Tópico 6

Sistema de Memória: Organização e Arquitetura

Autores: José Raimundo de Oliveira e Wu Shin-Ting
DCA - FEEC - Unicamp
Setembro de 2019

Vimos no Capítulo 5 que, em decorrência do desempenho e do custo das memórias, é muito comum usar num projeto digital diferentes tecnologias para compor um sistema de memória. Vimos ainda que memórias de acesso mais rápido, porém de custo mais alto, são aplicadas diretamente no armazenamento de dados e instruções necessárias para execução de uma instrução programada, enquanto unidades de armazenamento mais lento, porém mais barato e de capacidade maior, são reservadas para armazenamento em massa. É comum organizar estes componentes numa **hierarquia de memória** conforme seus tempos de acesso, como mostra a Figura 6.1, para dar melhor suporte a um microprocessador.

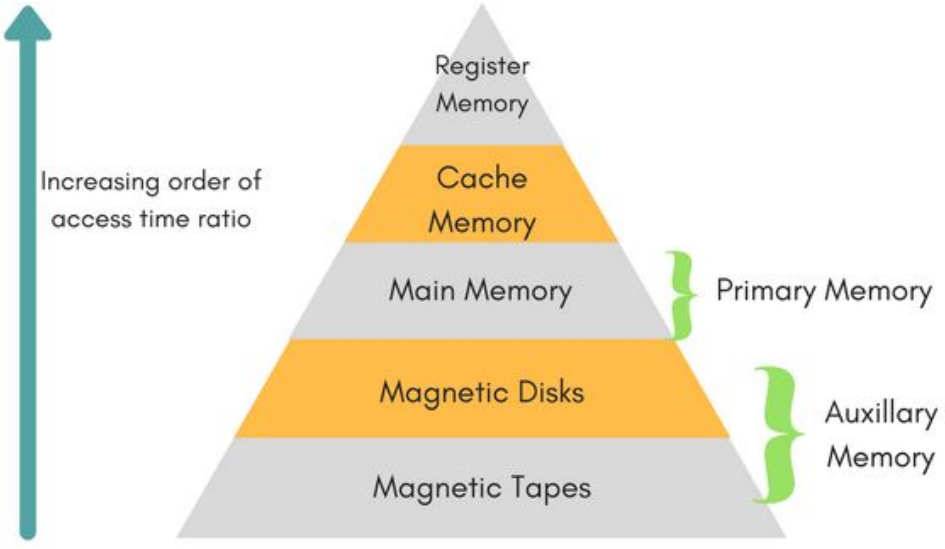


Figura 6.1: Hierarquia da memória.

Além dos registradores que são integrados na CPU para armazenar os operandos a serem processados pela CPU (Seção 4.1), vimos no Capítulo 4 que as instruções

processadas pela CPU são buscadas da memória (principal). Pela sua capacidade de armazenamento e pelo seu custo, as memórias RAM dinâmicas (DRAMs, Seção 5.1.2.2) são as mais aplicada na implementação da memória principal de um sistema computacional moderno. No entanto, a discrepância entre a velocidade de processamento da CPU e das DRAMs tem crescido exponencialmente. Até em torno de 2001-05, a frequência do relógio das CPUs aumentaram em torno de 55%, enquanto a velocidade das DRAMs cresceu anualmente em 7% [3], gerando uma **barreira de memória**, em inglês *memory wall*, A **memória *cache***, de menor latência, menor capacidade de armazenamento e maior custo, foi uma solução encontrada para superar esta barreira e melhorar o fluxo de dados e instruções entre a CPU e a memória principal. Essas memórias são formadas pelas memórias RAM estáticas (SRAMs, Seção 5.1.2.1)

A memória *cache* é tipicamente integrada ao processador, porém não faz parte do conjunto de registradores. Tem uma capacidade de armazenamento maior do que a dos registradores e muito menor do que a memória principal. Quando não está integrada na CPU, ela é conectada com a CPU por um barramento especial, de velocidade maior, conhecido como **barramento por trás**, em inglês *back-side bus*, enquanto a conexão da CPU com as outras unidades se dá pelo **barramento frontal**. Quanto à memória secundária de armazenamento em massa, como os discos e fitas magnéticas, ela é organizada numa estrutura conhecida por **sistema de arquivos** para facilitar acessos a uma informação específica. As interfaces destes dispositivos de massa foram descritas brevemente na Seção 5.4.

Figura 6.2 sintetiza as características funcionais, temporais e o custo das diferentes tecnologias de memória que podem fazer parte de um sistema de memória. Ressaltamos aqui que, embora a CPU não consiga se comunicar diretamente com uma memória de massa, a unidade de gerenciamento de memória permite estender o espaço de endereçamento físico para o espaço de endereçamento virtual do tamanho de um disco rígido através da **técnica de paginação** (Seção 5.6).

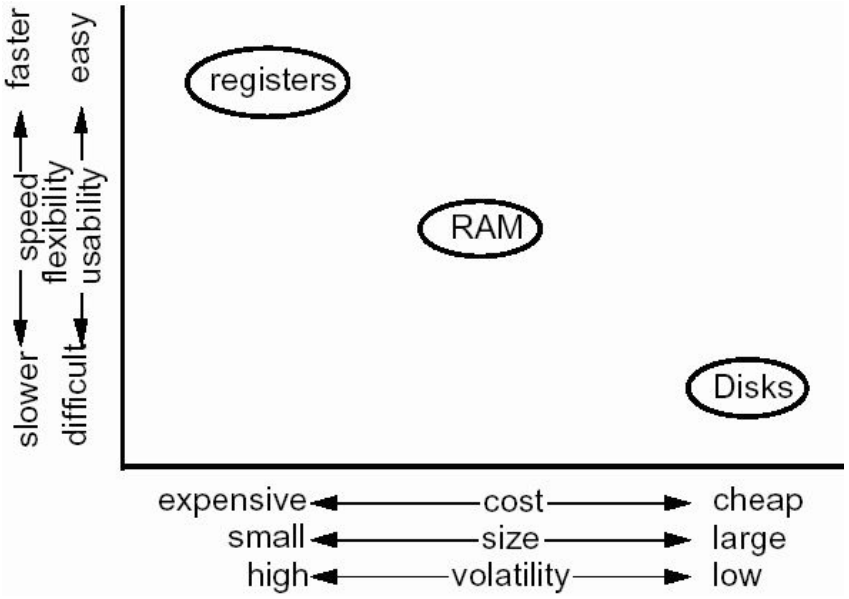


Figura 6.2: Características dos componentes de um sistema de memória (Fonte: [1]).

Com o sistema de memória organizado hierarquicamente conforme os seus tempos de acesso, fica mais fácil otimizar as transferências de dados entre a CPU e o sistema de memória. Quando se executa um programa, a CPU executa a instrução que estiver no seu registrador de instrução e busca a próxima instrução na memória *cache*. Caso esta não estiver na memória *cache*, busca-se na memória principal. E, se não a encontrar na memória principal, tenta-se por último pelo disco rígido antes de retornar um erro. É fácil perceber que, quanto mais buscas são feitas, mais lentos são os acessos. Isso pode comprometer o desempenho do sistema. Neste capítulo vamos apresentar algumas estratégias para reduzir número de insucessos na sequência de busca por instruções.

6.1 Memória Principal

São duas as características das interfaces entre uma CPU e uma memória principal: a latência e a largura de banda. A **latência** é o tempo necessário para uma CPU completar um ciclo de acesso, seja de leitura ou de escrita. A **largura de banda** tem a ver com a taxa de transferência de *bytes* entre a CPU e a memória. Usualmente, é expressa na unidade de *bytes*/segundo. Porém, duas características da memória principal impactam diretamente nestas características: **localidade** e **capacidade de armazenamento**.

A **localidade** diz respeito à posição dos dados a serem acessados, permitindo a predição de padrões e, portanto, a otimização em acessos. Dois tipos de localidades relevantes são **temporal** e **espacial** (ou **de referência**). Localidade temporal está relacionada com os dados que são acessados num intervalo curto de

tempo, enquanto a localidade espacial tem a ver com a proximidade dos dados nos acessos. Quanto maior a localidade, menor será a latência. A **capacidade de armazenamento** é o tamanho do espaço de endereçamentos da memória principal (Seção 5.6). Esta capacidade é um limitante ao tamanho de um aplicativo que um sistema computacional consegue executar. Para contornar esta limitação, vimos na Seção 5.6 que os sistemas operacionais modernos conseguem criar virtualmente um espaço de memória ilimitado, fazendo de forma transparente as trocas de páginas de dados inativos na memória principal pelas páginas de dados hospedadas numa memória de massa, tipicamente um disco rígido [6].

Além do problema da capacidade de armazenamento dos aplicativos, cujos tamanhos crescem exponencialmente, tem-se ainda o problema do crescimento exponencial na diferença entre as velocidades das CPUs e das DRAMs. Isso impacta diretamente na latência. Até os anos 80s, CPUs e DRAMs eram temporalmente compatíveis. A partir dos anos 80s, como mostra a Figura 6.3, os microprocessadores tiveram um desenvolvimento muito mais acelerado do que as DRAMs em termos da frequência de operação. Um dos fatores que dificultam o aumento do desempenho das DRAMs é a necessidade de regeneração periódica (Seção 5.1.2.2). Enquanto as frequências das CPUs chegam à ordem de GHz, as DRAMs não conseguiram ultrapassar do limite de 10MHz! Como já mencionamos, isso motivou à introdução de SRAMs como memórias *cache* entre as DRAMs e as CPUs. Pois, o tempo de acesso às memórias SRAMs, que não precisam da regeneração periódica, pode chegar na ordem de nanosegundos por ciclo.

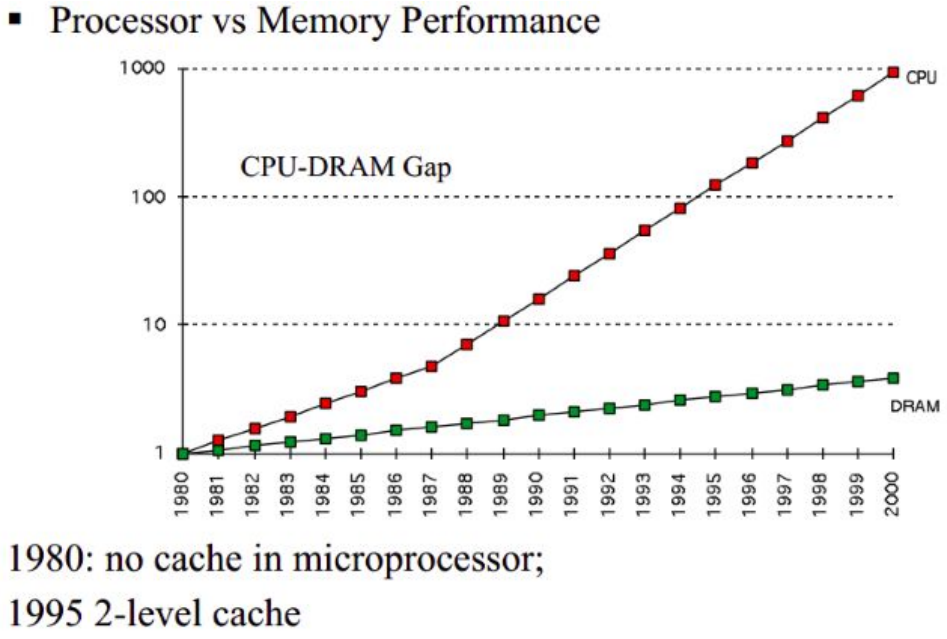


Figura 6.3: Evolução da discrepância entre as velocidades de operação dos microprocessadores e das DRAMs (Fonte: [2])

Embora a técnica de memória *cache* tenha como o foco a redução da latência e a técnica de memória virtual almeje ao aumento da capacidade de armazenamento, é importante observar que ambas as técnicas compartilham um mesmo princípio de usar elementos auxiliares à memória principal para melhorar, respectivamente, a latência e a capacidade de armazenamento do sistema computacional.

6.2 Memória Cache

Memórias *cache*, também conhecidas como memórias da CPU, são as SRAMs introduzidas no sistema de memória para reduzir as latências. Com base nas estimativas e suposições acerca o programa em execução, a unidade de gerenciamento de memória (Seção 5.6) procura fazer uma cópia antecipada da instrução e dos dados necessários à sua execução nessa memória *cache*, de forma que, quando o processador busca uma instrução/um dado, tenha-se **acerto em *cache***, em inglês *cache hit*. Se a grande maioria dos dados buscados se encontra em *cache*, a latência média do sistema é menor do que num sistema de memória sem *cache*. Veja na Figura 6.4 o princípio de organização de uma memória *cache* em relação a uma CPU e a uma DRAM (memória principal). Note que os acessos às DRAMs, de latência maior, só acontecem se ocorrer **falta em cache**, em inglês *cache miss*.

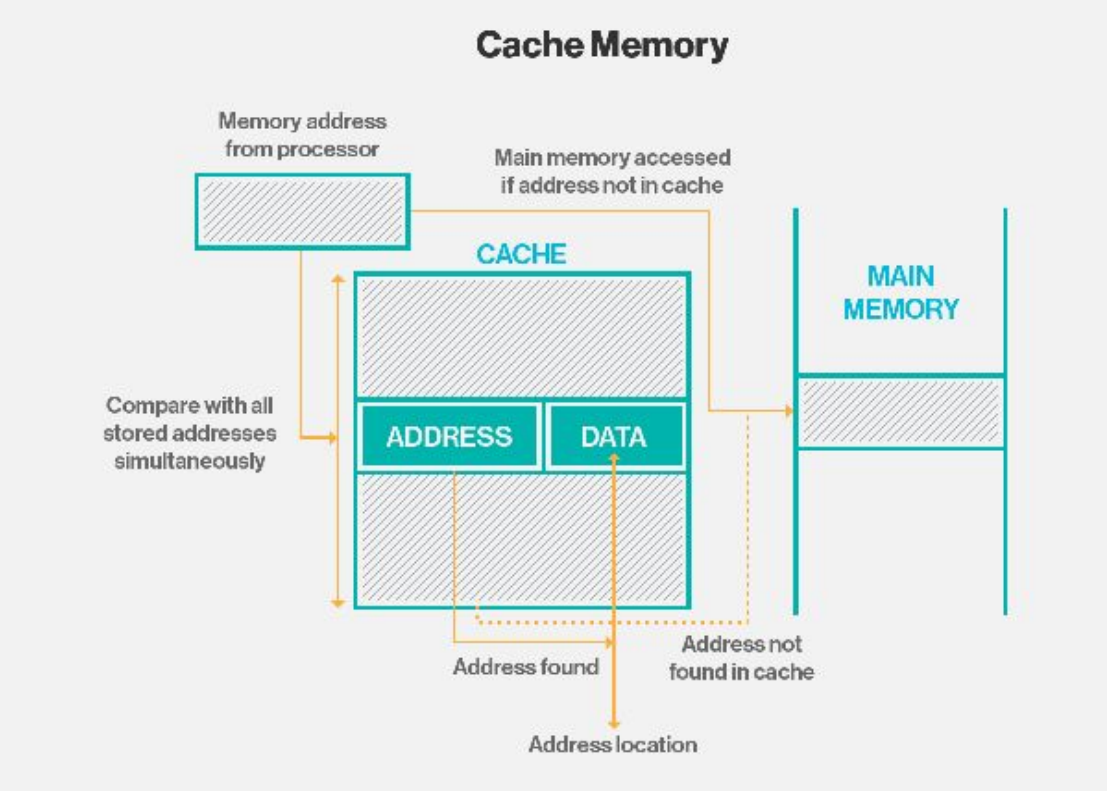


Figura 6.4: Organização da memória *cache* e da memória principal (Fonte: [4]).

Além de antecipar a busca dos dados armazenadas nas DRAMs, uma outra estratégia adotada nas memórias *cache* é o aumento da largura de banda nas transferências. As memórias cache fazem transferências por **linhas de cache**, em inglês *cache lines*, ao invés de acessos por *bytes* através dos barramentos tradicionais de dados, como vimos no Capítulo 4. Estas linhas de *cache* são, de fato, cópias dos blocos de 8, 32, 64 ou 128 *bytes* da memória principal. A pergunta que se faz é como localizar os dados nestes blocos com os endereços tipicamente usados pela CPU e detectar a sua falta em *cache* (*cache miss*) ou acerto em *cache* (*cache hit*) (Figura 6.5)?

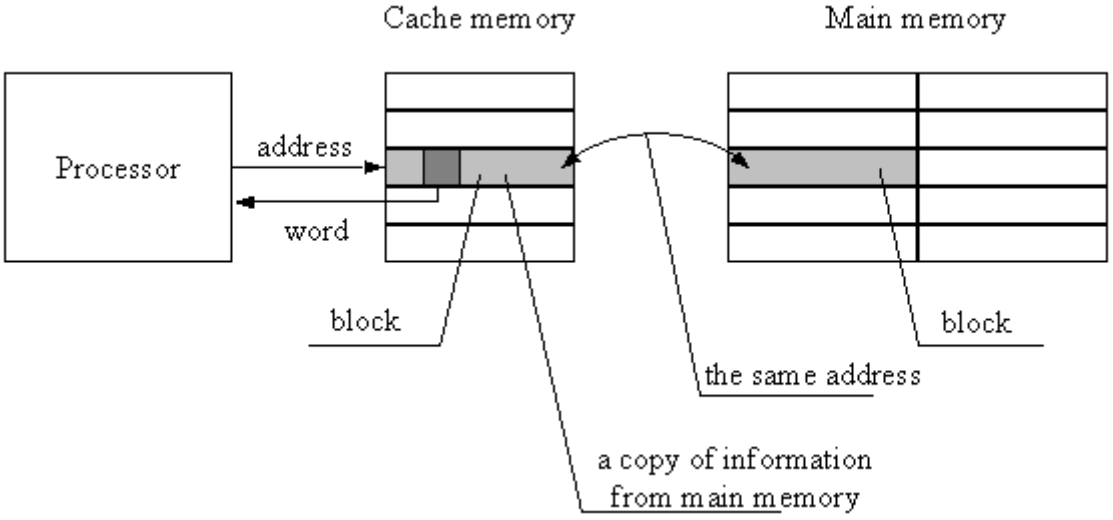


Figura 6.5: Mapeamento entre o espaço de endereços na CPU e o espaço de endereços na memória *cache* (Fonte: [21]).

Distinguem-se três técnicas para mapear os endereços da memória principal, utilizadas pela CPU, num endereço da memória *cache*:

- **mapemento direto**, em inglês *direct-mapping cache*: esta técnica é a mais simples. Ela mapeia explicitamente cada linha de *cache* num **conjunto fixo de blocos** da memória principal. Seja uma memória *cache* de 2^m linhas de *cache*, contendo cada linha 2^n bytes, e um espaço de memória principal igual a 2^{k+m+n} *bytes* , cada linha de cache é associada igualmente a 2^k blocos da memória principal. uma linha de cache de tamanho igual a 2^m *bytes*. O endereço da memória principal de *k+m+n* bits é dividido em três campos: os *n bits* menos significativos correspondem aos endereços dos 2^n bytes em cada linha, os *m bits* subsequentes representam 2^m linhas que existem na memória cache, e os últimos *k bits* mais significativos correspondem a um dos 2^k blocos aos quais uma linha de cache foi associada. Dado um endereço A, pelo seus *m bits* (*slot* na Figura 6.6) consegue-se localizar a linha do *cache* correspondente. Observe na Figura 6.5 que há dois *flags* associados a cada bloco, "*Valid*" e "*Dirty*". O primeiro *flag* é para indicar se o

conteúdo de um bloco é válido ou não, e o segundo é para indicar se o bloco foi alterado ou não. Identificado o bloco correspondente ao endereço A e verificado o *bit* “Valid”, comparamos então os *k bits* mais significativos de A com o campo *Tag* associado ao bloco identificado. Não é possível a coexistência de blocos distintos associados a um mesmo bloco (*Slot*) nesta técnica de mapeamento.

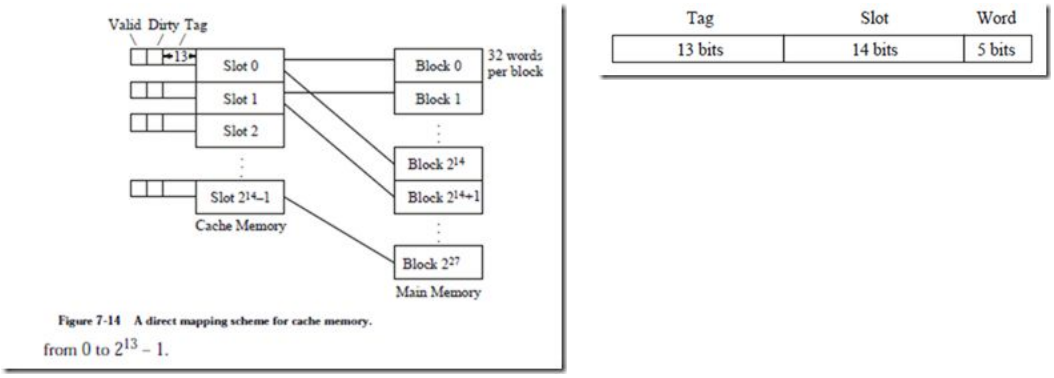


Figura 6.6: Mapeamento direto (Fonte: [7]).

Figura 6.7 mostra um circuito combinacional que gera o sinal *cache hit* (dado encontrado) com base na comparação dos *bits* de *Tag* de um endereço em busca com o campo *Tag* do bloco da memória *cache* correspondente.

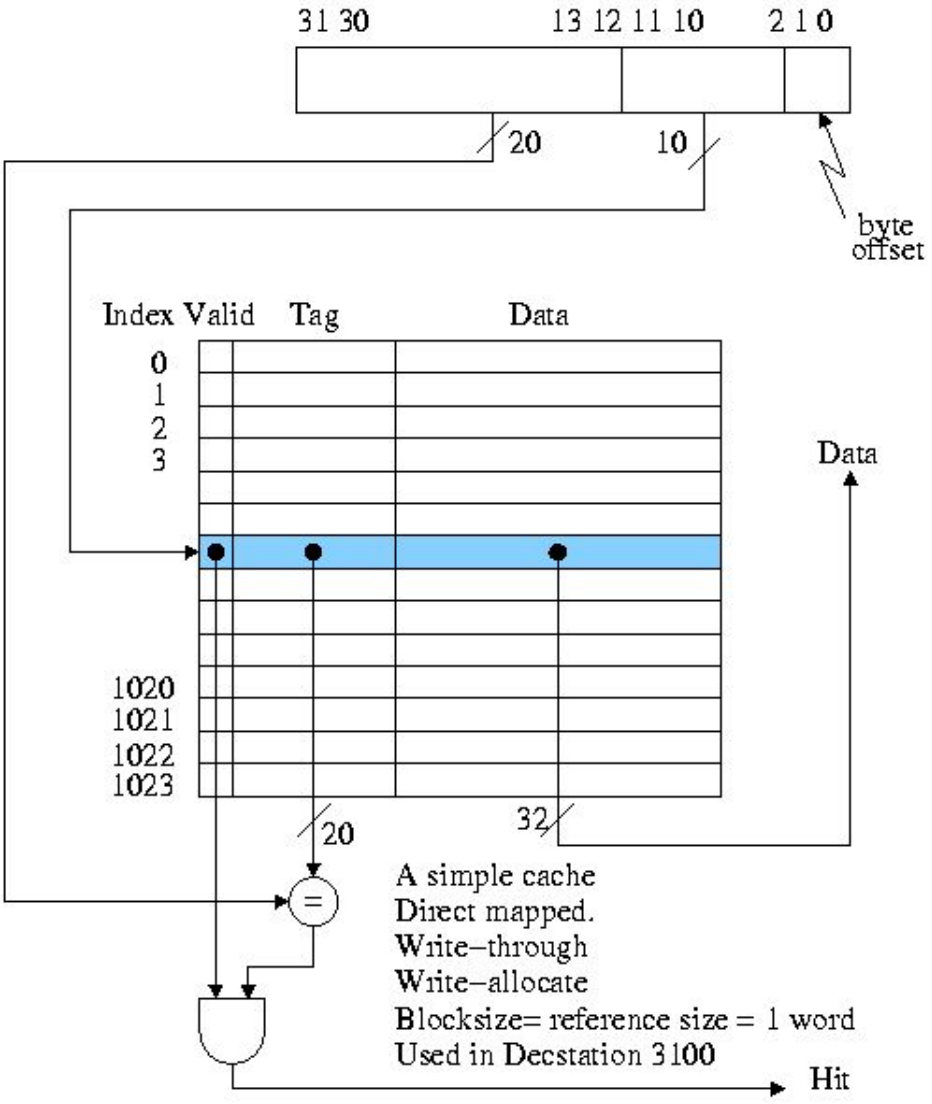


Figura 6.7: Circuito de detecção da presença de um dado na memória *cache* cuja linha de *cache* é 4 *bytes* com 2^{10} linhas(Fonte: [9]).

- **memória associativa completa**, em inglês *fully associative cache*: O mapeamento fixo de um bloco da memória principal para um único bloco da memória *cache* é o principal problema da técnica de mapeamento direto. Uma alternativa seria permitir que um bloco da memória principal possa ser copiado para qualquer linha (bloco) da memória *cache*. Neste caso, os *n+m bits* (*Tag*) correspondem aos endereços dos blocos da memória principal, como mostra a Figura 6.8.

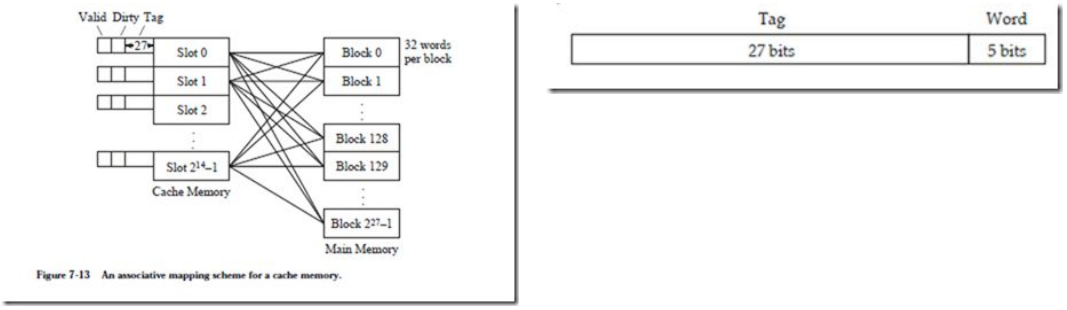


Figura 6.8: Mapeamento associativo (Fonte: [7]).

O custo da flexibilização da técnica de memória associativa na cópia dos blocos da memória principal para a memória *cache* são: (1) a decisão pelo bloco que deve ser trocado quando ocorre um *cache miss*, e (2) a complexidade de um circuito de comparação paralela de todos os *Tags* simultaneamente. Para o primeiro problema, as mesmas **políticas de substituição de páginas** adotadas pelo sistema operacional [6] são usadas. As mais aplicadas são **menos recentemente usado**, em inglês *least recently used* (LRU), **primeiro-entra primeiro-sai**, em inglês *first-in first-out* (FIFO), **menos frequentemente usado**, em inglês *least frequently used* (LFU), **aleatório**, em inglês *random*. Para o segundo problema, é fácil perceber que um circuito de comparação paralela pode ficar tecnologicamente infactível se a quantidade de blocos na memória *cache* for muito grande. Figura 6.9 ilustra um comparador 3 blocos.

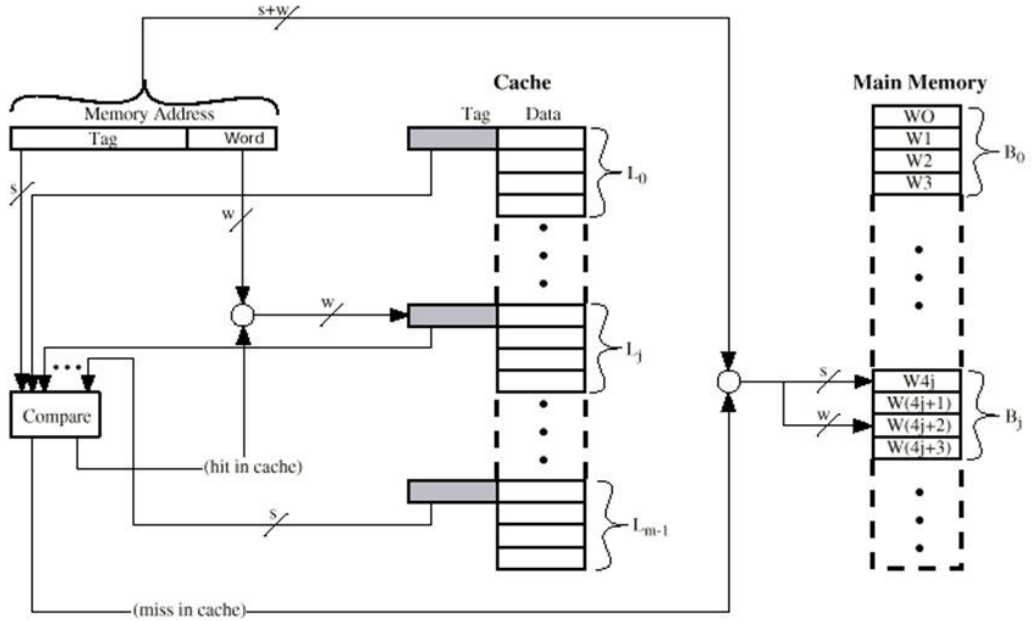


Figura 6.9: Circuito de detecção de falta e acerto em *cache* pela técnica de memória associativa completa (Fonte: [10]).

- **memória associativa por conjunto**, em inglês *set-associative cache*: É uma resposta ao problema de busca pelo *Tag* na técnica de memória associativa. Nesta técnica, os *m bits* do campo *Set* (campo *Set* na Figura 6.10) correspondem a um grupo de *j* "memórias *cache*" mostradas na Figura 6.6. Portanto, ao invés de ter somente um bloco na memória *cache* associado a um *Slot*, podemos ter mais de um bloco, tipicamente 4 blocos.

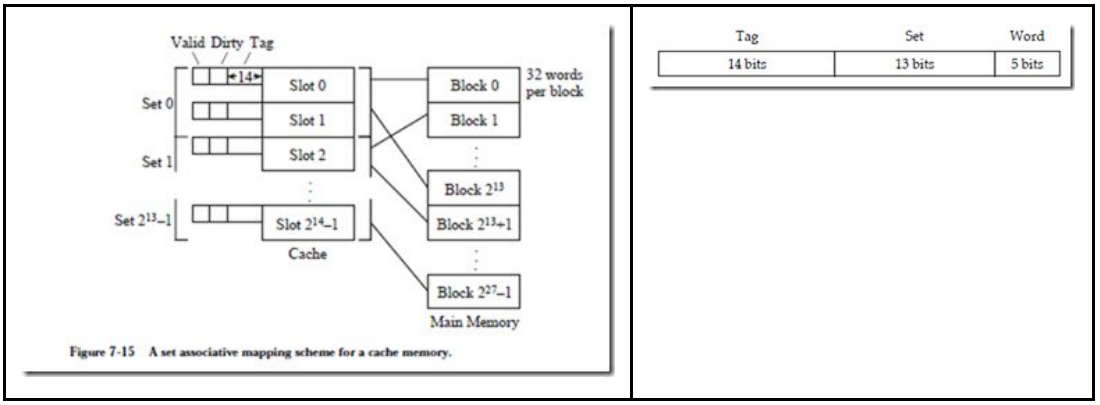


Figura 6.10: Mapeamento associativo por conjuntos (Fonte: [8])

Um comparador adicional é adicionado para comparar os endereços das diferentes linhas (*Tag*) associadas a um mesmo *Slot*. Assim, ganhamos a flexibilidade de copiar mais de um bloco da memória principal com *m* bits iguais para a memória *cache*, sem aumentar exageradamente as entradas no circuito comparador, como mostra a Figura 6.11.

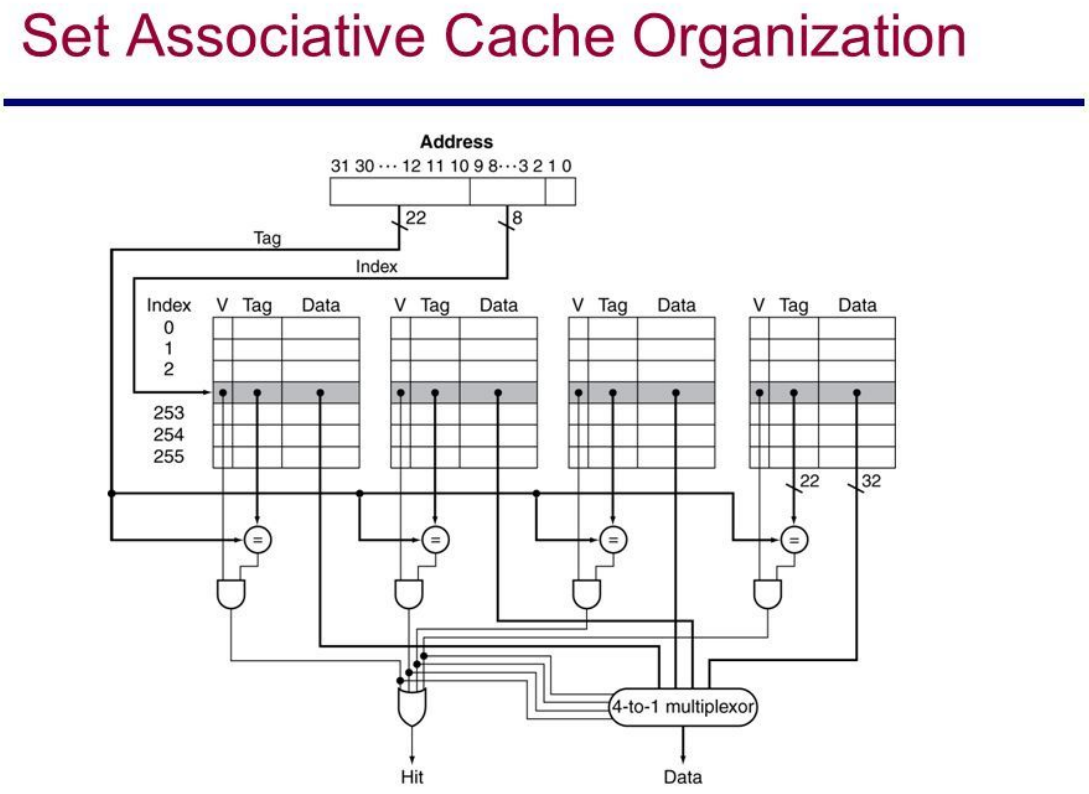


Figura 6.11: Circuito de detecção de falta e acerto em cache pela técnica de memória associativa por conjunto (Fonte: [11]).

Uma vez detectada a presença (*cache hit*) ou não (*cache miss*) do endereço requisitado na memória *cache*, existem ainda diferentes políticas para tratar os dois casos em ciclos de leitura e de escrita. Figura 6.12 sintetiza as diferentes políticas. Observe que, quando o dado é modificado na memória *cache*, a memória principal deve ser consistentemente atualizada. O *bit* “Dirty” mostrado nas Figuras 6.6, 6.8 e 6.10 serve para mostrar o estado de modificação de cada linha de *cache*.

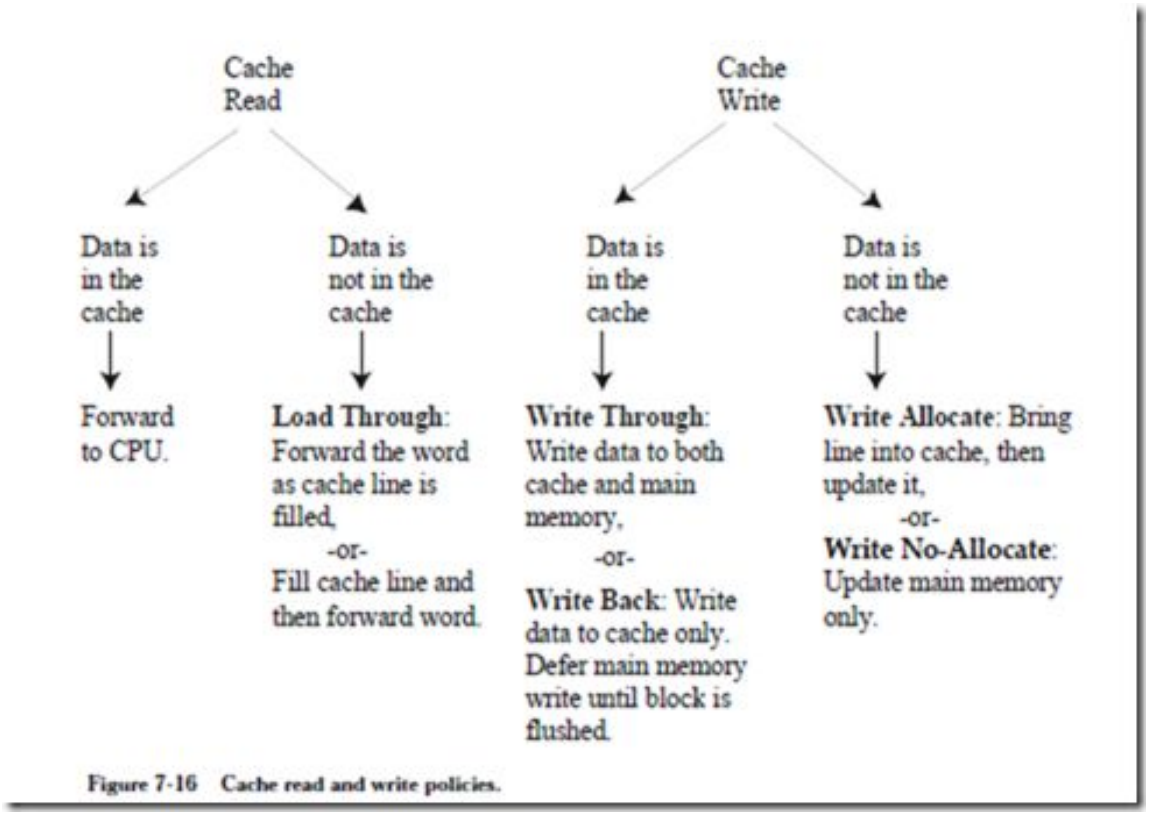


Figura 6.12: Estratégias para leitura e escrita numa memória *cache* (Fonte: [8]).

Para avaliar quantitativamente o desempenho de uma memórias *cache*, introduzimos duas métricas: **taxa de hit**, em inglês *hit ratio*, e **taxa de *miss***, em inglês *miss ratio*.

$$H = taxa\ de\ hit\ = \frac{Número\ de\ cache\ hits}{N},$$

$$M = taxa\ de\ miss\ = \frac{Número\ de\ cache\ misses}{N} = 1 - H,$$

onde N é o número total de acessos à memória com *cache*. A taxa de *hit* ou a taxa de *miss* é, de fato, uma estatística dos acessos à memória. Ela pode variar muito com a localidade de referência do programa em execução. Usualmente, um programa com uma elevada localidade de referência resulta numa taxa de *hit* alta, quase em torno de 98% [22].

Outra métrica é o **tempo de acesso efetivo** , em inglês *effective access time*. Ela estima o tempo médio de um acesso numa memória com *cache*, levando em conta o tempo de acesso à memória principal t_m, ou tempo por *miss*, e o tempo de acesso à memória *cache* t_c

$$TAE = tempo\ de\ acesso\ efetivo\ = \frac{(Número\ de\ cache\ hits)(t_c)+(Número\ de\ cache\ misses)(t_m)}{N} = Ht_c + Mt_m.$$

Com isso, podemos estimar a **taxa de aceleração** *S*, em inglês *speedup ratio*, introduzida com o uso de memória *cache*

$$S = \frac{t_m}{Ht_c + Mt_m} = \frac{t_m}{Ht_c + (1-H)t_m} .$$

Tipicamente, as memórias *cache* integradas na CPU tem uma frequência 100 vezes maior do que a frequência da memória principal, porém com uma capacidade de armazenamento muito menor, em torno de 256KB. Este tamanho é limitado pela tecnologia de integração na pastilha da CPU uma memória de tamanho maior. Daí, surgiu a ideia de adicionar outras unidades com capacidade de armazenamento maior fora da CPU, porém conectadas a ela via barramentos com alta taxa de transferência, e distingui-las em diferentes níveis de *cache* (Figura 6.13) [5]:

- L1: é a memória *cache* primária, integrada à CPU. Ela é extremamente rápida, com uma latência em torno de 1ns, porém com uma capacidade de armazenamento bem limitada, tipicamente em torno de dezenas *kilobytes*. Usualmente, os dados e as instruções são separados em **cache de dados**, em inglês *data cache*, **cache de instruções**, em inglês *instruction cache*. Neste caso, o circuito de *cache* de instruções pode ser mais simples, uma vez que as instruções nunca são alteradas durante a execução de um programa.
- L2: é a memória *cache* secundária, usualmente localizada fora da CPU mas integrada à pastilha do processador. Como está fora da CPU, é possível aumentar o seu tamanho para 256KB a 8MB. Embora seja conectada por um barramento dedicado à CPU, a sua latência, tipicamente em torno de 7ns, é bem maior que a *cache* L1. Usualmente, as instruções e os dados não são separados neste nível de *cache*. Porisso, é também conhecido por **cache unificado**, em inglês *unified cache*.
- L3: é a memória *cache* compartilhada pelos núcleos de um processador multi-núcleo, usualmente colocada na placa-mãe e conectada aos processadores por barramentos de alta velocidade. Embora a sua velocidade seja menor do que a *cache* do nível L2, é ainda mais rápida que a memória principal.

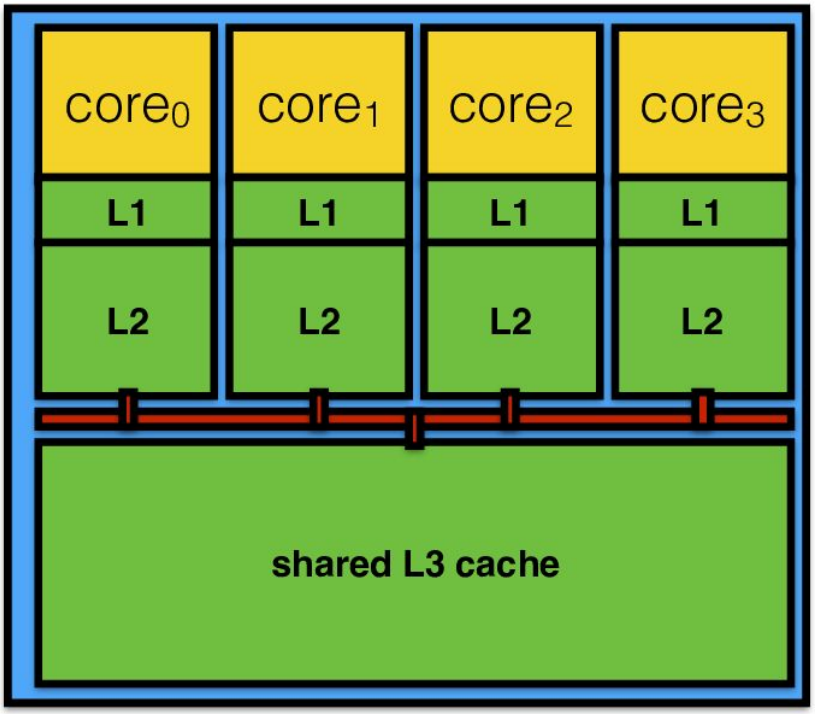


Figura 6.13: Níveis de organização de uma memória *cache* (Fonte: [24]).

Para uma organização multinível de memória *cache*, as fórmulas para cômputo das métricas de desempenho são reformuladas. Por exemplo, para uma organização de dois níveis, as taxas de *hits* em L1 e em L2 ficam, respectivamente,

$$H1 = taxa\ de\ hit\ em\ L1 = \frac{Número\ de\ cache\ hits\ em\ L1}{N}$$

$$H2 = taxa\ de\ hit\ em\ L2 = \frac{Número\ de\ cache\ hits\ em\ L2}{N}$$

$$TAE = \frac{(N.\ de\ hits\ em\ L1)(tempo/hit\ em\ L1)+(N.\ de\ hits\ em\ L2)(tempo/hit\ em\ L2)+(N.\ de\ cache\ misses)(tempo/miss)}{N}$$

6.3 Memória Virtual

Vimos na Seção 5.3 uma variedade de tecnologias para memória secundária, ou memória de massa. Esta memória se comunica com a CPU através do barramento de periféricos, a uma velocidade bem menor do que a sua comunicação com a memória principal. Mesmo assim, os sistemas operacionais modernos [6] conseguem estender virtualmente a capacidade de armazenamento da memória principal para a do disco rígido, formando um **sistema de memória virtual**.

Na Seção 5.6 vimos que o sistema operacional (um *software*), com suporte da unidade de gerenciamento de memória MMU, é o responsável pelo mapeamento dos endereços virtuais para os endereços físicos [22]. Comparando as Figuras 6.5, 6.7 e 6.9 com a Figura 5.42, percebe-se uma certa similaridade na solução da memória de *cache* com a memória virtual. Em ambas as técnicas, o espaço de

endereços é dividido em linhas/blocos/páginas de tamanho igual e quando ocorre a falta de uma página na memória principal, as mesmas políticas de substituição são aplicadas (Seção 6.2). Porém, diferente da técnica de mapeamento de *cache*, o endereço utilizado pela CPU é virtual e precisa ser mapeado para um endereço físico da memória principal.

Para os projetos de sistemas embarcados, em que se priorizam o desempenho, a consistência no tempo de resposta e a área de ocupação, a memória virtual não parece ainda ser uma opção atraente.

6.4 Sistema de Arquivos

Na Seção 5.6 explicamos que num sistema de memória virtual, quando o dado demandado não estiver na memória principal, a execução é interrompida e o sistema operacional busca a página que contém o dado demandado na memória secundária, tipicamente no disco rígido, e transfere-a para a memória principal. Na seção 5.3.1 vimos que a organização e os acessos aos discos rígidos são bem diferentes dos acessos à memórias ROM, sRAMs e DRAMs. Os acessos aos discos rígidos são tipicamente por setores de tamanho típico de 256 *bytes*. As regras que organizam e controlam acessos aos dispositivos de armazenamento de massa são denominados os **sistemas de arquivos**.

Há uma grande variedade de sistemas de arquivos para atender as especificidades dos dispositivos de armazenamento, como sistema de arquivos de disco rígido híbrido, em inglês *hybrid hard drive* (HDD), que abrange o disco rígido (HD) e os dispositivos de armazenamento de tecnologia flash NAND (*File Allocation Table* - **FAT**, *New Technology File System* - **NTFS**), sistema de arquivos de disco óptico (ISO 9660 e *Universal Disk Format* - UDF), sistema de arquivos de memória *flash*, sistema de arquivos de fita magnética e sistema de arquivos de rede [12,23]. Os sistemas de arquivos de HD podem variar ainda entre os sistemas operacionais. Por exemplo, o sistema de arquivos de disco rígido para o MacOS é **Hierarchical File System** (HFS), para UNIX, o sistema **Unix File System** (*UFS*) e, para Linux, o sistema nativo é o **Ext**. Alguns sistemas operacionais suportam mais de um sistema de arquivos e provêem uma interface amigável comum para manipulá-los. Para usufruir as vantagens de um sistema de arquivos, é necessário que as mídias sejam **formatadas** previamente.

Nesta seção vamos detalhar o sistema de arquivos FAT que é o mais difundido entre os dispositivos de armazenamento de tecnologia FLASH NAND cujo uso tende-se a crescer entre os projetos de sistemas embarcados. A primeira versão de FAT foi proposta por Marc McDonald e Bill Gates em 1977. É denominado *8-bit*

File Allocation Table (FAT8), porque os elementos da tabela de alocação de arquivos são de 8 *bits*. Este sistema de arquivos tem recursos bem limitados. Suporta uma massa de armazenamento de até 8MB com os nomes de arquivos formados por, no máximo, 9 caracteres ASCII, 6 para o nome e 3 para a extensão do arquivo. Só existe um nível de diretório (diretório-raíz).

Com o aumento da capacidade de armazenamento dos HDDs, novas versões foram propostas para adequar à quantidade de *bits* necessários para endereçar os *clusters* e a organização de sub-diretórios. As versões sucessoras foram nomeadas conforme o tamanho de *bits* alocados aos elementos das tabelas de alocação: 12 *bits* (FAT12), 16 *bits* (FAT16), 32 *bits* (FAT32) e 64 *bits* (exFAT). Quando formatados os HD numa dessas versões de FAT, os discos são particionados em *clusters* de tamanhos iguais de setores e agrupados em até quatro regiões:

- setores reservados: como o setor de *boot* (códigos de máquina que são carregados na memória principal e executados durante a inicialização);
- região de FAT: contém as tabelas de locação de arquivos;
- região de diretório-raíz: contém as entradas do diretório-raíz do sistema; e
- região de dados: contém as informações dos diretórios e dos dados.

O setor de *boot* ocupa sempre o primeiro setor de um disco, cujo número lógico é 0. Além de conter códigos responsáveis pela inicialização do sistema computacional, ele contém uma série de informações sobre o próprio sistema de arquivos, como o tipo de mídia de armazenamento, o nome do volume, a quantidade de FAT etc [14]. A região de FATs contém todas as referências aos *clusters*. Em cada uma das suas entradas pode-se encontrar um dos seguintes dados, como mostra o sistema FAT32 na Figura 6.14: (1) a referência do próximo, (2) marca do fim de uma lista (EOF), (3) *cluster* não usado (00), e (4) *cluster* reservado. Observe que a entrada ao primeiro *cluster* de uma lista está num registro de um diretório.

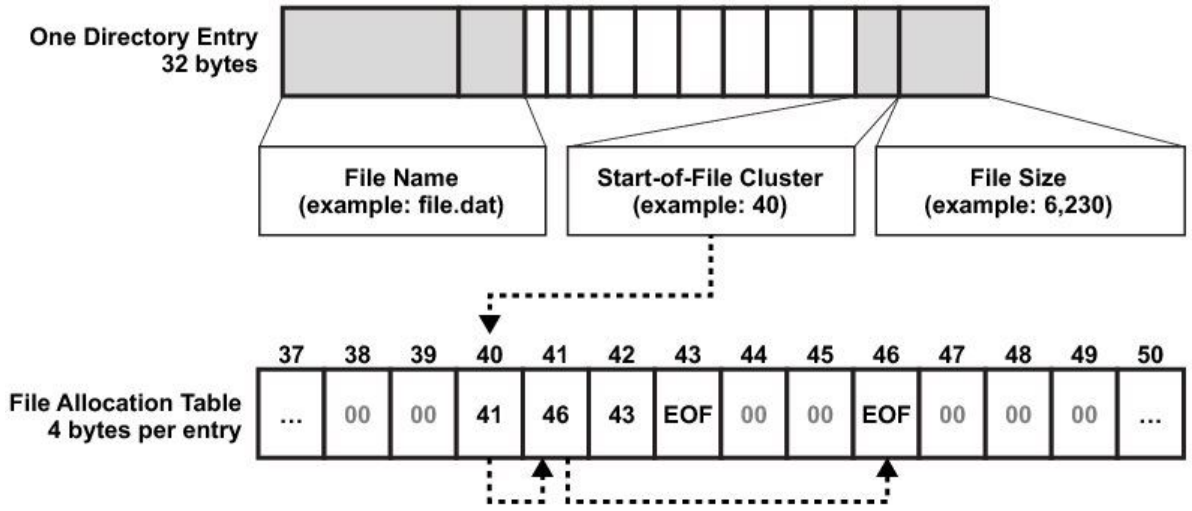


Figura 6.14: Elementos de um FAT 32 referenciados por um diretório.

No FAT12 e FAT16 as informações sobre os arquivos e os sub-diretórios dos primeiro nível de diretórios são armazenadas na região de diretório-raíz. Isso facilita a localização do diretório-raíz e seus sub-diretórios, mas limita o tamanho máximo de arquivos e sub-diretórios, pois a região de diretório-raíz é pré-alocada. Para contornar esta limitação, esta região de diretório-raíz foi deslocada para a região de dados no FAT32, como mostra a Figura 6.15. Note que, neste exemplo, a tabela FAT é replicada duas vezes. Há sistema em que a tabela não é replicada e há outros em que ela é replicada diversas vezes. As réplicas aumentam a confiabilidade do sistema. A quantidade de réplicas depende da susceptibilidade da mídia aos erros.

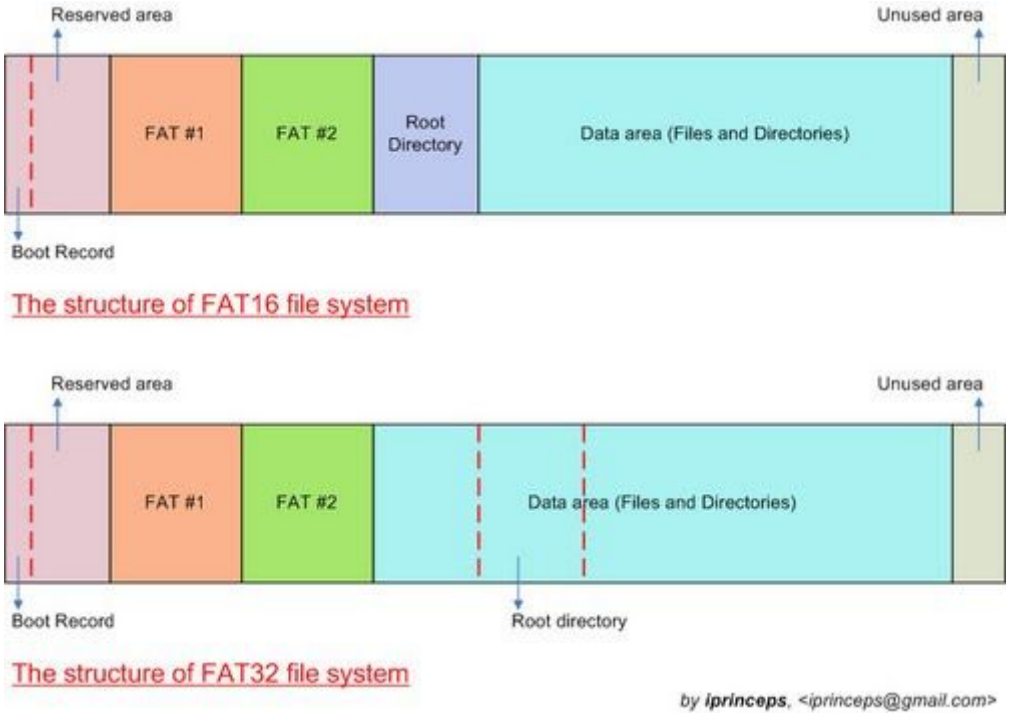


Figura 6.15: Agrupamentos de *clusters* no FAT12/FAT16 e FAT32 (Fonte: [13])

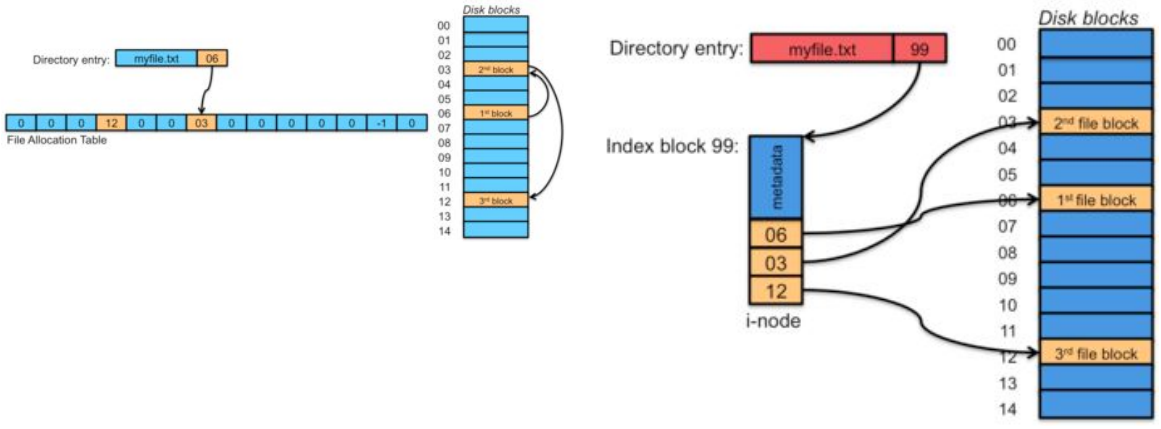
Apesar dos esforços em aprimorar a capacidade de organização do sistema FAT, o FAT 32 só consegue suportar um volume de dados de até 2TB com arquivos individuais não superiores a 4GB. O sistema exFAT foi introduzido pelo Microsoft em 2006 e otimizado para os dispositivos de armazenamento de massa FLASH NAND. Foi adotado pela Associação de Cartões SD, em inglês *SD Card Association*, como o formato de cartões padrão para cartões de capacidade maior que 32 GiB (*gibibytes* = 2^{30} *bytes*).

Mesmo assim, FAT não tem suporte (1) à manutenção de um relatório de alterações no sistema de arquivos antes de fazer qualquer modificação num HDD, (2) ao controle de permissões de acessos e (3) ao *link* simbólico (atribuir diferentes

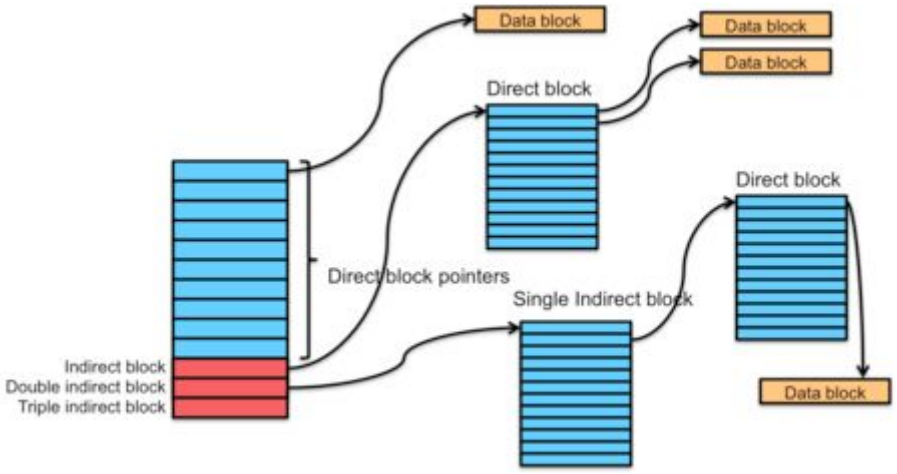
entradas a um mesmo arquivo físico). Daí o Microsoft decidiu introduzir o sistema de arquivos NTFS em 1993 com Windows NT 3.1 que dispõe de uma árvore de estruturação das referências dos *clusters* mais elaborada, conhecida como árvore B+ [15], de maneira que a busca por algum arquivo ou algum diretório fique mais eficiente.

Apesar das melhorias introduzidas ao sistema NTFS, os fabricantes dos dispositivos de armazenamento baseados na tecnologia FLASH NAND ainda adotam o sistema de arquivo FAT32, pois todos os três problemas mencionados, capacidade de armazenamento, *log* e segurança, não são ainda problemas para a maioria desses dispositivos de armazenamento.

Vale também comentar que, ao invés de percorrer a lista de *clusters* ligados (vários acessos ao dispositivo), para acessar um arquivo completo (Figura 6.16.(a)), podemos pensar em armazenar os índices de todos esses *clusters*, denominados **inodes**, num mesmo bloco de dados e fazer um único acesso (Figura 6.16.(b)). Porém, esta segunda alternativa apresenta o problema da variabilidade do tamanho da lista de inodes associados a cada arquivo. Uma terceira alternativa é criar uma lista de números fixos de inodes e de ponteiros para novas listas de inodes. Figura 6.16.(c) ilustra o sistema UFS proposto pelo Bell Labs, onde no primeiro nível de bloco temos 10 inodes, 1 ponteiro para uma lista de inodes, 1 ponteiro para uma lista de ponteiros de inodes e 1 ponteiro para uma lista das listas de inodes.



(a) FAT (b) inode



(c) inodes combinados.

Figura 6.16: Sistemas de arquivo baseados na FAT e em inodes (Fonte: [16]).

Para uniformizar acessos a esta diversidade de sistemas de arquivos, a *SUN Microsystems* introduziu em 1985 um nível de abstração sobre os sistemas de arquivos denominado **sistema de arquivos virtual**, em inglês *virtual file system* (VFS). Esta camada implementa uma série de operações sobre os dados num dispositivo físico deixando transparentes os detalhes de organização de cada um. Com isso, é possível usar o mesmo conjunto de funções do sistema operacional, conhecidas por **chamadas de sistema**, em inglês *system calls*, para acessar os dados armazenados em diferentes sistemas de arquivos, conforme ilustra a Figura 6.17..

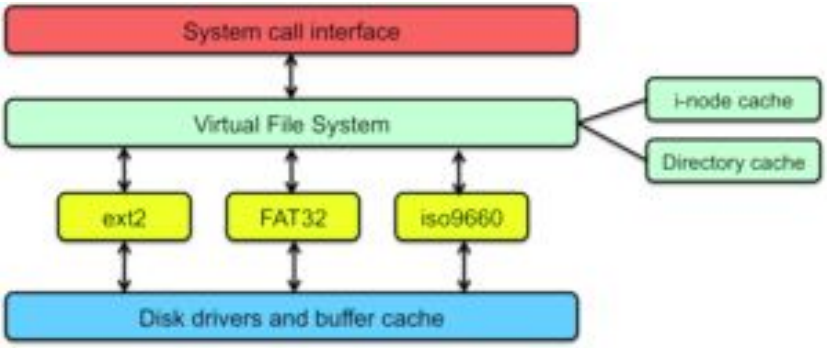


Figura 6.17: Sistema de arquivos virtual (Fonte: [16]).

6.5 Abstração em Programação de Alto Nível

Na seção 5.7 vimos que, em programação de alto nível C, podemos usar os diferentes tipos de dados para o compilador entender o tamanho do espaço de memória que gostaríamos que ele aloque para as variáveis definidas num programa. Neste capítulo vimos, no entanto, que isso não é suficiente para assegurar o desempenho do programa, pois entre o disco rígido onde está armazenado o programa e a CPU temos não só uma hierarquia de memórias de diferentes velocidades e de diferentes capacidades (Figura 6.1) como também uma série de algoritmos que procuram otimizar os acessos ao disco rígido e à memória principal aplicando o princípio de localidade. Nesta seção vamos mostrar que a linguagem C dispõe de alguns recursos para um projetista de sistemas embarcados ajudar o compilar e o ligador a organizar os dados que propicie a localidade.

Em relação aos acessos às memórias secundárias de massa, como os discos rígidos, vamos ver que a linguagem C disponibiliza um conjunto de funções de manipulação de arquivos básicas implementadas sobre o sistema de arquivos virtual (Seção 6.4) para ler e escrever os dados nas memórias secundárias.

6.5.1 Memória Principal

Nesta seção vamos apresentar alguns comandos e diretivas da linguagem C que podem ter efeitos na alocação dos espaços em memória principal para os dados de um programa.

6.5.1.1 Qualificadores

Vimos na Seção 5.7 que os **qualificadores** são palavras-chave que modificam as propriedades dos tipos de dados básicos. Já apresentamos os qualificadores relacionados aos tamanhos dos dados: signed, unsigned, short e long. Existem ainda duas classes de qualificadores:

1. Em relação à variabilidade dos dados: **const** e **volatile**.
2. Em relação à localização dos dados: **auto**, **register**, **static** e **extern**.

O qualificador **const** é utilizado para estabelecer que uma variável, que pode ser de qualquer tipo, não terá seu valor alterado durante a execução do programa após a atribuição. Vale comentar aqui que há outras formas para declarar um (valor) **constante** em C, além de “const int valor=5”, como:

1. #define valor 5
2. enum {valor=5}

Qual é, então, a melhor forma para declarar um constante em sistemas embarcados? Vai depender da aplicação. O importante é ter em mente qual efeito de cada uma das declarações. Para a primeira declaração o valor constante é

usualmente armazenado numa memória ROM; portanto, é considerado de fato um valor constante. A segunda declaração é uma simples diretiva que instrui o pré-processador a substituir todas palavras “valor” no código por “5” antes da compilação; portanto, é também conhecido como declaração de um literal. E a terceira declaração, utilizando o tipo de dado enumeração **enum**, define “valor” como um identificador de uma variável a ser armazenada na memória RAM que tem um conjunto restrito de valores. No caso, só o valor 5.

O qualificador **volatile** é adicionado na definição de um tipo de dado para indicar que o valor pode ser modificado sem uma instrução explícita do seu programa a qualquer momento, como uma variável que é atualizada pelo sistema de *clock* ou por fontes externas ao programa em execução.

Os qualificadores **const** e **volatile** podem ser utilizados juntos, para garantir que uma variável não tenha o seu valor alterado dentro do programa, mas que o valor do endereço por ele indicado pode ser modificado externamente, como um caractere que entra no sistema computacional através de um teclado.

Toda variável é ainda qualificada pela sua forma de armazenamento: **auto**, **extern**, **static** e **register** [17]. Por padrão, todas as variáveis declaradas dentro do escopo de uma função são qualificadas como **auto**. Elas são também conhecidas como **variáveis locais** com o seu tempo de vida limitado ao tempo em que a função é executada. Quando as variáveis são definidas fora do escopo das funções, mas dentro de um arquivo de programa, elas são chamadas **variáveis globais**, cujo acesso é compartilhável entre todas as funções de um mesmo arquivo. Podemos usar o qualificador **extern** para referenciar uma variável global declarada num arquivo de programa diferente.

O qualificador **register** tem como função declarar variáveis que devem ser armazenadas em um registrador ao invés na memória principal, para acessos rápidos. Entretanto, essa declaração não necessariamente significa que o valor da variável será armazenado em um registrador. Isso depende dos recursos em *hardware* e das restrições de implementação. Por isso, ele não é muito utilizado, a menos em casos de otimização para aplicações bem específicas.

Já o qualificador **static** define que uma variável persiste na memória durante o tempo de vida do programa, não tendo que ser criada toda a vez que é chamada pelo programa/função, ou descartada toda a vez que termina o ciclo de execução do programa/função.

6.5.1.2 Alocação de Memória

O processo de alocação de memória consiste em reservar parte do espaço de memória

para a execução de um programa ou um processo. A linguagem C permite alocar os espaços da memória principal nas seguintes formas: estatica, automatica e dinamicamente. O espaço de memória alocado para cada variável depende da quantidade de *bytes* que ele precisa para representar os dados.

Variáveis alocadas de forma **estática** no código do programa persistem na memória por todo o tempo de execução do programa, independentemente de serem necessárias ou não. Quando se declara uma variável de certo tipo de dado, vimos que o compilador aloca automaticamente a quantidade de *bytes* necessários para representar os seus valores.

Já variáveis **automáticas** são alocadas em uma **pilha** numa chamada de rotinas. Elas são empilhadas antes de entrar na rotina e desempilhadas antes de retornar ao ponto de entrada. Os valores dessas variáveis não são preservados para outras chamadas de mesmas funções. Para esse tipo de variável, um tamanho de memória é alocado durante o tempo de execução conforme o seu tipo de dados declarado.

Para os casos em que o tamanho de memória necessário não é conhecido a priori, tanto a alocação estática quanto a alocação automática de memória não são adequadas. A alocação **dinâmica** de memória, em que a memória é manipulada de forma explícita dentro de C, seria a solução. Numa alocação dinâmica, a memória pode ser alocada e liberada conforme o seu uso durante a execução do programa. Isso permite que o programa tenha mais espaço de memória. Tabela 1 apresenta as funções básicas de manipulação do espaço da memória em C. Para usá-las, basta incluir no código o arquivo <stdlib.h> que contém o protótipo destas funções.

Tabela 6.1: Descrição das funções em linguagem C referentes à alocação de memória

Função	Finalidade
calloc - void *calloc(int N, int size);	Aloca um bloco de *N* elementos, com cada elemento com tamanho em bytes conforme *size.*
malloc - void *malloc(int N);	Aloca um bloco de *N* bytes.
free - void free(void * ptr);	Libera um bloco da memória previamente alocado de endereço especificado pelo ponteiro *ptr*
realloc - void *realloc(void * ptr, int newsize);	Realoca um espaço de memória dado por *ptr* estendendo-o para *newsize* em bytes.

Observe que o tipo de dado retornado por estas funções é *void*. Que tipo de dado é este? Em inglês ***void*** significa vazio. Ele só é utilizado em três situações bem

específicas:

1. Quando uma rotina ou uma função não retorna nenhum valor, ela é declarada como do tipo *void*. Por exemplo, a função *free* que libera um bloco de memória endereçado pelo "ptr", *void free (void *ptr)*;

2. Quando a lista de argumentos de uma função ou subrotina é vazia. Por exemplo, a função *clock* que retorna o tempo consumido por um processo, *clock_t clock (void)*;

3. Quando se refere a um endereço da memória, sem especificar o tipo de dado que será armazenado naquela posição de memória, como mostrado na Tabela 6.1.

Uma vez esclarecido que o tipo de dado *void* não associa nenhum tipo de dado ao endereço do espaço alocado, é necessário fazer um *casting* no endereço retornado (Seção 5.7), convertendo-o do tipo *void* para o tipo desejado. Vale observar que o tamanho do espaço de memória a ser alocado é especificado em *bytes*, a menor unidade de armazenamento considerada pela CPU. A função **sizeof** é uma função em C que nos ajuda a descobrir o tamanho em *bytes* de uma variável de um tipo de dado específico.

Um erro comum é a alocação dinâmica de um espaço de memória para uma string de *n* caracteres, como "hello, world!" de 13 caracteres. Ao invés de n bytes, devemos alocar *n*+1 *bytes*, pois o terminador de uma string é '\0'. Vale ainda frisar que o recurso de memória não é infinito, por isso é importante adquirir o bom hábito de liberar a memória quando não é mais utilizado, com uso da função **free** que desaloca a memória.

6.5.1.3 Endereços da Memória

Vimos na Seção 6.5.1.2 que em C, quando chamamos uma função de alocação dinâmica de um espaço de memória, recebemos como retorno um endereço da memória principal. Este endereço é conhecido em C como um **ponteiro**. Por padrão, as funções de alocação retornam o endereço como uma variável do tipo de dado void. Porém, quando fizermos um *casting* com um outro tipo de dado, como **int**, o endereço continua o mesmo, mas o seu conteúdo será processado como do tipo **int**. Para diferenciar uma variável do <tipo de dado> do endereço desta variável na memória principal, utilizamos o operador "*" no formato:

*<tipo de dado *> <nome_do_ponteiro>;*

que significa que a variável <*nome_do_ponteiro*> é um ponteiro, ou o seu valor é o endereço de um valor do <*tipo de dado*>.

Em C existem operadores especiais que permitem acessar o endereço de um valor e o conteúdo de um endereço. Em uma operação, diferentemente da declaração, o compilador traduz o operador **"*"** aplicado sobre uma variável ponteiro por "o conteúdo da variável ponteiro" e o operador "**&**" aplicado sobre uma variável qualquer por "o endereço da variável". O endereço de pNumber, &pNumber, na Figura 6.18, não é conhecido, mas o seu conteúdo pNumber é um endereço a um valor do tipo inteiro number, pois foi declarado como "int * pNumber;" e foi feita a atribuição pNumber = &number. O conteúdo do number é 80.

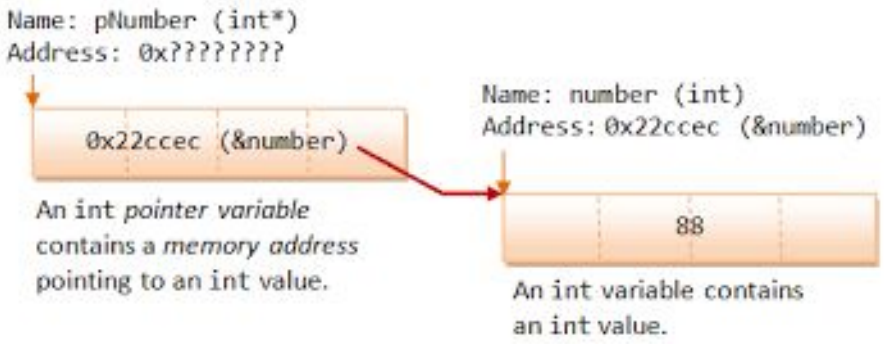


Figura 6.18: Ponteiros (Fonte: [19])

Cabe reforçar aqui que, para um ponteiro, o compilador só aloca um espaço correspondente a um endereço. Portanto,

- antes de usá-lo deve-se garantir que o seu conteúdo seja um endereço válido. Na prática, recomenda-se inicializar uma variável ponteiro sempre com NULL, que é uma constante de valor 0 definida na biblioteca <stdlib.h>. Essa atribuição indica que o ponteiro não aponta para um endereço da memória específico;
- o tamanho do valor de um ponteiro é sempre igual à quantidade total de *bits* necessários para endereçar o espaço da memória principal do sistema computacional, não importa o tamanho do valor armazenado no endereço.

6.5.2 Sistema de Arquivos

Existem uma grande variedade de funções para tratamento de transferência de dados entre a memória principal e a memória de massa, mais especificamente o disco rígido, na linguagem C. A maioria destas funções estão disponíveis numa biblioteca-padrão da linguagem C denominada **stdio.h**. As básicas são [20]:

- criação de um novo arquivo (*fopen* com os atributos do novo arquivo)
- abertura de um arquivo já existente (*fopen*)
- leitura de um arquivo (*fscanf* ou *fgetc*)

- escrita num arquivo (*fprintf* ou *fputc*)
- mover o “cabeçote do disco” para uma posição específica do arquivo (*fseek*, *rewind*)
- fechar o arquivo (*fclose*)

6.6 Exercícios

1. Qual é a relação entre os endereços processáveis pela CPU e os endereços da memória *cache* e entre os endereços da CPU e os da memória virtual?
2. A capacidade de armazenamento da memória *cache* deve ser somada à capacidade de armazenamento da memória principal para o cômputo do espaço de memória endereçável pela CPU num sistema de memória que não seja virtual? Justifique.
3. Quais são as três técnicas de mapeamento entre os endereços da memória principal e os da memória *cache*?
4. Por quê a técnica de mapeamento direto pode apresentar um desempenho pior do que o mapeamento de memória associativa em algumas circunstâncias? [22]
5. Considere uma CPU com uma memória cache de 1kB dividida em blocos de 64 *bytes* de mapeamento direto. É executado o seguinte programa na CPU

   ```
   short int dado[8][8];
   int soma = 0;
   for (int i=0; i<32; i++)
   for (int j=0; j<32; j++)
       soma += dado[j][i];
   ```

   Assumindo que o tamanho do tipo de dado “short int” é 2 *bytes,* que a matriz “dado” é armazenada como um vetor de 32 linhas, cada uma contendo 32 valores do tipo “short int”, que o endereço inicial do vetor “dado” coincida com o ínicio de um bloco da memória cache, que as variáveis i, j estão armazenadas nos registradores. Qual é a taxa de *hit* e de *miss* na execução do programa?
6. Considere a estimativa de taxa de hit e o tempo de acesso efetivo de um programa executado num sistema computacional com técnica de mapeamento direto na memória cache com 4 *slots* (blocos, linhas) de 16 words (32 *bits*). A organização da memória cache e da memória principal é mostrada na Figura 6.19 [8]

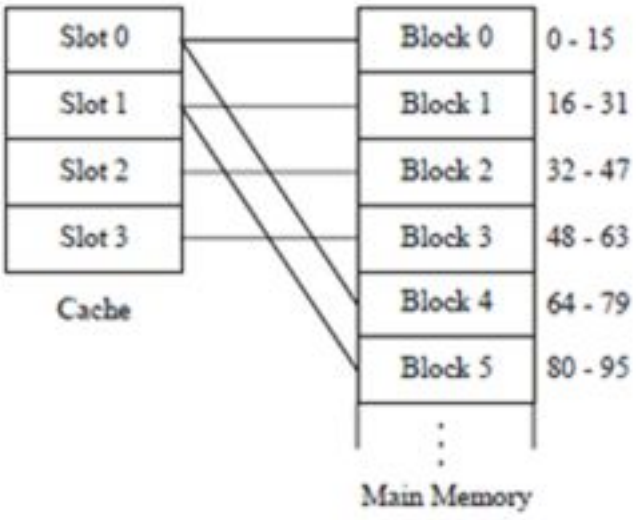


Figure 7-17 An example of a direct mapped cache memory.

Event	Location	Time	Comment
1 miss	48	2500ns	Memory block 3 to cache slot 3
15 hits	49-63	80ns×15=1200ns	
1 miss	64	2500ns	Memory block 4 to cache slot 0
15 hits	65-79	80ns×15=1200ns	
1 miss	80	2500ns	Memory block 5 to cache slot 1
15 hits	81-95	80ns×15=1200ns	
1 miss	15	2500ns	Memory block 0 to cache slot 0
1 miss	16	2500ns	Memory block 1 to cache slot 1
15 hits	17-31	80ns×15=1200ns	
9 hits	15	80ns×9=720ns	Last nine iterations of loop
144 hits	16-31	80ns×144=12,240ns	Last nine iterations of loop
Total hits = 213		Total misses = 5	

Figure 7-18 A table of events for a program executing on an architecture with a small direct mapped cache memory.

Figura 6.19: Memória cache mapeada diretamente (Fonte: [8]).

O tempo de acesso à memória *cache* é 80ns e o tempo de transferência da memória principal para a memória *cache* é 2500ns. Considere ainda que o processo de carga a uma memória *cache* seja ***load-through***, ou seja, a palavra que gerou *cache miss* é transferida diretamente da memória principal para os registradores da CPU e o resto dos seus dados vizinhos para um *slot* da memória *cache*. Calcule a taxa de *hit,* a taxa de *miss*, tempo de acesso efetivo e a taxa de aceleração.

7. Por quê o projeto de uma memória *cache* L1 para as instruções é mais simples do que o projeto de uma memória *cache* L1 para os dados?
8. Qual é a taxa de aceleração S máxima teórica para uma memória cache com os seguintes parâmetros: t_m = 100ns, t_c = 20ns, e H = 0.95. A taxa de

aceleração máxima teórica dificilmente será alcançada. Suponha num sistema real com um processador da família 6800 da Motorola com os seguintes parâmetros: frequência do clock = 16Mhz, estado de espera pela memória principal = 4 clocks e estado de espera pela memória cache = 0 clocks. Qual deve ser a taxa de hit deste sistema para que a taxa de aceleração seja 1.4? [22]

9. Quando a CPU escreve na memória *cache*, ambas as memórias, *cache* e principal, devem ser atualizadas. Se o dado não estiver na memória *cache*, ele deve ser transferido da memória principal para a memória *cache* para ser atualizado. Seja t_1 o tempo necessário para carregar na memória *cache* o dado em falta. Mostre que o tempo de acesso efetivo do sistema de memória com *cache* é [22]

$$t_{ave} = Ht_c + (1-H)t_m + (1-H)t_1$$

10. Seja uma memória cache de 512 bytes com linhas/blocos de 64 bytes. O endereço da CPU é de 16 bits. Um programa acessa os seguintes endereços da memória principal: 13, 136, 490, 541, 670, 74, 581, 980. Considere que o estado inicial da memória cache é inválido.
    a. Quais são os valores de "Tag", "Slot"/"Set" e "Word" desses endereços no caso de: mapeamento direto, memória associativa completa, e memória associativa por 2 conjuntos?
    b. Qual é o conteúdo da memória cache os três casos de técnica de mapeamento?
11. O que você entende por sistema de arquivos?
12. Por quê os sistemas de arquivos FAT, mesmo considerados muito limitados, são usados entre as modernas memórias FLASH NAND?
13. Quais seriam os argumentos da função malloc em C para alocar um espaço de memória para os seguintes dados:
    a. uma *string*: "EA075",
    b. um vetor de 50 elementos do tipo signed int,
    c. uma matriz bidimensional de 20x10 elementos do tipo unsigned .

    Quantos *bytes* são alocados em cada caso? Como o endereço do espaço alocado é retornado?
14. Considere a sequência das duas instruções de um código:
    char *dado;
    dado = 'A';
    O que acontece quando se executa estas duas instruções?

6.7 Referências

[1] https://www.doc.ic.ac.uk/~eedwards/compsys/memory/index.html

[2] Joel Hruska. How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips. https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips

[3] Wikipedia. Cache performance measurement and metric. https://en.wikipedia.org/wiki/Cache_performance_measurement_and_metric

[4] Margaret Rouse. Cache memory. https://searchstorage.techtarget.com/definition/cache-memory

[5] Karthik Amr. Programming: How to improve application performance by undestanding the CPU Cache levels. https://hackernoon.com/programming-how-to-improve-application-performance-by-understanding-the-cpu-cache-levels-df0e87b70c90

[6] Tanenbaum, A. S. Sistemas Operacionais Modernos. ISBN: 978-8543005676. Pearson Universidade. 2015

[7] 8051 Microcontrollers.blogpost.com. Memory: cache Memory (associative mapped cache and direct mapped cache). http://8051-microcontrollers.blogspot.com/2015/01/memory-cache-memory-associative-mapped.html#.XV9O2vxG3CJ

[8] 8051 Microcontrollers.blogpost.com. Memory: cache memory (set associative mapped cache, hit ratios, and effective access times, multilevel caches and cache management). http://8051-microcontrollers.blogspot.com/2015/01/memory-cache-memory-set-associative.html#.XV_7X_xG3CK

[9] https://tex.stackexchange.com/questions/210826/drawing-a-single-line-of-cache-memory

[10] https://slideplayer.com/slide/5063013/

[11] https://slideplayer.com/slide/7498397/

[12] Wikipedia. File Systems. https://en.wikipedia.org/wiki/File_system#Disk_file_systems

[13] My Hobbies. Understanding file system. https://sites.google.com/site/iprinceps/Home/embedded-system-and-operating-systems/understanding-file-system

[14] Phobos. A tutorial on the FAT file system. http://www.tavi.co.uk/phobos/fat.html

[15] GeeksForGeeks. Introduction of B+ Tree. https://www.geeksforgeeks.org/introduction-of-b-tree/

[16] Paul Krzyzanowski. File systems.
https://www.cs.rutgers.edu/~pxk/416/notes/12-filesystems.html
[17] Codes Dope. Storage Class in C.
https://www.codesdope.com/c-storage-classes/
[18] cppreference.com. Enumerations.
https://en.cppreference.com/w/c/language/enum
[19] - C++ Programming Language: Pointers, References and Dynamic Memory Allocation.
https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_pointerreference.html
[20] GeeksForGeeks. Basics of File Handling in C.
https://www.geeksforgeeks.org/basics-file-handling-c/
[21] Cache memory organization.
https://edux.pjwstk.edu.pl/mat/264/lec/main84.html
[22] Alan Clements. Microprocessor Systems Design: 68000 Hardware, Software, and Interfacing. ISBN 0-534-94822-7
[23] Recovery Explorer. File systems of different devices and operating systems.
https://www.r-explorer.com/blog/general-issues/file-systems.php
[24]
https://www.researchgate.net/figure/A-typical-mass-market-multicore-design-with-shared-third-level-L3-cache-Levels-1-L1_fig1_280154268