***Q1:*** *Uma cache está sendo projetada para um computador com 232 bytes de memória, a qual terá 4K slots e usará blocos de 32 bytes. Calcule o número de bytes que a cache terá considerando as duas organizações: associativa e mapeamento direto.*

Como nossa memória principal tem 2³² bytes, sabemos que para endereçar todos precisamos de 32bits. Portanto, o nosso endereço nessa configuração tem 32 bits, o que nos permite definir:

O layout de endereço da cache em **mapeamento direto** seria:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tag(15) | Índice (12) | Block Offset(3) | Byte Offset (2) | |

Para endereçar 4K entradas, são necessários 12 bits para o índice.

Para endereçar um bloco de tamanho 32 bytes, temos 8 words dentro do bloco e portanto são necessários 3 bits para Block offset

Para endereçar uma word de tamanho 4 bytes, são necessários 2 bits para o Byte Offset.

Vamos contar o tamanho que nossa cache teria, em bytes:

4.096 slots com (32\*8 bits + 1 valid bit + 15 bits de tag): 4096\*272 = 1088 Kbits = 136 kbytes

O **tamanho total** da nossa cache é 136 KB.

O layout de endereço da cache na configuração **totalmente associativa** seria:

|  |  |  |  |
| --- | --- | --- | --- |
| Tag(27) | Block Offset(3) | Byte Offset (2) | |

Nessa configuração nossa cache possui apenas um grande “slot”. Assim, não existe índice.

Nesse slot teremos de armazenar:

4.096 slots com (32\*8 bits + 1 valid bit + 27 bits de tag): 4096\*284 bits = 142 KB

O **tamanho total** da nossa cache é 142 KB.

***Q2:*** *Considere referências aos seguintes endereços de memória: 1,4,8,5,20,17,19,56, 9,11,4,43,5,6,9, 17. Calcule o número de faltas e mostre o estado final da cache. Considere uma cache de 16 palavras e blocos de 4 palavras, com as configurações descritas abaixo. Compare os resultados.*

1. *mapeamento direto*
2. *two way set associative*
3. *completamente associativa*

**a)**

Relações de mapeamento de endereço:

Seguindo a ordem fornecida:

* **[MISS]** Endereço 1 mapeia para índice/slot 0, levando consigo todos os dados que possuem endereços de 0 a 3.
* **[MISS]** Endereço 4 mapeia para índice/slot 1, levando consigo todos os dados que possuem endereços de 4 a 7.
* **[MISS]** Endereço 8 mapeia para índice/slot 2, levando consigo todos os dados que possuem endereços de 8 a 11.
* **[HIT]** Endereço 5
* **[MISS]** Endereço 20, mapeia para índice/slot 1 levando consigo todos os dados com endereços de 20 até 23.
* **[MISS]** Endereço 17, mapeia para índice/slot 0 levando consigo todos os dados com endereços de 16 até 19.
* **[HIT]** Endereço 19
* **[MISS]** Endereço 56, mapeia para índice/slot 2, levando consigo todos os dados com endereços de 56 até 59.
* **[MISS]** Endereço 9 mapeia para índice/slot 2, levando consigo todos os dados que possuem endereços de 8 a 11.
* **[HIT]** Endereço 11
* **[MISS]** Endereço 4 mapeia para índice/slot 1, levando consigo todos os dados que possuem endereços de 4 a 7.
* **[MISS]** Endereço 43, mapeia para índice/slot 2, levando consigo todos os dados com endereços de 40 até 43.
* **[HIT]** Endereço 5
* **[HIT]** Endereço 6
* **[MISS]** Endereço 9 mapeia para índice/slot 2, levando consigo todos os dados que possuem endereços de 8 a 11.
* **[HIT]** Endereço 17

**Total Misses:** 10

**Estado Final da Cache:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cache Slot | Endereço dos Bytes | | | |
| 0 | 16 | 17 | 18 | 19 |
| 1 | 4 | 5 | 6 | 7 |
| 2 | 8 | 9 | 10 | 11 |
| 3 | x | x | x | x |

**b)** Relações de mapeamento de endereço:

Seguindo a ordem de acessos:

* **[MISS]** Endereço 1 mapeia para o set 0, levando consigo os dados contidos nos endereços 0-3.
* **[MISS]** Endereço 4 mapeia para o set 1, levando consigo os dados contidos nos endereços 4-7.
* **[MISS]** Endereço 8 mapeia para o set 0, levando consigo os dados contidos nos endereços 8-11. Não sobrescreverá o END. 1 por conta da associatividade.
* **[HIT]** Endereço 5
* **[MISS]** Endereço 20 mapeia para o set 1, levando consigo os dados contidos nos endereços 20-23
* **[MISS]** Endereço 17, mapeia para set 0, levando consigo os dados contidos nos endereços 16-19. O algoritmo de LRU irá retirar o bloco com os endereços de 0-3, por ter sido menos usado recentemente.
* **[HIT]** Endereço 19
* **[MISS]** Endereço 56 mapeia para o set 0, levando consigo os dados contidos nos endereços 56-59. O algoritmo de LRU irá retirar o bloco com os endereços de 8-11, por ter sido menos usado recentemente.
* **[MISS]** Endereço 9 mapeia para o set 0, levando consigo os dados contidos nos endereços 8-11. O algoritmo de LRU irá retirar o bloco com os endereços de 16-19, por ter sido menos usado recentemente.
* **[HIT]** Endereço 11
* **[HIT]** Endereço 4
* **[MISS]** Endereço 43 mapeia para o set 0, levando consigo os dados contidos nos endereços 40-43. O algoritmo de LRU irá retirar o bloco com os endereços de 56-59, por ter sido menos usado recentemente.
* **[HIT]** Endereço 5
* **[HIT]** Endereço 6
* **[HIT]** Endereço 9
* **[MISS]** Endereço 17 mapeia para set 0, levando consigo os dados contidos nos endereços 16-19. O algoritmo de LRU irá retirar o bloco com os endereços de 40-43, por ter sido menos usado recentemente.

**Total Misses:** 9

**Estado Final da Cache:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set | Endereço dos Bytes | | | |
| 0 | 8 | 9 | 10 | 11 |
| 16 | 17 | 18 | 19 |
| 1 | 4 | 5 | 6 | 7 |
| 20 | 21 | 22 | 23 |

**c)**

Layout de Endereço:

|  |  |  |  |
| --- | --- | --- | --- |
| Tag(28) | Block Offset (2) | Byte Offset (2) | |

Sabemos, pelas relações de mapeamento de endereço numa cache completamente associativa, que não existem fórmulas para determinar a posição que o endereço irá ocupar, uma vez que sequer existem índices nesse tipo de cache**.** Quando houver um conflito (que só irá acontecer quando a cache estiver cheia) usaremos LRU para decidir qual elemento da cache será removido.

Seguindo a ordem fornecida:

* **[MISS]** Endereço 1 leva consigo os dados de 0-3
* **[MISS]** Endereço 4 leva consigo os dados de 4-7
* **[MISS]** Endereço 8 leva consigo os dados de 8-11
* **[HIT]** Endereço 5
* **[MISS]** Endereço 20 leva consigo os dados de 20-23
* **[MISS]** Endereço 17 leva consigo os dados de 16-19, substituindo os dados de 0-3.
* **[HIT]** Endereço 19
* **[MISS]** Endereço 56 leva consigo os dados de 56-59, substituindo os endereços de 8-11
* **[MISS]** Endereço 9 leva consigo os dados de 8-11, substituindo os endereços de 4-7.
* **[HIT]** Endereço 11
* **[MISS]** Endereço 4 leva consigo os dados de 4-7, substituindo os endereços de 20-23
* **[MISS]** Endereço 43 leva consigo os endereços de 40-43, substituindo os endereços de 16-19
* **[HIT]** Endereço 5
* **[HIT]** Endereço 6
* **[HIT]** Endereço 9
* **[MISS]** Endereço 17 leva consigo os endereços de 16-19, substituindo os endereços de 56-59

**Total Misses:** 10

**Estado Final da Cache:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Set | Endereço dos Bytes | | | |
| 0 | 40 | 41 | 42 | 43 |
| 8 | 9 | 10 | 11 |
| 16 | 17 | 18 | 19 |
| 4 | 5 | 6 | 7 |

**Q3:** Descreva e compare as principais políticas de escrita em memórias cache citando vantagens e desvantagens de cada uma.

São duas as principais políticas de escrita: **“write back”**  e “**write through”.**

Ambas as políticas tratam de um problema existente ao se usar memórias cache: Devido à hierarquia de memória, o processador irá modificar dados na cache, por ser mais rápida e de fácil acesso. No entanto, os valores na memória principal agora estão desatualizados! Podemos resolver isso mantendo ambas sempre atualizadas: ou seja, sempre que a cache for modificada, a memória principal também será atualizada, em sincronia. A essa técnica damos o nome de **write through.** Esse sistema consegue, de forma simples de se implementar, manter os dados sempre consistentes. O problema disso é que é muito lento fazer isso toda vez que uma mudança for feita na memória. Isso arruinaria todo o propósito de manter uma cache para se aproveitar de sua velocidade.

A outra abordagem, chamada **write back**, mantém um bit extra para cada slot de cache, chamado “**dirty bit**”. Esse bit é responsável por indicar se o conteúdo está “sujo”, ou seja, se ele sofreu modificações. Dessa forma, a memória principal será atualizada, mas somente quando o slot sofrer uma substituição e o dirty bit estiver em 1. Dessa forma, todas as mudanças serão feitas de uma só vez, e somente quando há a troca de slots. A desvantagem de um sistema desses é o aumento da complexidade para sua implementação, por exemplo, num computador multiprocessado, manter a consistência de cache necessita de cuidados a mais.

**Q4:** Descreva as principais técnicas de hardware para minimizar o tempo médio de acesso em memórias cache fazendo uma análise comparativa entre as mesmas.

Vale lembrar que quando falamos de redução de tempo médio de acesso em memórias cache, estamos falando da fórmula:

Então, para diminuirmos o tempo médio, precisamos diminuir qualquer uma das três variáveis do lado direito da equação. Algumas técnicas são:

**Early restart**

Quando um bloco for requisitado e não estiver presente na cache será enviado ao CPU assim que estiver disponível, sem esperar que os dados que ainda estejam porventura sendo transferidos, finalizam a transferência.

**Critical Word First**

O bloco requisitado e não presente na cache será o primeiro a ser lido da memória e transferido, e depois os elementos adjacentes.

**Cache Multi-Nível**

Adicionar mais camadas de cache, de forma a reduzir, no primeiro nível, o tempo de acesso, usando as duas técnicas citadas acima, e, no segundo nível, reduzir a taxa de **miss**, já que a cache de segundo nível é maior que a cache de primeiro nível (embora um pouco mais lenta)

**Memórias mais Largas**

Separar a memória principal em “bancos de memória”, de forma que a cache possa acessar os dados que precisa mais rapidamente no caso de miss.

**Aumento de associatividade**

Com o aumento da associatividade, um mesmo bloco pode ser colocado em mais de um lugar da cache. Quanto mais associativa uma cache for, menos miss ela terá.

**Q5:** Detalhe os problemas de fragmentação interna e de fragmentação externa. Quais as técnicas para minimizar cada um destes problemas?

A *fragmentação interna* é a perda de espaço dentro de uma área de tamanho fixo. Numa [memória secundária](https://pt.wikipedia.org/wiki/Mem%C3%B3ria_secund%C3%A1ria), ela ocorre quando um [arquivo](https://pt.wikipedia.org/wiki/Arquivo_de_computador) ou *fragmento* de arquivo não ocupa completamente o espaço da [unidade de alocação](https://pt.wikipedia.org/wiki/Unidade_de_aloca%C3%A7%C3%A3o) destinado a ele, causando desperdício de espaço.

Fragmentação externa se dá em alocação dinâmica, os programas vão preenchendo os espaços da memória. No término de programas contidos em espaços intermediários, ocorrem vãos de memória livre, que pode não ser suficiente para programas maiores.

***Q6.****Considere um sistema de memória virtual implementado como paginação com as seguintes características:*

*a. espaço de endereçamento virtual de 16 GBytes*

*b. memória principal de 64 MBytes com páginas de 8 Kbytes*

*Qual o layout do endereço virtual e das tabelas de tradução? Considere que cada entrada na tabela possui 1 bit de presença e um dirty bit. Explique como o endereço virtual é traduzido num endereço físico, neste caso.*

Para endereçar 16Gb de memória **virtual** precisamos de 34 bits.

Para endereçar 64Mb de memória **física** precisamos de 26 bits.

As páginas tem 8Kb então para endereçar uma página precisamos de 13 bits.

Layout do endereço **virtual**:

|  |  |  |
| --- | --- | --- |
| Virtual Page Number(21) | Page Offset(13) | |

Layout do endereço **físico**:

|  |  |  |
| --- | --- | --- |
| Physical Page Number(13) | Page Offset(13) | |

Layout da Tabela de Páginas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dirty(1) | Valid(1) | Physical Page Number(13) | | |

Layout da TLB **(Assumindo Fully Associative):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dirty(1) | Valid(1) | **Tag** / Virtual Page Number(21) | Physical Page Number(13) | |

**Como o Endereço Virtual é Traduzido em um Endereço Físico:**

Quando um programa requisita um endereço, ele se encontra no Layout de Endereço virtual. Portanto, necessita ser traduzido para a posição onde se encontra na memória física. O hardware responsável por essa tradução é a **MMU** (memory management unit). Como o endereço requisitado pode estar “no meio” de uma página, separamos os 13 bits menos significativos para endereçar uma página internamente - page offset -. Assim, os 21 bits restantes compõem o nosso endereço virtual, que a MMU irá traduzir para o endereço correspondente de 13 bits (pois nossa memória principal é de 64Mb).

Essa tradução é feita através de uma tabela de páginas. Usando os 21 bits de página virtual como **tag**, conseguimos através da tabela saber quais os 13 primeiros bits do endereço **físico** correspondente. Caso o endereço esteja na tabela, basta concatenar os bits encontrados com o offset de página, e temos nosso **physical page address.** Quando um endereço não for localizado na tabela, isto é, sua página não estiver na memória principal, **(bit de validade 0)** uma exceção será lançada. A isso damos o nome de **page fault**. Agora, a MMU deve esperar que a página seja trazida do disco ou da **swap area**, antes de poder converter para o endereço físico.

***Q7.*** *Faça uma análise comparativa dos diferentes modos de tradução de endereços.* Sim!

**Paginação:** esse modo de tradução consiste em dividir a memória em blocos que são chamados de páginas. A alocação de memória é requisitada por essas páginas, sendo a unidade de transferência entre a memória e o disco. É necessário uma tabela de páginas para saber onde qual endereço virtual corresponde na memória física.

Uma vantagem dessa técnica é a impossibilidade de acontecer fragmentação externa, já que as páginas de um processo não precisam ficar contínuas na memória. A desvantagem é a existência da fragmentação interna, ou seja, desperdício de espaço por consequência do tamanho de páginas ser fixo.

**Segmentação:** já nesse modo, os processos são divididos em segmentos de tamanhos variados. A alocação é feita por uma tabela de segmentos, que guarda o início do segmento e o tamanho.

É o contrário da paginação: pode existir fragmentação externa mas não pode existir fragmentação interna. Outra vantagem é a facilidade de proteção, já que cada processo tem seu segmento e tamanho determinados.

**Segmentação Paginada:** para contornar as desvantagens anteriores, é possível dividir os segmentos em páginas, esta também permite que segmentos maiores que a memória possam ser carregados. Uma desvantagem é quantidade de tabelas, aumentando o acesso à memória.

***Q8.*** *Considere que o sistema de memória da questão anterior possui uma memória cache de 4K slots com blocos de 32 bytes organizada como four-way associativa. Como a cache é acessada, qual o layout e o tamanho da mesma em bytes? (Assuma que cada slot possui um bit de validade adicional e 1 dirty bit)*

Assumindo um endereço físico de memória de 26 bits, como na questão anterior, temos:

Como os blocos tem 32 bytes, temos **8 palavras por bloco**, e precisamos de 3 bits para o **block offset**.

Com 4K slots numa configuração 4-way set associative, teremos

Desse modo, precisamos de **10 bits para endereçar todos os sets** da cache, ficando com o seguinte layout de endereço:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tag(11) | Set (10) | Block Offset (3) | Byte Offset (2) | |

O tamanho de nossa cache será:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | D | Tag | Data | Data | Data | Data | Data | Data | Data | Data |

Para acessá-la, é necessário que o gestor de cache, através do endereço fornecido pela CPU, decodifique e obtenha seus offsets e tag, obtendo as respectivas tags contidas em cada conjunto e comparando com a extraída do endereço.

***Q9.*** *Para acelerar a tradução de endereços o sistema possui uma TLB two-way set associative com 1K entradas. Como se dá a tradução de endereços via TLB, qual o layout da TLB e quantos bytes a mesma ocupa?*

**Layout de endereço da TLB:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tag (virtual page number) (12) | | | Set(9) | Physical Page Number (13) |

**Layout da TLB:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Valid(1) | Dirty(1) | Tag (virtual page number) (12) | Physical Page Number (13) | |

**Tamanho da TLB (bytes)**

Teríamos 1K = 2¹⁰ slots, mas, como é uma two-way associative, teremos 2⁹ slots. A tag é baseada na VPN, que possui 21 bits, como 9 desses bits já estão implícitos nas linhas, precisaremos de uma tag de 12 ( = 21 - 9 ) bits. Precisaremos armazenar 13 bits para PPN e mais 2 bits, um valid e outro dirty. No total:

**Passo a passo da tradução:**

A TLB serve como uma cache para a tabela de páginas, devido ao modo como o acesso é realizado. Como a tabela de páginas se encontra na RAM, e é necessário acessá-la para converter o endereço virtual num endereço físico (ou seja, um endereço da RAM), se fizéssemos isso toda vez que uma página fosse referenciada, acessaremos a RAM 2x, o que deixa a performance terrível.

Fazendo uso de uma TLB podemos contornar esse problema, olhando primeiro na TLB se já acessamos essa página recentemente. A TLB guarda a tag da página e o número de página físico. Assim, a tradução é feita muito mais rapidamente, uma vez que não foi preciso acessar a RAM para olhar a tabela de páginas. Com isso, páginas frequentemente acessadas ficam gravadas na TLB, e podem ser traduzidas sem acessar a RAM desnecessariamente. Leva-se em conta que se referencia poucas páginas muitas vezes.