiadamente a implementação da associatividade, ainda mais no presente cenário, com caches pequenos e com um escopo de tempo limitado, que aumentaria significativamente a dificuldade da implementação (por exemplo, a implementação de LRU para um conjunto associativo de 2 conjuntos é simples, como já dito em 2.5, bastando a implementação de um bit para verificar o bloco mais recente e o menos recente. Mas a implementação de um LRU para um caso associativo de 8 conjuntos demanda, por exemplo, a implementação do algoritmo pseudo-LRU, que poderia ser feito com uma árvore binária).

Ademais, eis o motivo de descarte de algumas das técnicas:

- **Critical Word First e Early Start:** Seus efeitos aparecem mais significativamente, sobretudo, acompanhados da presença de blocos grandes, o que não era o caso da presente aplicação. Assim, optou-se por não utilizar a técnica de otimização de cache.
- **Nonblocking Caches, juntamente a Multibanked Cache:** Primeiramente, nota-se que o segundo é condição para o primeiro, embora isso não fosse um fator de descarte. Como dito em 2.7.6, a técnica de nonblocking cache se torna interessante para o caso do dado projeto visto o contexto de execução fora de ordem. Entretanto, o escopo de tempo disponível, para uma técnica com certo grau de complexidade de implementação, fez com que o grupo optasse pela não implementação de tal método, apesar de que, registra-se aqui, se trata de um método interessante para a presente aplicação. Já a respeito da técnica de multibanked cache, primeiramente se destaca novamente o fato de que é necessária para o nonblocking cache, uma vez que permite os acessos simultâneos que este demanda. Quanto ao uso sem nonblocking cache, é vantajoso sobretudo em situações em que de fato há acessos múltiplos aos caches, como por exemplo em aplicações multicore, o que não é o caso. Devido à simplicidade do processador aqui implementado, optou-se por não implementar a técnica de otimização de multibanked caches.

- **Uso de compilador e compiler prefetching:** Simplesmente foge do escopo do presente projeto. Outro grupo ficaria responsável pela implementação de um compilador.
- **Pipelined Cache:** Esse projeto partiu de uma arquitetura ARM com 5 estágios de pipeline, sendo que os caches, cada um, ocupavam um único estágio. A técnica de otimização “Pipelined Cache” exigia que o acesso ao cache se desse em mais de um estágio do pipeline. Com a intenção de não se promover uma mudança na arquitetura do processador, inserindo mais estágios de pipeline, descartou-se tal técnica de otimização.
- **Merging Write Buffers:** Essa técnica se faz vantajosa principalmente no cenário de escrita em diversos endereços consecutivos. Primeiramente, seria interessante a observação dessa técnica de otimização em caches um pouco maiores, que permitissem de fato o teste com a escrita de vários endereços consecutivos, além do alcance dos blocos de duas palavras implementado (apesar de que, mesmo assim, se esperaria um resultado positivo da adoção dessa técnica no presente projeto). De toda forma, preferiu-se o foco em outros métodos de otimização em detrimento deste, crendo na obtenção de resultados melhores (como por exemplo a concentração de esforços na implementação da política de exclusão).

4 Implementação do Projeto

4.1 Hierarquia Básica

A hierarquia básica é representada na figura 21.

Trata-se, simplesmente, de uma implementação o mais básica possível de uma hierarquia de memória, que se utilize somente de um nível de cache, sem o uso de buffers e nenhuma técnica de otimização.

Implementou-se esse cenário com o intuito de comparar uma implementação livre de quaisquer otimização com a uma hierarquia otimizada, como descrita em 4.2

4.1.1 Cache L1 - Instruções

O Cache de instruções está inserido dentro da CPU e enquadra-se no estágio de Instruction Decode (ID) do pipeline do ARM. Dado o seu fim, possui apenas a função de leitura sendo portanto a escrita não permitida.

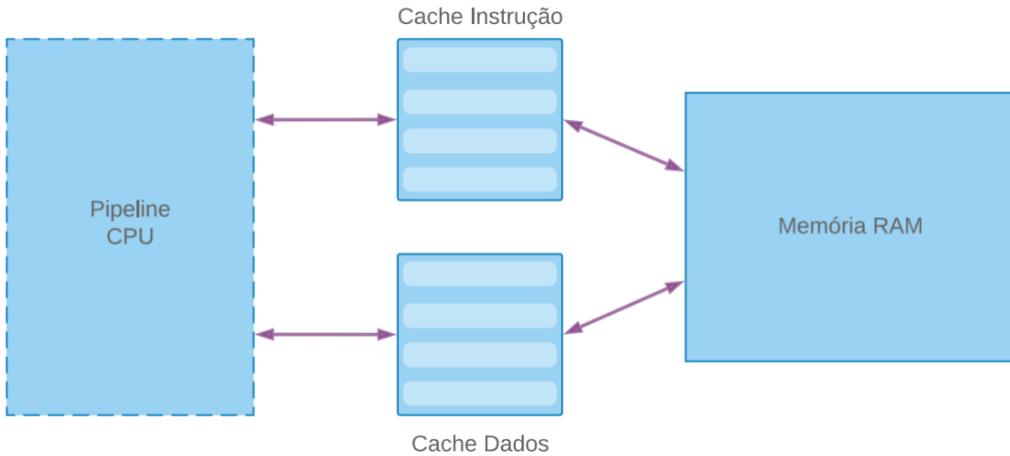


Figura 17: Hierarquia Básica

É importante mencionar que a leitura feita no cache é assíncrona em relação ao pipeline, isto é: não depende do sinal de clock deste componente. Contanto que as entradas do cache de instruções sejam válidas, sempre haverá um valor de saída para o pipeline. No entanto, toda vez que o program counter muda, este valor de saída é recalculado. A garantia da validade deste valor se dá pelo controle de stall implementado pelo controle do cache. Se um stall não for provocado, é garantido que o valor de saída corresponde àquele dado pelo endereço presente no pc. Do contrário, o valor de saída pode não ser o correto, devido ao fato do cache ainda estar calculando o index de acesso ao bloco ou mesmo estar acessando a memória para então fornecer o valor correspondente e correto para a cpu.

Fluxo de Dados

O Fluxo de dados do cache de instruções consiste basicamente em todo o circuito combinatório ou sequencial que concede o acesso de leitura para cpu e a correta seleção dos dados de leitura através da validação dos blocos no cache. A unidade de controle, em essência, apenas define os tempos e a sequência em que cada operação dentro do cache é feita, como será visto mais adiante. Dessa maneira, ainda que haja alguma dependência temporal dentro do fluxo de dados, no sentido de que não necessariamente uma entrada reflete imediatamente na saída, o fluxo de dados não possui estados. Isto é: suas saídas são definidas inteiramente por suas entradas.

A seguir, segue a interface do fluxo de dados:

Unidade de controle

A unidade de controle do cache de instruções, como já mencionado, tem como objetivo definir a sequência de operações que são realizadas dentro do cache de instruções. Em



```
-- I/O relacionados ao pipeline
clk:      in  bit;
cpuAddr:  in  bit_vector(9 downto 0);
stall:    out bit := '0';
dataOut:  out word_type := (others => '0');

-- I/O ao nível L2
dataIn:   in  word_vector_type(1 downto 0);
memReady:  in  bit;
memRW:     out bit := '0';  --- '1' write e '0' read
memEnable: out bit := '0';
memAddr:   out bit_vector(9 downto 0) := (others => '0');
state_d:   out bit_vector(2 downto 0)
```

Figura 18: Hierarquia Básica

resumo, o cache de instruções precisa receber um endereço de memória, verificar se os dados destes endereços estão presentes no cache, se estiver, retornar um Hit, se não estiver, mandar um stall para o pipeline enquanto busca os dados da memória. Finalmente, quando este dado estiver disponível no cache, atualizamos suas informações de tag e bit de validade e liberamos o pipeline do seu stall. Toda essa sequência de atividades é controlada para uma sequência de estados. A seguir, podemos ver um code snippet que mostra as entradas e saídas da unidade de controle. Note que aqui também dividimos essas entradas e saídas de acordo com a interface.

A seguir, segue a interface do fluxo de dados:

Em relação ao estágio ID do pipeline, segue as descrições de cada entrada/saída:

- **clk:** clock da unidade de controle do cache de instruções. Este clock não é o clock do pipeline. Em verdade, o clk deve ser pelo menos 5 a 10 vezes maior do que o clock do pipeline.
- **stall:** sinal enviado ao pipeline. Se '1', significa que ocorreu um miss no cache e que o pipeline deve ser parado até a devida coleta de dados na memória principal. Se '0', não é um stall e o pipeline pode seguir normalmente.
- **pc:** endereço de memória do program counter.

Em relação ao fluxo de dados do cache:



```
-- I/O relacionados ao controle
writeOptions:  in  bit;
updateInfo:    in  bit;
hit:          out bit := '0';

-- I/O relacionados ao IF stage
cpuAddr: in  bit_vector(9 downto 0);
dataOut: out word_type := (others => '0');

-- I/O relacionados a Memoria principal
memBlocoData: in  word_vector_type(1 downto 0);
memAddr:       out bit_vector(9 downto 0) := (others => '0')
```

Figura 19: Hierarquia Básica

- **hitSsingal:** sinal de hit que é enviado do fluxo de dados para a unidade de controle.
- **writeOptions:** sinal de controle que define o tipo de operação de write feito no cache. No caso do cache de instruções, há apenas duas opções. A primeira é manter o cache inalterado e a segunda é escrever um novo dado a partir da memória principal (no cenário onde ocorreu um miss).
- **updateInfo:** sinal de controle que sinaliza que as informações de valid bit e de tag do bloco devem ser atualizadas.

Em relação à memória:

- **memReady:** sinal da memória principal que avisa ao cache que a última operação (leitura ou escrita) foi concluída com sucesso e que portanto a memória está disponível para uma nova requisição.
- **memRw:** sinal enviada para a memória principal que diz se a operação é de read ou write.
- **memEnable:** habilita a memória principal.

A seguir, apresentamos a relação dos estados da unidade de controle:

- **INIT:** Estado inicial. Garante que todos os sinais de saída começam em zero. Na prática, este estado só deve ser atingido na inicialização do cache.
- **READY:** Estado que sinaliza que o cache já processou tudo que deveria e está esperando um novo endereço do pc.
- **CTAG:** Estado que compara Tags. Com isso, ele verifica se ocorreu um Hit ou um Miss.
- **CTAG2:** Estado que compara Tags após a escrita do novo bloco no cache a partir da memória principal
- **HIT:** Estado no qual simboliza que o cache teve um Hit
- **MISS:** Estado no qual simboliza que o cache teve um MISS
- **MEM:** Estado que indica que a memória principal completou sua operação

Como mencionado, o estado INIT é o primeiro estado a ser atingido pelo programa, mas não foi implementado uma maneira de retorno ao mesmo. Em sequência, o cache atinge o estado READY, e permanece nele até que ocorra uma mudança na entrada pc. Com esta mudança, o próximo estado é o CTAG, no qual toma a decisão de ir para o estado MISS ou HIT de acordo com o hitSignal. Se ocorreu Hit, não é necessário mas nenhuma operação e o sistema é direcionada ao estado READY. Com isso, é interessante notar que o estado Hit pode ser considerado redundante, pois com efeito, ele não modifica nenhuma saída, como será visto. No entanto, optou-se por permanecer com este estado, pois facilita a leitura do sistema. Se ao invés de Hit, o cache obteve um Miss, então o estado após o CTAG é o MISS, que é o estado no qual aciona-se o stall para o pipeline e habilita-se a memória para realizar a leitura. No momento em que a memória enviar um sinal de ready, o sistema passa para o estado MEM, no qual atualiza-se os dados do cache, utilizando-se para isso os sinais de writeOptions e updateInfo. Por fim, do estado MEM, passa-se para o estado CTAG2, que realiza a mesma operação que o estado CTAG. Esse estado, idealmente não deveria direcionar para um novo Miss, pois a esta altura os dados solicitados já foram transferidos da memória para o cache. No entanto, colocamos este estado por segurança. Além disso, diferente do estado CTAG, o CTAG2 mantém o sinal de stall ativo, que só será desativado quando a unidade de controle atingir novamente o estado de HIT.

4.1.2 Cache L1 - Dados

O Cache de dados, assim como o cache de instruções, também está inserido dentro do CPU enquadrando-se no estágio Memory (MEM) do pipeline do projeto. Diferentemente do cache de instruções onde só era permitida a função de leitura, aqui, tem-se também a possibilidade de escrita (que é usado no contexto de uma função Store).

a maior parte da lógica presente no primeiro cache mantém-se aqui. A grande diferença é que implementar a função de escrita traz consigo outras complexidades até então ausentes no cache do estágio de Instruction Fetch. Além disso, a troca da lógica de endereço que passou de direto para associativo, também aumentou o nível de complexidade do projeto do cache. Dado estas ressalvas, as discussões apresentadas na seção anterior continuando fazendo-se verdadeiras neste cenário.

Fluxo de Dados

O Fluxo de dados do cache de dados consiste basicamente de todo o circuito combinatório ou sequencial que concede o acesso de leitura e escrita para para cpu e a correta seleção dos dados de leitura através da validação dos blocos no cache. Assim como no cache de instruções, a unidade de controle, em essência, apenas define os tempos e a sequência em que cada operação dentro do cache é feita. O grande diferencial do fluxo de dados em relação ao cache anterior é que ao invés de utilizarmos um índice para blocos como índice principal do cache, usa-se aqui um índice para conjuntos. Dado que um conjunto é composto por dois blocos, teremos um total de 128 conjuntos e tão logo, um total de 128 índices principais possíveis para o cache de dados. O Fluxo de dados faz interface diretamente com três outros componentes: o controle do cache, o estágio MEM do pipeline e a memória principal. Em relação à unidade de controle, segue as descrições de cada entrada/saída:

A seguir, segue a interface do fluxo de dados:

```

● ● ●

-- I/O relacionados ao controle
writeOptions:  in bit_vector(1 downto 0);
updateInfo:    in bit;
hit:           out bit := '0';
dirtyBit:      out bit := '0';

-- I/O relacionados ao MEM stage
cpuAddr:       in bit_vector(9 downto 0);
dataIn :        in word_type;
dataOut:        out word_type;

-- I/O relacionados a Memoria principal
memBlockIn:    in word_vector_type(1 downto 0);
memAddr:       out bit_vector(9 downto 0) := (others => '0');
memBlockOut:   out word_vector_type(1 downto 0) := (others => word_vector_init)

```

Figura 20: Hierarquia Básica

- **writeOptions:** sinal de controle que define o tipo de operação de write feito no cache. No caso do cache de dados, existem três opções. A primeira é manter o cache inalterado, a segunda é escrever um novo dado a partir da memória principal (no cenário onde ocorreu um miss). Por fim, o último cenário é a escrita da cpu diretamente no cache.
- **memWrite:** sinal de controle que indica necessário para indicar o cenário que o fluxo de dados deve escrever na memória principal.
- **dirtyBit:** sinal enviado do fluxo de dados para unidade de controle que define que como o dado presente no cache está marcado como dirty_bit, então é necessário primeiramente escrever na memória, antes de persistir a escrita no cache.
- **updateInfo:** sinal de controle que sinaliza que as informações de valid bit, dirty bit e de tag do bloco devem ser atualizadas.
- **hit:** sinal que indica se ocorreu um hit ('1') ou um miss ('0').

Em relação ao estágio MEM do pipeline:

- **cpuAddr:** endereço fornecido pela cpu. Na prática é o valor do PC.
- **dataIn:** dado de entrada da cpu para o cache. Associada a operação de escrita.
- **dataOut:** dado de saída do cache para a cpu. Na prática é uma instrução.

Em relação à memória:

- **memBlockIn:** bloco de dados de transferência da memória para o cache.
- **memBlockOut:** bloco de dados de transferência do cache para a memória.
- **memAddr:** endereço de memória associado a primeira palavra do bloco de dados.

Unidade de controle

A unidade de controle do cache de instruções, como já mencionado, tem como objetivo definir a sequência de operações que são realizadas dentro do cache de dados. Em resumo, o cache de instruções precisa receber um endereço de memória, verificar se os dados destes endereços estão presentes no cache, se estiver, retornar um Hit, se não estiver, mandar um stall para o pipeline enquanto busca os dados da memória. Finalmente, quando este dado estiver disponível no cache, atualizamos suas informações de tag e bit de validade

e liberamos o pipeline do seu stall. Toda essa sequência de atividades é controlada para uma sequência de estados. A seguir, podemos ver um code snippet que mostra a entrada e saída da unidade de controle. Note que aqui também dividimos essas entradas e saídas de acordo com a interface.

A seguir, segue a interface do fluxo de dados:

```


-- I/O relacionados ao stage MEM
clk:          in bit;
clkPipeline:  in bit;
cpuWrite:     in bit;
cpuAddr:      in bit_vector(9 downto 0);
stall:        out bit := '0';

-- I/O relacionados ao cache
dirtyBit:     in bit;
hitSignal:    in bit;
writeOptions: out bit_vector(1 downto 0) := "00";
updateInfo:   out bit := '0';

-- I/O relacionados a Memoria principal
memReady:    in bit;
memRW:        out bit := '0';  --- '1' write e '0' read
memEnable:   out bit := '0';
--Para testes no top level
state_d :    out bit_vector(3 downto 0)

```

Figura 21: Hierarquia Básica

Em relação ao estágio ID do pipeline:, segue as descrições de cada entrada/saída:

- **clk:** clock da unidade de controle do cache de instruções. Este clock não é o clock do pipeline. Em verdade, o clk deve ser pelo menos 5 a 10 vezes maior do que o clock do pipeline.
- **clkPipeline:** é o clock do pipeline.
- **cpuWrite:** sinal que define se a operação da cpu é de leitura ou escrita.
- **stall:** sinal enviado ao pipeline. Se '1', significa que ocorreu um miss no cache e que o pipeline deve ser parada até a devida coleta de dados na memória principal. Se '0', não um stall e o pipeline pode seguir normalmente
- **cpuAddr:** endereço de memória do program counter

Em relação ao fluxo de dados do cache:

- **dirtyBit**: sinal que representa o dirtyBit do bloco atualmente selecionado no cache do fluxo de dados. É um sinal necessário para a unidade de controle definir se será necessário uma escrita na memória principal.
- **hitSignal**: sinal de hit que é enviado do fluxo de dados para a unidade de controle.
- **writeOptions**: sinal de controle que define o tipo de operação de write feito no cache. No caso do cache de dados, existem três opções. A primeira é manter o cache inalterado, a segunda é escrever um novo dado a partir da memória principal (no cenário onde ocorreu um miss). E por fim, o terceiro caso é escrever um bloco do cache na memória.
- **updateInfo**: sinal de controle que sinaliza que as informações de valid bit, de tag e do dirty bit do bloco devem ser atualizadas.

Em relação à memória:

- **memReady**: sinal da memória principal que avisa ao cache que a última operação (leitura ou escrita) foi concluída com sucesso e que portanto a memória está disponível para uma nova requisição.
- **memRW**: sinal enviada para a memória principal que diz se a operação é de read ou write.
- **memEnable**: habilita a memória principal.

Como se como se pode ver, utilizamos ao todo 9 estados, dois a mais do que no caso do cache de instruções. Estes dois estados estão relacionados à operação de escrita. Um para a escrita no cache e outro para a escrita na memória.

Apresentamos uma descrição do que é feito por cada estado.

- **INIT**: Estado inicial. Garante que todos os sinais de saída começam em zero. Na prática, este estado só deve ser atingido na inicialização do cache.
- **READY**: Estado que sinaliza que o cache já processou tudo que deveria e está esperando um novo endereço do pc.
- **CTAG**: Estado que compara Tags. Com isso, ele verifica se ocorreu um Hit ou um Miss.

- **CTAG2:** Estado que compara Tags após a escrita do novo bloco no cache a partir da memória. principal
- **HIT:** Estado no qual simboliza que o cache teve um Hit
- **MISS:** Estado no qual simboliza que o cache teve um MISS
- **MEM:** Estado que indica que a memória principal completou sua operação
- **WRITE:** Estado que indica que deve ocorrer uma escrita no cache a partir dos dados da cpu.
- **MWRITE:** Estado que indica que deve ocorrer uma escrita na memória.

Como mencionado, o estado INIT é o primeiro estado a ser atingido pelo programa, mas não foi implementado uma maneira de retorno ao mesmo. Em sequência, o cache atinge o estado READY, e permanece nele até que ocorra uma mudança na entrada pc. Com esta mudança, o próximo estado é o CTAG, no qual toma a decisão de ir para o estado MISS ou HIT de acordo com o hitSignal. Se ocorreu Hit, não é necessário mas nenhuma operação e o sistema é direcionada ao estado READY. Aqui surge a primeira diferença em relação ao cache de instruções, pois além das operações anteriores, o CTAG, também avalia se a operação que a cpu deseja fazer é de leitura ou escrita. Se for leitura, o estágio se comporta da mesma maneira que no cache de instruções. Se for caso de escrita, o cache avalia o dirtyBit para definir se a escrita precisa ser feita apenas no cache ou também na memória. No caso da leitura, a sequência possível se repete assim como no caso do cache de instruções. Se o leitor tiver dúvidas sobre isto, pode consultar a seção do cache de instruções no qual detalhamos este cenário em maior detalhe. Para o caso de escrita, se o dirtyBit for zero, significa que apenas uma escrita no cache é suficiente. Dessa forma, o próximo estágio é WRITE e depois passa para o READY. Contudo, se o dirty_bit for um, significa que é necessário antes escrever na memória o dado presente no cache e só após reescrever o novo dado no cache. Desta forma, o próximo estado é MWRITE e após o término da escrita na memória, passamos para o estágio WRITE e, finalmente, o READY.

4.1.3 Memória L2

Até então, falamos dos caches, que são o nível um na hierarquia de memória. Os caches, por definição armazenam dados de maneira temporária. Dessa forma precisamos, dentro da hierarquia, de componentes onde a persistência de dados seja permanente. Esta é a função da memória principal para este projeto.

A memória principal nesta hierarquia é responsável por atender a requisição de ambos caches (instruções e dados) de maneira síncrona, isto é: deliberadamente escolhe-se primeiro atender um cache e depois o outro. A cada "loop" da sua máquina de estados,

a memória verifica se um dos caches requisita algo, seja em uma operação de leitura ou escrita, e a memória então os atende em ordem. Importante dizer que o cache de instrução tem prioridade sobre o de dados.

4.2 Hierarquia Otimizada

A hierarquia otimizada baseia-se na hierarquia anterior e a evolui implementando caches multinível e outras otimizações. Como foi visto, existem diversas otimizações passíveis de serem aplicadas em uma hierarquia de Memória. Dentre aquelas estudadas, as escolhidos foram as seguintes:

- Associatividade 2-way
- Cache multinível (2 níveis) + memória principal
- Buffer Write-Back
- Buffer Write-Back
- Política de Exclusão

A seguir, apresentamos a nova hierarquia:

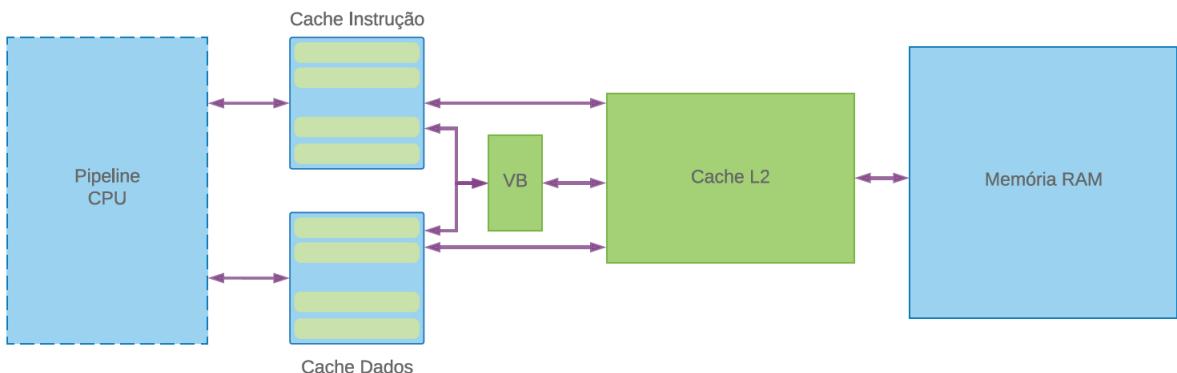


Figura 22: Hierarquia Otimizada

Note que os componentes em verde são os acréscimos em relação à hierarquia básica.

4.2.1 Caches L1

Os caches de instrução e de dados são essencialmente aqueles apresentados na hierarquia básica, com exceção da nova associatividade e da pequena adaptação de suas interfaces para lidar não mais diretamente com a memória principal, mas com outro cache, maior, de nível 2. Por essa razão, dispensamos maiores detalhes sobre os caches de nível 1 na hierarquia otimizada e, se interessar ao leitor a implementação em vhdl da associatividade 2-way, pedimos que consulte diretamente ao código fonte dos componentes.

4.2.2 Cache L2

A cache L2 tem a responsabilidade de responder a requisições de dados vindas do nível inferior, ou seja, lida com sinais vindos da cache de dados e de instruções. Dito isso foi necessária uma lógica a mais a fim de possibilitar receber solicitações simultâneas dessas entidades, além de administrar a escrita do Victim Buffer. O cache L2 possui três componentes: o seu topLevel **chacheL2**, o seu fluxo de dados e unidade de controle.

Fluxo de dados A seguir, vemos a interface do fluxo de dados:

A seguir, detalhamos algum dos sinais, que podem trazer dúvidas em sua função, dispensando assim comentários óbvios sobre sinais como é o caso do hit.

- **writeOptions**

Sinal de controle que indica qual ação o fluxo de dados deve tomar em relação ao write no cache, possivelmente sobrescrevendo os valores até então presentes.

- | | |
|-----|---|
| 0 0 | - mantém valor do cache inalterado |
| 0 1 | - usa o valor do mem (ocorreu miss) |
| 1 0 | - usa o valor do vbDataIn (victim buffer write) |

- **addrOptions**

Sinal de controle que indica qual endereço o cacheL2 deve utilizar para fazer seus processamento. Vamos lembrar que o L2 pode receber endereços tanto L1 de instrução, L1 de dados e ainda do victim buffer.



```
-- I/O relacionados ao controle
writeOptions:    in  bit_vector(1 downto 0);
addrOptions:    in  bit_vector(1 downto 0);
updateInfo:     in  bit;
ciL2Hit:        in  bit;
cdL2Hit:        in  bit;
delete:         in  bit;
hit:             out bit := '0';
dirtyBit:        out bit := '0';
vbWrite:         out bit := '0';

-- I/O relacionados ao victim buffer
vbDataIn:       in  word_vector_type(1 downto 0) := (others => word_vector_init);
vbAddr:          in  bit_vector(9 downto 0);
dirtyData:        in  bit;

-- I/O relacionados ao cache de dados
cdAddr:          in  bit_vector(9 downto 0);
cdDataOut:        out word_vector_type(1 downto 0) := (others => word_vector_init);

-- I/O relacionados ao cache de instruções
ciAddr:          in  bit_vector(9 downto 0);
ciDataOut:        out word_vector_type(1 downto 0) := (others => word_vector_init);

-- I/O relacionados a Memoria principal
memBlockIn:      in  word_vector_type(1 downto 0);
memAddr:          out bit_vector(9 downto 0) := (others => '0');
memBlockOut:      out word_vector_type(1 downto 0) := (others => word_vector_init)
```

- | | |
|-----|---|
| 0 0 | - não há nenhuma solicitação |
| 0 1 | - utiliza o endereço vindo da cache de instruções |
| 1 0 | - utiliza o endereço vindo da cache de dados |
| 1 0 | - utiliza o endereço vindo do Victim Buffer |

- **ciL2Hit/cdL2Hit**

Informa ao nível acima que para determinado bloco na cache L2, com isso tal sinal é usado para condicionar a mudança do bus de saída para a cache de instruções ou de dados.

delete: UC Cache L2 realiza o delete do bloco indexado. Bloco já enviado ao L1 (política de exclusion.)

- **vbWrite**

Identifica um write do victim buffer. O Write do Victim Buffer não necessariamente deixa o bloco “sujo”, isso só ocorre se esse sinal em questão estiver com o valor 1 enquanto estiver ocorrendo o write.

- **dirtyData**

Informa ao cache L2 se o dado que está sendo escrito pelo Victim Buffer é um dado defasado com a memória, ou seja, ocorreu um CPU write nesse bloco, na cache L1.

- | | |
|---|----------------------------|
| 0 | - Atualizado com a memória |
| 1 | - Defasado |

Unidade de Controle A seguir, comentamos sobre alguns sinais chaves para o funcionamento do componente:

- **vbReady**

Sinaliza ao vb que o write solicitado foi realizado. Dessa forma, este sinal só é avaliado quando há uma tentativa explícita do vb de escrever no cache L2.

- **delete**

realiza o delete do bloco indexado. Bloco já enviado ao L1 (política de exclusion.)

- **memRW**

- | | | |
|---|---|---------|
| 0 | 0 | - Idle |
| 0 | 1 | - Write |
| 1 | 0 | - Read |



```
clk:           in bit;
-- I/O relacionado ao victim buffer
vbDataIn:      in word_vector_type(1 downto 0) := (others => word_vector_init);
vbAddr:        in bit_vector(9 downto 0);
vbReady:       out bit := '0';

-- I/O relacionado ao cache de dados
cdEnable:      in bit;

-- I/O cacheL e datapath do L2
cdL2Hit:       out bit := '0';

-- I/O relacionado ao cache de instruções
ciEnable:      in bit;

-- I/O cacheL e datapath do L2
cilL2Hit:      out bit := '0';

-- I/O relacionados ao cache L2
dirtyBit:       in bit;
hitSignal:      in bit;
vbWriteL2:      in bit;
writeOptions:   out bit_vector(1 downto 0) := "00";
addrOptions:   out bit_vector(1 downto 0) := "00";
updateInfo:    out bit := '0';
delete:         out bit := '0';

-- I/O relacionados a Memoria principal
memReady:      in bit;
memRW:          out bit := '0';  --- '1' write e '0' read
memEnable:     out bit := '0'
```

Máquina de Estados Como dito anteriormente a cache L2 precisa lidar com diversos tipo de requisições que podem eventualmente acontecer simultaneamente. Para resolver isso adotou-se a seguinte estratégia:

Ordem de prioridade:

- Requisição Cache de Instruções
- Requisição Cache de Dados
- Escrita VB

Observação: Não fica preso apenas em requisição de maior prioridade, uma vez que a

lógica implementada alterna no caso de haver solicitações de menor prioridade.

A máquina de estados em questão não está otimizada, existem vários estados que só existem para facilitar o entendimento, sem muita utilidade prática.

Estado	Descrição
INIT	Estado inicial
READY	Indica que a cache não possui nenhum tipo de solicitação pendente.
REQ	Ocorreu uma solicitação, tal estado define a ordem de prioridade descrita anteriormente.
ICTAG	Requisição de instrução, compara a tag a fim de saber se houver hit ou não.
IMISSMWRITE	Ocorreu um Miss de instrução, contudo o bloco da cache para o índice em questão que irá ser substituído está defasado com a memória, logo é necessário primeiro enviar tal bloco para a memória antes de sobrescrever.
IHIT	Ocorreu um Hit, fim da requisição por instrução.
IMISS	Ocorreu um Miss, é necessário pegar o bloco da memória.
IMREADY	Bloco de instrução foi pego com sucesso da memória.
ICTAG2	Tal estado é apenas uma verificação extra para garantir que o bloco solicitado agora está na cache de fato. Obs.: Durante os testes quando ocorreu um miss nesse estado alguma coisa na lógica estava errada, logo foi bom para a depuração.
DCTAG	Mesma lógica do de instrução.
DMISSMWRITE	Mesma lógica do de instrução.
DHIT	Mesma lógica do de instrução.
DMISS	Mesma lógica do de instrução.
DMREADY	Mesma lógica do de instrução.
DCTAG2	Mesma lógica do de instrução.
CHECKVB	Verifica se o Victim Buffer possui algum write pendente.
VBCDIRTY	Verifica se bloco que será sobreescrito está defasado com a memória.
VBWRITE	Espera término da escrita no cache L2 para sair desse estado.

Tabela 1: Listagem de Estados

4.2.3 Memória L3

A memória implementada para a hierarquia otimizada é bem mais simples do que a memória da hierarquia básica, uma vez que não precisa diferenciar dado de instrução (tal diferenciação ocorre na cache L2). Dessa forma, o número de interfaces é menor, e o número de estados da UC também.

Assim como na hierarquia básica, a memória possui leitura e escrita num arquivo .dat denominado “memory.dat”, sendo cada linha do arquivo corresponde a uma word (32 bits).

Fluxo de Dados

Sinal	Descrição
clk (in)	clock
RW (out)	Destino: FD Memória. 00 Idle. 10 Write. 01 Read
cRead (in)	Origem: FD Memória. 1 Informa a unidade de controle que o read terminou. 0 Read não finalizado ou não iniciado.
cWrite (in)	Origem: FD Memória. 1 Write terminou. 0 Write não finalizado ou não iniciado.
enable (in)	Origem: Cache L2. Se o valor for igual a 1 significa que a memória possui uma requisição pendente da cache L2.
memRw (in)	Origem: Cache L2. 1 Write. 0 Read
addr (in)	Origem: Cache L2. Endereço do Bloco.
dataIn (in)	Origem: Cache L2. Bloco de entrada.
memReady (out)	Destino: Cache L2. Informa ao L2 que memória está disponível.

Tabela 2: Interface da Memória L3

Unidade de Controle

Máquina de Estados A seguir estão descritos os estados implementados:

Estado	Descrição
INIT	Estado Inicial
READY	Memória está disponível.
WRITE	Memória está processando um write.
READ	Memória está processando um read.

4.3 Victim Buffer

4.3.1 Descrição

O victim buffer consiste em um buffer que recebe blocos “despejados” de ambos os caches L1 (instrução e dados) que, posteriormente, serão lidos pelo cache L2 e nele carregados em momento oportuno para esse cache L2.

Está implementado como uma fila FIFO. Criou-se, no datapath do victimBuffer, um vectorType correspondente a um bloco, e também processes para a retirada de blocos (o primeiro elemento do vetor de blocos, daí a FIFO) e para a inserção de blocos pelos caches de instrução e de dados.

4.3.2 Interface

Conforme descrita na tabela 3.

4.3.3 Implementação

- Top Level

Somente são instanciados o top level e a unidade de controle do victim buffer, e feitas as ligações de sinais entre os módulos que forem necessárias.

- Fluxo de Dados

é implementado o buffer como um vetor de blocos. Também são implementados processos para a inserção de blocos no buffer, oriundos ou do cache de dados, ou do cache de instruções; e também para a leitura e remoção dos blocos do buffer (sempre é retirado

Sinal	Descrição
clk	sinal de clock
queueInst	sinal de controle que o cache L1 de instruções ativa informando gravação de bloco “despejado” no victim buffer
queueData	sinal de controle que o cache L1 de dados ativa informando gravação de bloco “despejado” no victim buffer
readyL2	sinal de controle oriundo do cache L2, indicando que irá ler e gravar um elemento do buffer, e em seguida eliminar esse elemento do buffer.
evictedBlockData	bloco “despejado” do cache de dados
evictedBlockDataAddress	Endereço do bloco “despejado” do cache de dados
evictedBlockDataDirty	dirty bit correspondente ao bloco “despejado” do cache de dados
evictedBlockInst	bloco “despejado” do cache de instruções
evictedBlockInstAddress	Endereço do bloco “despejado” do cache de instruções
evictedBlockInstDirty	dirty bit correspondente ao bloco “despejado” do cache de instruções
blockOut	saída do buffer: bloco lido pelo cache L2
blockOutAddress	endereço do bloco de saída
blockOutIsDirty	dirty bit do bloco de saída

Tabela 3: Interface do victim buffer

o índice 0 do vetor, e os blocos nos índices subsequentes são deslocados à direita após a leitura do índice 0 do vetor).

Tais processos são sensíveis a sinais oriundos da unidade de controle (no caso, queueBlockData, queueBlockInst e readyRead, como já mencionado na 3.)

4.3.4 Unidade de Controle

Segue a máquina de estados implementada, na figura 23.

Destaca-se aqui que na situação em que os caches requisitam acesso ao victim buffer simultaneamente, a máquina de estados identifica o fato da simultaneidade das requisições

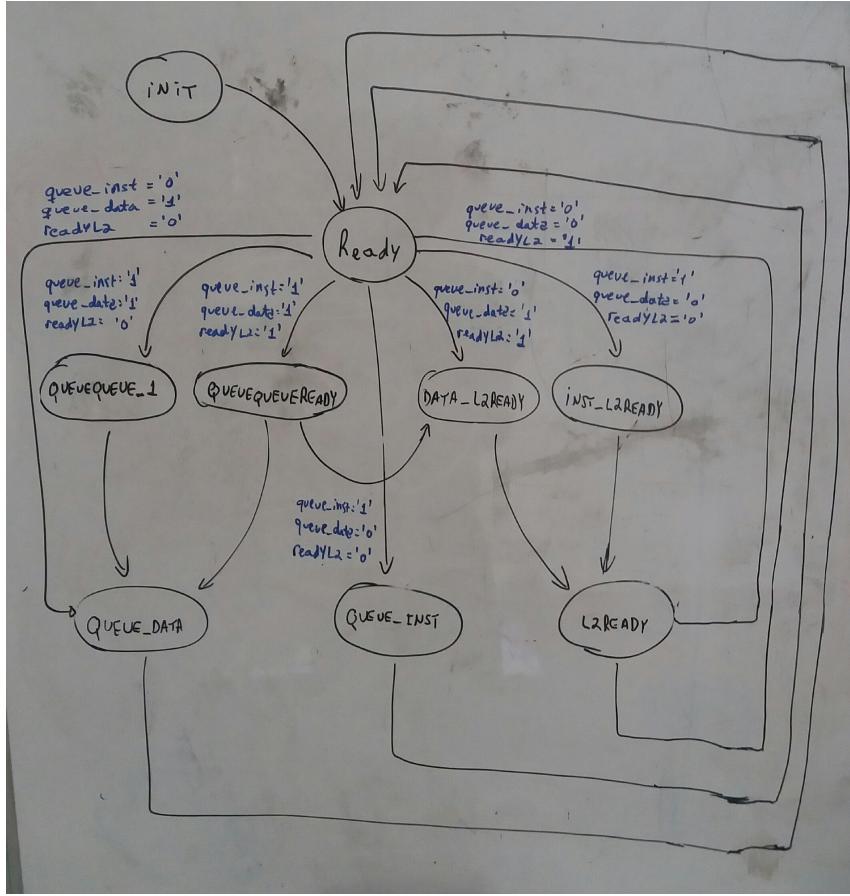


Figura 23: Unidade de Controle para o Victim Buffer.

e trata as requisições uma a uma, em cada estado. A tabela e os diagramas de estados documentam essa dinâmica implementada.

É importante também destacar os sinais de controle associados a essa máquina de estados. Seguem na tabela 5.

Além disso, as descrições dos estados associados a essa unidade de controle constam na tabela 4.

- Arquivos: victimBuffer.vhd; victimBufferControl.vhd; victimBufferPath.vhd

Estado	Descrição
INIT	Estado inicial da máquina de estados
READY	Estado em que a máquina de estado está pronta para que seja executada uma operação de inserção ou remoção de bloco no buffer
QUEUE_DATA	Cache de dados “despeja” bloco, a ser inserido no victim buffer
QUEUE_INST	Cache de instruções “despeja” bloco, a ser inserido no victim buffer
L2READY	Cache L2 lê e insere bloco do victim buffer. Posteriormente, deleta-se o bloco lido do victim buffer.
QUEUEQUEUE_1	Estado é ativado quando os caches de instrução e de dados, simultaneamente, demandam ao victim buffer inserção de bloco. Nesse estado, QUEUEQUEUE_1, é feita a inserção do bloco do cache de instrução, e em seguida o próximo estado é o QUEUE_DATA, completando a operação.
"INST_L2READY	Inserção de bloco do cache de instruções e leitura de bloco por cache L2 requisitados simultaneamente. Nesse estado, é inserido o bloco do cache de instrução. O estado seguinte é o L2READY
DATA_L2READY	Idem INST_L2READY, porém bloco inserido vem do cache de dados
QUEUEQUEUEREADY	Situação em que os três caches (dados, instrução e L2) requisitam uso do victim buffer.

Tabela 4: Estados da UC do Victim Buffer.

Sinal	Descrição
queueBlockData	Sinal de controle oriundo da Unidade de controle. Diferente do sinal queueData*
queueBlockInst	Sinal de controle oriundo da Unidade de controle. Diferente do sinal queueInst*
readyRead	Sinal de controle oriundo da Unidade de controle. Diferente do sinal "readyL2"

Tabela 5: Sinais VictimBuffer

4.4 Módulo Tester

4.4.1 Descrição

O “Tester” é um módulo de teste para as hierarquias básica e otimizada. Continuamente promove a entrada de endereços à hierarquia de memória, em diversos modos possíveis. Em caso de stalls na hierarquia de memória, a emissão de endereços pelo tester também é interrompida, até que o miss seja resolvido.

4.4.2 Interface

Para os “generic”, tem-se o “addressSize”, definindo o tamanho do endereço de memória em bits; e os demais sinais, que se tratam simplesmente de variáveis auxiliares para a geração de endereços aleatórios. Alterando-se seus valores, diferentes sequências de endereços aleatórios são obtidas.

Sinal	Descrição
clk	clock do módulo tester. Deve ser o mesmo clock que alimenta o pipeline. É maior que o clock dos caches(ao menos 5 vezes, correspondentes ao número de ciclos de clock para um L1 hit)
restartAddr	executa um “reset”, e endereço volta a ser o inicial (variáveis startAddressData e startAddressInst)
addressMode	Modo de endereço. vide tópico “how to use” abaixo da tabela.
cacheMode	Vide seção “how to use” abaixo da tabela
startAddressData	endereço inicial para cache de dados
startAddressInst	endereço inicial para cache de instruções
endAddressData	Endereço de fim da simulação para cache de dados
endAddressInst	Endereço de fim da simulação para cache de instruções
setAddressEnable	se ‘1’, altera endereços para “setAddressData” e “setAddressInst”
setAddressData	endereço a ser alterado, se “setAddressEnable” = ‘1’
setAddressInst	endereço a ser alterado, se “setAddressEnable” = ‘1’
stallData	stall vindo da hierarquia de memória
stallInst	stall vindo da hierarquia de memória
isBranchData	enable de branch no cache de dados
branchUpDownData	define se branch no cache de dados será com soma ou subtração ao endereço corrente
isBranchInst	enable de branch no cache de instruções
branchUpDownInst	define se branch no cache de instruções será com soma ou subtração ao endereço corrente
branchDataOffset	offset do branch no cache de dados
branchInstOffset	offset no branch no cache de instruções
toTestAddressData	endereço a ser buscado na hierarquia de memória, no cache de dados
toTestAddressInst	endereço a ser buscado na hierarquia de memória, no cache de instruções

Tabela 6: Listagem dos arquivos da Hierarquia Otimizada

4.4.3 How To Use

O uso do “tester” é simples, e basicamente consiste, nas simulações, da estipulação de valores para os seguintes sinais:

- clock

Para definir modos de funcionamento:

- addressMode;
- cacheMode;
- startAddressData;
- startAddressInst;
- endAddressData;
- endAddressInst;

Para forçar a busca de um endereço:

- setAddressEnable;
- setAddressdata;
- setAddressInst

Para forçar salto:

- isBranchData;
- branchUpDownData;
- isBranchInst;
- branchUpDownInst;
- branchDataOffset;
- branchInstOffset.

Observações:

- O “Tester” provê endereços tanto ao cache de dados como ao cache de instruções, em separado (sinais toTestAddressData e toTestAddressInst).
- O clock do “Tester” deve ser o clock do pipeline, que deve possuir maior período em relação ao clock dos caches. Adotou-se para as simulações o clock do tester como cinco ciclos de clock dos caches, correspondentes ao tempo de hit dos caches L1.
- Há 4 modos de endereçamento, conforme manipulados através do sinal “addressMode”.
- Modo “00”: endereços consecutivos a partir do valor dos sinais “startAddress” (sinais “startAddressData” e “startAddressInst”);
- Modo “01”: endereços com offset randômico. Os endereços começam no “startAddress” e os endereços consecutivos consistem em saltos com valores de offsets aleatoriamente definidos;
- “Modo 10”: instruções totalmente randômicas. Aqui, não é o offset de salto que é gerado aleatoriamente, mas sim os próprios endereços. A aleatoriedade de endereços abrange uma área maior, que pode cobrir todo o espaço de endereçamento de um programa, não somente os endereços das proximidades do que se está sendo buscado no momento.
- Modos de cache: define-se o funcionamento ou não dos testes para os caches de dados e de instruções, segundo o sinal “cacheMode”.
- Modo “10”: promove-se endereçamento somente para cache de instruções;
- Modo “01”: somente cache de dados;
- Modo “11”: ambos os caches, instrução e dados

Sobre os demais sinais mencionados no início dessa subseção, seus funcionamentos já contam na tabela descritiva dos sinais da interface do módulo.

4.4.4 Exemplo de funcionamento do Tester

Ilustra-se, através de um exemplo, o funcionamento do módulo "tester".

Para manipulá-lo, pode-se utilizar os sinais correspondentes na interface do component "top level" de cada hierarquia (tal componente encapsula o tester e a hierarquia de memória correspondente).

Suponha-se as seguintes condições de simulação desejadas:

- Iniciar o funcionamento do cache de dados a partir do endereço "100", e funcionamento do cache de instruções a partir do endereço "101";
- Endereços de palavras consecutivas vão sendo lidos em cada caso
- Simulação no cache de dados deve terminar no endereço 040 (hex), e no cache de instruções, no endereço 039 (hex)

Essas configurações são atingidas definindo os valores dos sinais da interface do tester da seguinte forma:

- startAddressData = "100"
- startAddressInst = "101"
- endAddressData = 040 (hex)
- endAddressInst = 039 (hex)

Nota-se, na figura 24, que com a definição dos sinais da interface da forma mencionada, os endereços de saída (toTestAddressData e toTestAddressInst) apresentaram o comportamento desejado.

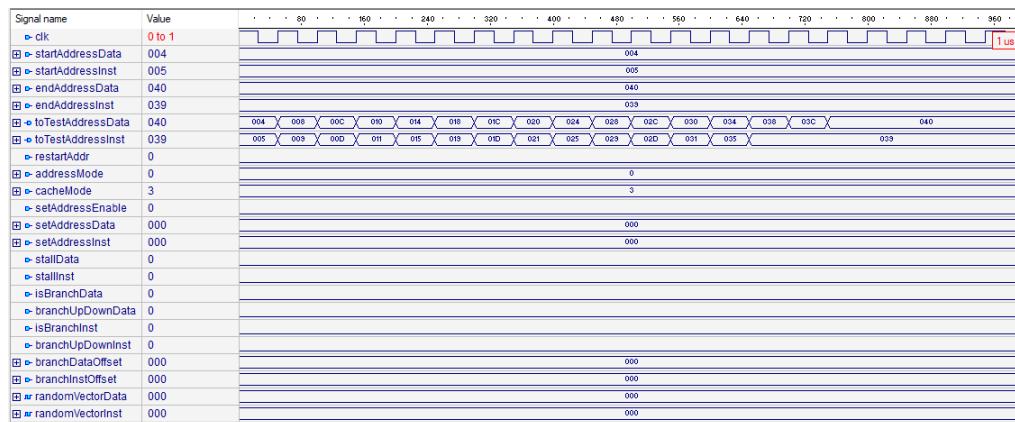


Figura 24: Teste de endereços consecutivos para o módulo tester. Arquivo "Tester1.awc".

Caso se desejasse o uso de branches, bastaria que se acrescentasse à simulação da figura 24 a definição de mais alguns sinais na interface do módulo em questão:

- **isBranchData = '1'**: se, numa borda de subida de clock esse sinal está ativo, ocorre salto conforme valor do offset definido.
- **branchUpDownData = '1'**: se '1', soma-se o offset ao valor corrente do endereço, e se '0', subtrai-se.
- **branchdataOffset = "1000"**: offset do salto.

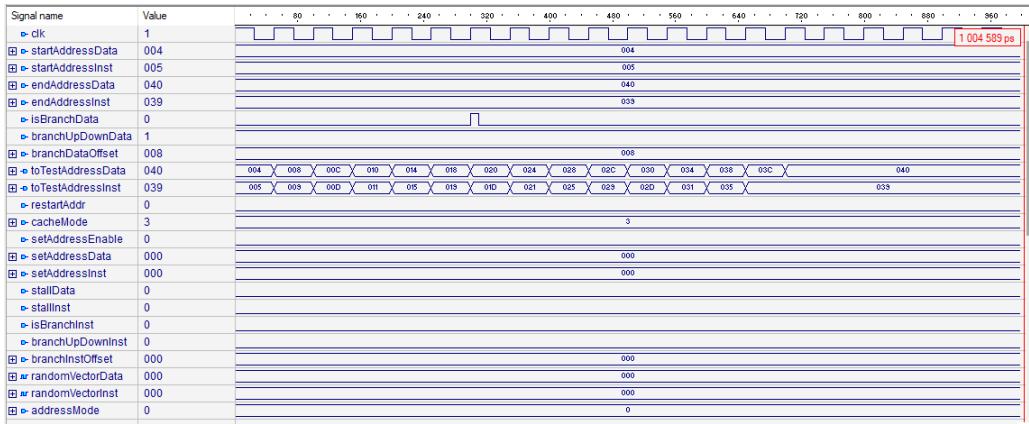


Figura 25: Teste de branch para o módulo tester. Arquivo "Tester₂.awc".

Enfim, para se utilizar o tester, basta utilizar adequadamente as interfaces desse componente. Além das funcionalidades aqui demonstradas, outras (várias) estão disponíveis para uso do tester, e basta que sejam configuradas segundo a descrição da interface em "How to Use".

Por exemplo, uma opção é a utilização, ao invés de endereços de palavras consecutivas, a utilização de endereços randômicos. Para tanto, basta definir "mode = "01", e conforme a figura 26.

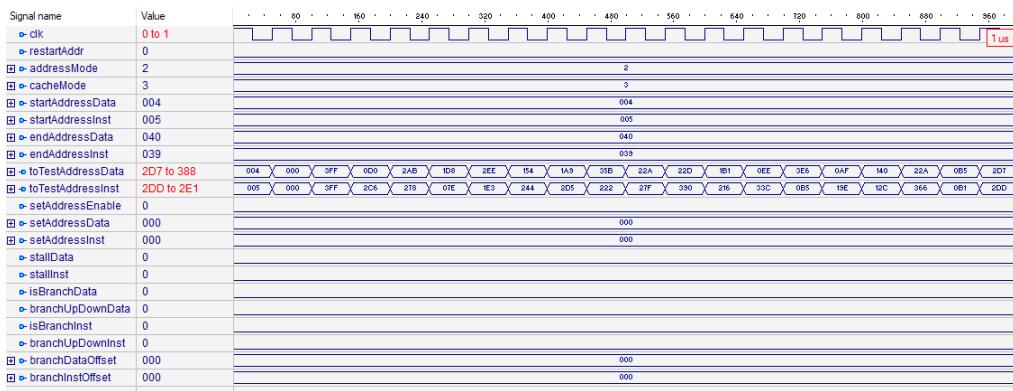


Figura 26: Teste de endereços aleatórios para o módulo tester. Arquivo "Tester₃.awc".

5 Testes Executados

Foram realizados uma série de testes com múltiplos cenários tanto da hierarquia básica quanto da memória. Vamos em detalhes especificar os resultados de cada teste ao decorrer desta seção. No entanto, gostaríamos de começar apresentando os resultados finais, que mostrar a comparação compilada da performance de ambas as hierarquias:

	Relative Stall Time	Miss Rate	Memory Acces Time
Básica	91.74%	52.08%	124 ns
Otimizada	70.40%	35.90%	74 ns
A/B	↓ 28%	↓ 31.1%	↓ 40.3%

Tabela 7: Resultados finais

A partir dos dados obtidos pela tabela 7, concluímos que de fato uma hierarquia que utiliza de técnicas de otimização de fato performa melhor, de acordo com os principais critérios utilizados para se medir a eficiência de uma hierarquia de memória.

5.1 Teste 1: Hierarquia Básica

- Arquivo: theSecondAccess.awc

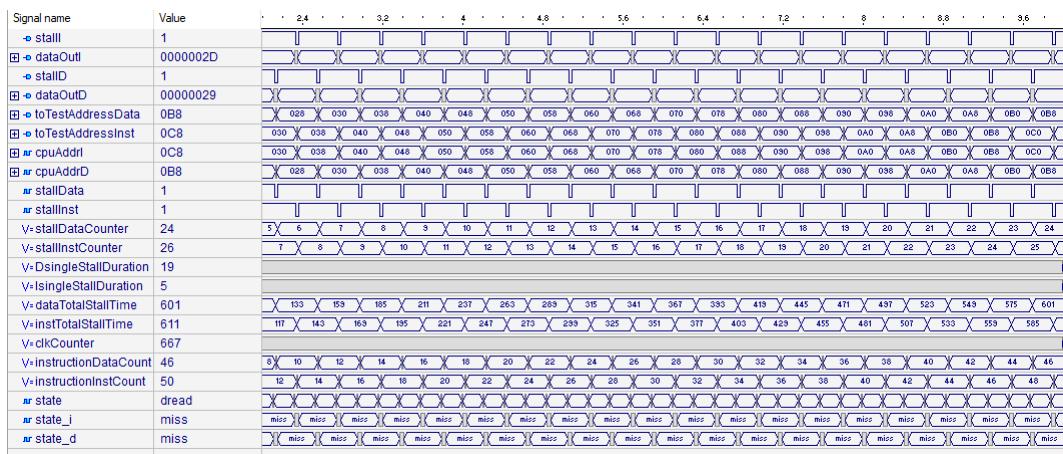


Figura 27: Forma de onda do teste 1.

Descrição	Parametros
Tempo de Simulação	10000 ns
Clock do Pipeline	15 ns
Clock do Cache	5 ns
accessTimeMemory	200 ns
ciclos de clock	666
Endereços de dados executados	46
Endereços de instrução executados	50
Número Stalls de dados	24
Número Stalls de instrução	26
Tempo stall de dados	601
Tempo stall de instruções	611
Relative Stall Time	$611/666=91,74\%$
Miss Rate	52.08%
Access memory Time	124 ns

Tabela 8: Resultados Teste 1

5.2 Teste 2: Hierarquia Otimizada

- **Arquivo:** hierarquiaOtimizadaComLRU.awc (padronizar nomes dps!)

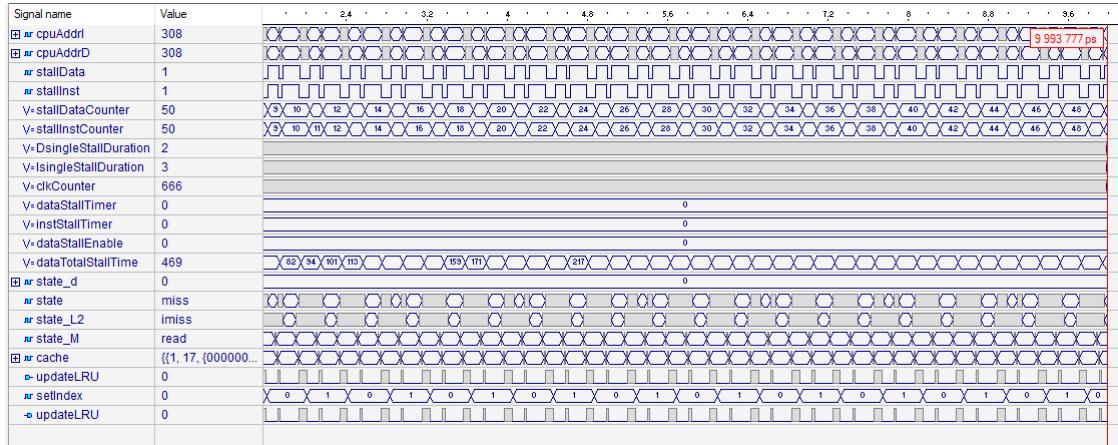


Figura 28: Forma de onda do teste 2.

Descrição	Parametros
Tempo de Simulação	10000 ns
Clock do Pipeline	15 ns
Clock do Cache	5 ns
accessTimeMemory	200 ns
ciclos de clock	666
Endereços de dados executados	194
Endereços de instrução executados	194
Número Stalls de dados	50
Número Stalls de instrução	50
Tempo stall de dados	469
Tempo stall de instruções	468
Relative Stall Time	$469/666 = 70,4\%$
Miss Rate	$/194 = 25,77\%$
Access memory Time	71,54%

Tabela 9: Resultados Teste 2

5.3 Teste 3: Hierarquia Otimizada

- Arquivo: 3fcOK.awc (padronizar nomes dps!)

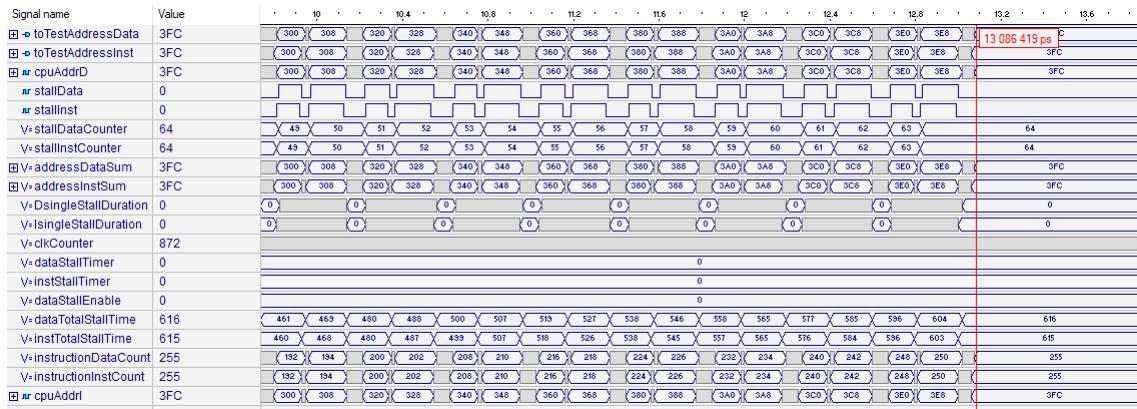


Figura 29: Forma de onda do teste 3.

Descrição	Parametros
Tempo de Simulação	10000 ns
Clock do Pipeline	15 ns
Clock do Cache	5 ns
accessTimeMemory	200 ns
ciclos de clock	872
Endereços de dados executados	255
Endereços de instrução executados	255
Número Stalls de dados	64
Número Stalls de instrução	64
Tempo stall de dados	616
Tempo stall de instruções	615
Relative Stall Time	$616/872 = 70,6\%$
Miss Rate	$64/255 = 25,1\%$
Access memory Time	70,2%

Tabela 10: Resultados Teste 3

5.4 Teste 4: Hierarquia Básica

- Arquivo: 3fc basico.awc (padronizar nomes dps!)

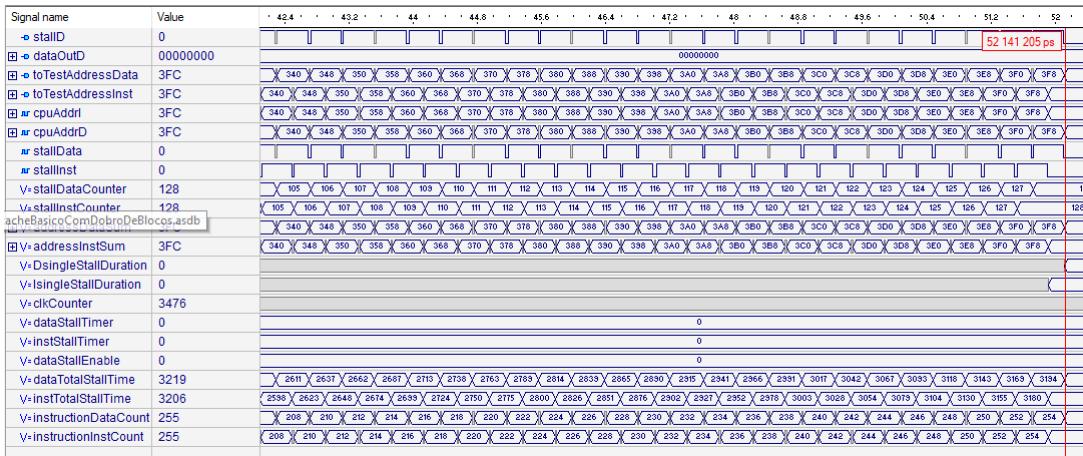


Figura 30: Forma de onda do teste 4.

Descrição	Parametros
Tempo de Simulação	10000 ns
Clock do Pipeline	15 ns
Clock do Cache	5 ns
accessTimeMemory	200 ns
ciclos de clock	3476
Endereços de dados executados	255
Endereços de instrução executados	255
Número Stalls de dados	128
Número Stalls de instrução	128
Tempo stall de dados	3219
Tempo stall de instruções	3206
Relative Stall Time	$3219/3476 = 92.6\%$
Miss Rate	$128/255 = 50.2\%$
Access memory Time	???

Tabela 11: Resultados Teste 4

6 Conclusão

Baseando-se no que fora relatado nesse documento, concluiu-se que o principal objetivo do projeto - observar efeitos de técnicas de otimização - foi atingido.

Apesar de que mais simulações e técnicas poderiam ter sido implementadas (em uma maior disponibilidade de tempo para tal), simulações comprobatórias do que fora implementado foram executadas. Destaca-se também que o dado projeto exigira uma grande carga de estudos, vista a ampla gama de técnicas de otimização distintas, englobando cenários adequados para uso de cada dessas técnicas, vantagens e desvantagens concomitantes.

Acerca desse estudo, ressalta-se que fora amplamente realizado, tal como apresentado no presente relatório. O grupo conseguiu entender muito da problemática da otimização do desempenho de uma hierarquia de memória.

Em suma, a despeito de questões técnicas e do escopo de tempo restrito pelo qual passou o projeto, um bom conhecimento acerca da problemática presente - formas e metodologias de otimização do desempenho de caches - foi devidamente atingido, tendo assim se tornado válida a execução do projeto.

Referências

- [1] Luna Backes Drault. *Evaluation of Cache Inclusion Policies in Cache Management*. Texas AM University. 2017. URL: <https://oaktrust.library.tamu.edu/bitstream/handle/1969.1/166081/BACKESDRAULT-THESIS-2017.pdf?sequence=1&isAllowed=y>.
- [2] John L. Hennessy e David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann. 2009.
- [3] Joel Hruska. *How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips*. ExtremeTech. 2018. URL: <https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips>.
- [4] Joel Hruska. *L2 vs. L3 cache: What's the Difference?* ExtremeTech. 2017. URL: <https://www.extremetech.com/computing/55662-top-tip-difference-between-l2-and-l3-cache>.
- [5] Mikko H. Lipasti. *Cache Replacement Policies*. University of Wisconsin-Madison. 2016. URL: <https://ece752.ece.wisc.edu/lect11-cache-replacement.pdf>.

- [6] Onur Mutlu. *Computer Architecture. Lecture 24: Prefetching*. Carnegie Mellon University. 2011. URL: <https://www.archive.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:lectures:onur-740-fall11-lecture24-prefetching-afterlecture.pdf>.
- [7] PRINCETON UNIVERSITY. *Computer Architecture - Non blocking Caches*. PRINCETON UNIVERSITY. 2017. URL: <https://www.youtube.com/watch?v=0k-LGG1677Y>.
- [8] Heechul Yun. *CachePartitioning on Contemporary COTS Multicore Processors*. University of Kansas. 2017. URL: http://rtsl-edge.cs.illinois.edu/CMAAS17/media/talk_7.pdf.
- [9] Ying Zheng, Brian T. Davis e Matthew Jordan. *Performance Evaluation of Exclusive cache Hierarchie*. Electrical Computer Engineering Department, Michigan Technological University. 2004. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.8452&rep=rep1&type=pdf>.