

階層記憶

demand: High speed&large space Solution: Memory Hierarchy

内存访问的局部性原理

Time and Space

記憶階層と命令セット

两种方式:

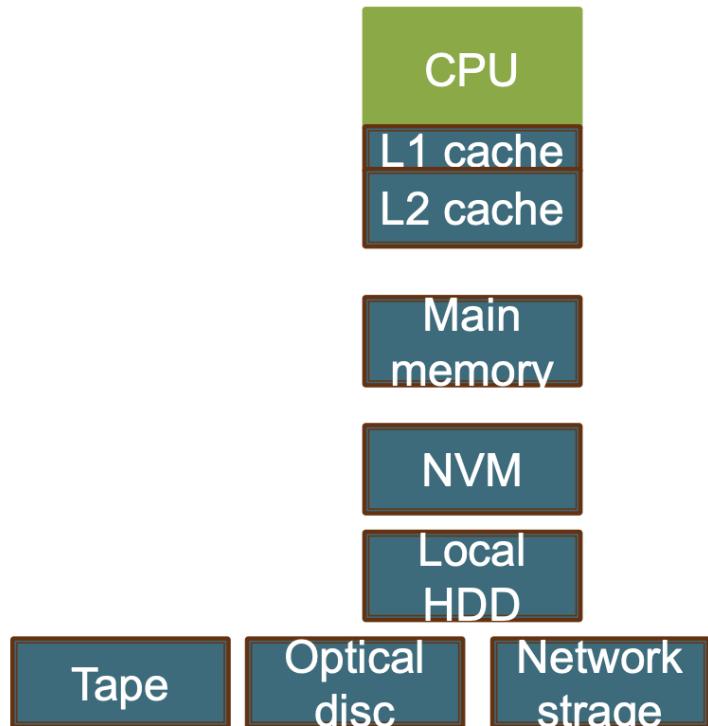
- 记忆阶层を陽に見せる方式: 将结构暴露给程序员, 要求程序(软件)来管理存储器层次结构, 潜力高。
- 記憶階層を見せない方式: 程序只需将存储器视为一个单一的、高速、大容量、统一的内存, 通过硬件自动管理存储器层次结构, 通用。

記憶階層 design

■ 記憶階層の設計

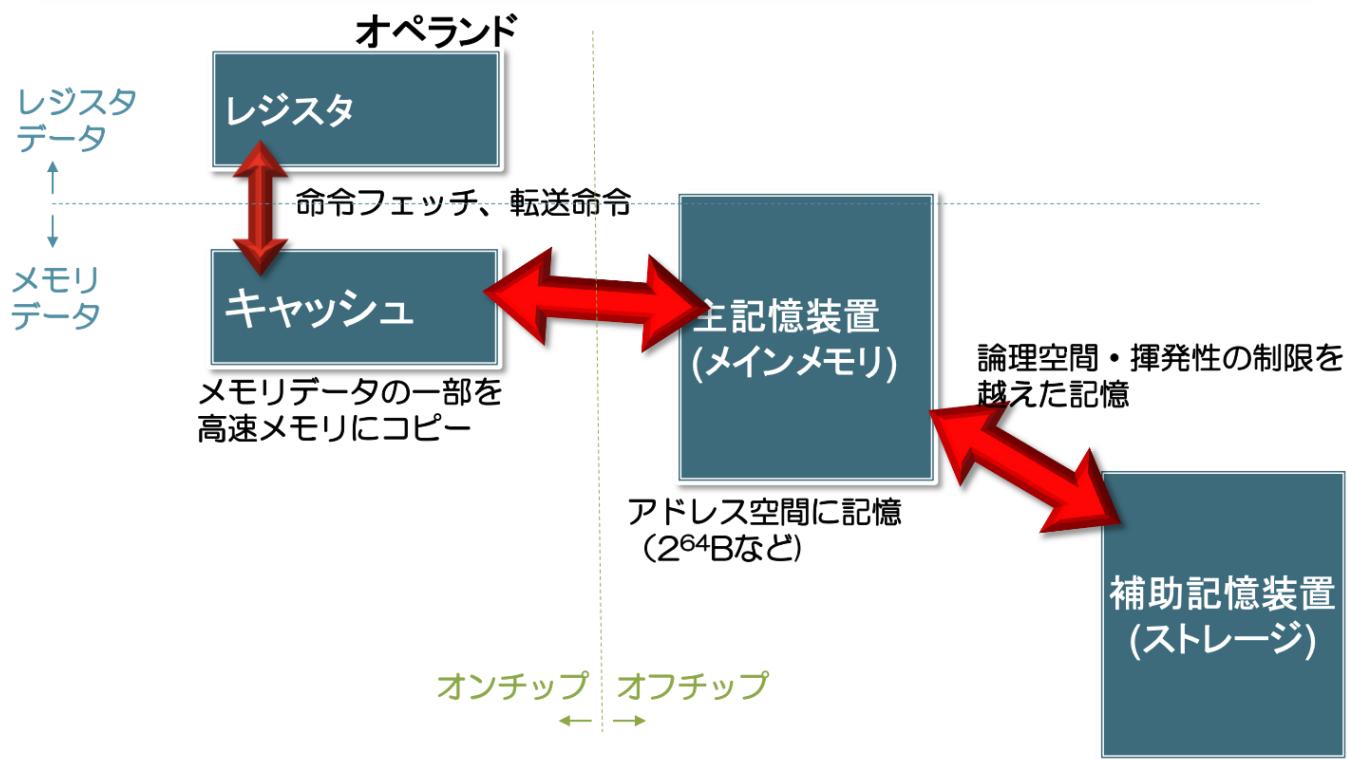
• CPUに近い方から

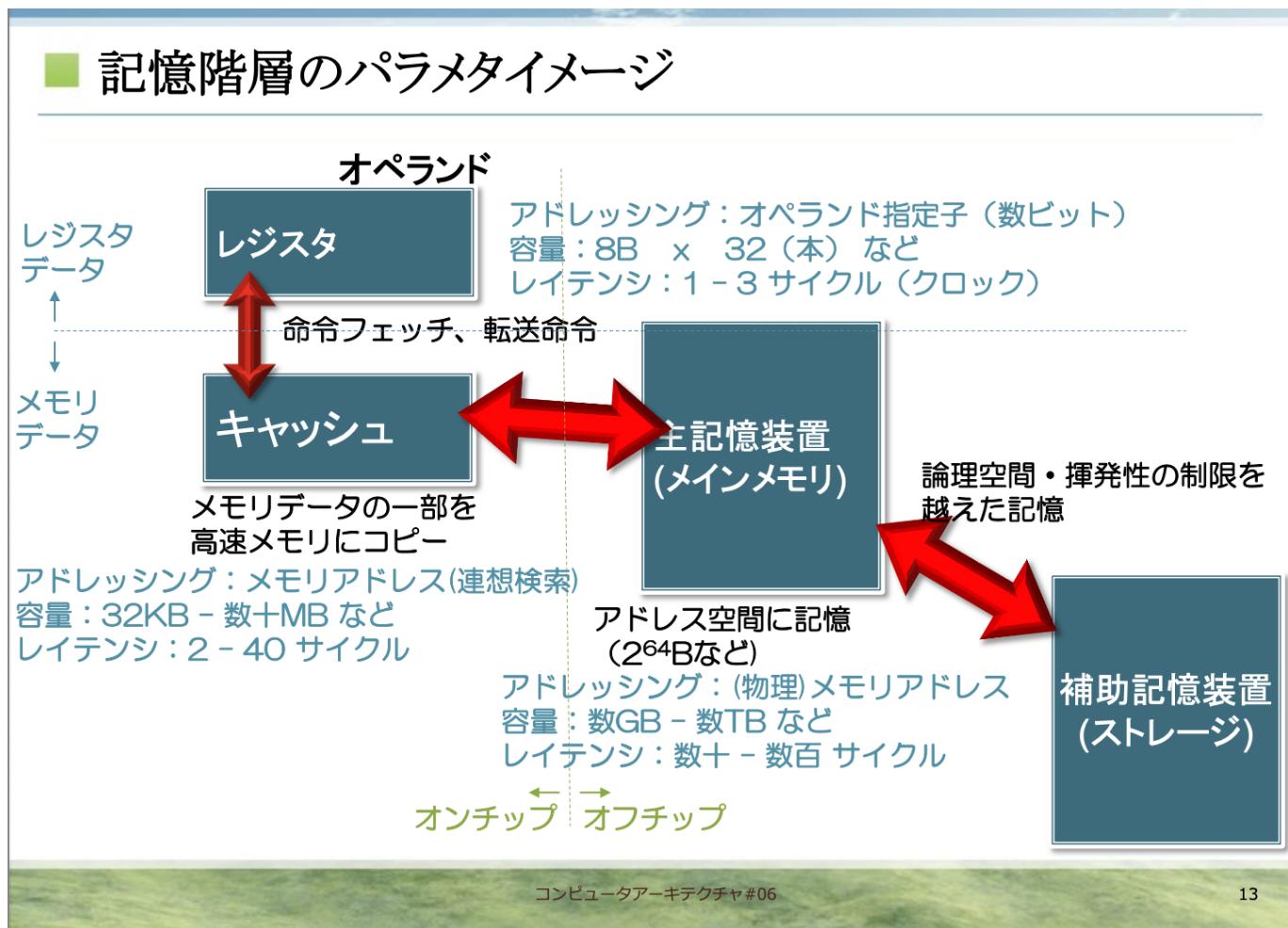
- 高速→低速
- 小容量→大容量
- 一時的→永続的
- (固定→柔軟)



例子:

■ 記憶階層の例





以下是对这张图的详细解读和总结，让我们可以对速度、容量和延迟有更具体的概念：

■ 記憶階層のパラメタイマー (存储器层次结构的参数估算) 解读

这张图列出了寄存器、高速缓存、主内存和辅助存储这四个层级的主要性能指标（地址、容量和延迟）。

1. 寄存器 (レジスタ)

参数	值 (估算)	解释
寻址 (アドレッシング)	オペランド指定子 (数ビット)	通过指令中的几位操作数指定符直接寻址，速度极快。
容量 (容量)	\$8\text{B} \times 32\$ (本) など	容量极小，例如 32 个 8 字节 (64位) 寄存器。
延迟 (レイテンシ)	1 - 3 サイクル (クロック)	访问时间极短，通常只需要 1 到 3 个 CPU 时钟周期。

2. 高速缓存 (キャッシュ)

参数	值 (估算)	解释
寻址 (アドレッシング)	メモリアドレス (連想検索)	使用内存地址，但通过关联查找来确定数据是否在 Cache 中。

参数	值 (估算)	解释
容量 (容量)	\$32\text{KB} - \text{数MB}\$ など	容量较小, 通常从几十 KB (L1) 到数 MB (L2/L3) 不等。
延迟 (レイテンシ)	2 - 40 サイクル	访问速度很快, 大约在 2 到 40 个时钟周期内。

3. 主记忆装置 (メインメモリ/RAM)

参数	值 (估算)	解释
寻址 (アドレッシング)	物理メモリアドレス	使用物理内存地址, 由 CPU 发出。
容量 (容量)	数 GB - TB など	容量较大, 通常在 GB 级别到 TB 级别。
延迟 (レイテンシ)	数十 - 数百 サイクル	访问时间明显变长, 通常需要数十到数百个时钟周期。

4. 辅助记忆装置 (ストレージ/SSD/HDD)

- 特性: 论理空间・揮发性の制限を越えた記憶 (超越逻辑空间和挥发性的限制)。
- 参数缺失: 这部分参数 (如延迟) 通常以毫秒 (ms) 为单位, 而不是时钟周期, 因为它涉及机械/I/O 操作, 其延迟远高于 CPU 周期, 因此图中未直接给出周期数。

性能对比总结 (延迟)

通过比较延迟 (レイテンシ) 的数量级, 可以清晰地看出存储器层次结构的速度差异:

存储层级	延迟 (时钟周期)	相对速度
寄存器	1 - 3	极快 (CPU 核心速度)
Cache	2 - 40	快
主内存	数十 - 数百	慢
辅助存储	远高于数百 (通常以毫秒计)	极慢

这个巨大的速度差异 (从几个周期到数百周期甚至毫秒) 正是局部性原理和Cache 机制必须存在的原因, 以保证程序在大部分时间都只访问最快的那几级存储器。

Cache

- Cache 定义与核心作用: 自动、高效地利用存储器层次结构的技术, 让上层存储器持有下层存储器的一个子集
- Cache 的控制方式 (暗默的制御): ハードウェアで実現 (通过硬件实现) + ミドルウェア (OS) で実現 (通过中间件/操作系统实现)

3. Cache 命中率 (キャッシュヒット率): Cache 的性能主要由“命中率”决定
 - シンプルなアルゴリズムでも意外と当たる → 性能改善 (即使是简单的算法，命中率也出奇地高 → 性能改善)
 - 空間的局所性・時間的局所性 (空间局部性・时间局部性)
 - LRU 置き換えアルゴリズム (LRU 替换算法)

Cache Hierarchy

1. 多级 Cache 的概念 现代 CPU 采用多级 Cache 结构。从离处理器 (CPU 核心) 最近的开始，依次命名为一级缓存 (1 次キャッシュ, L1)、二级缓存 (2 次キャッシュ, L2)、三级缓存 (3 次キャッシュ, L3)，依此类推，其容量逐渐增大，但速度逐渐变慢。
2. L1 Cache 的分割 在高性能处理器设计中，一级缓存 (L1 Cache) 通常被分割成两部分：指令缓存和数据缓存。I1, D1 などと略称 (缩写为 I1, D1 等)，用于存放 CPU 将要执行的指令代码和用于存放程序运行时使用的数据。这种分离 (哈佛结构) 可以允许 CPU 在同一时钟周期内同时取出指令和访问数据，极大地提高了流水线的吞吐率。
3. 这是一个常见的现代多核 CPU Cache 结构：
 - プロセッサー (CPU 核心): 直接与 L1 交互。
 - I1, D1: 分离的一级指令和数据缓存。
 - L2 (コアにプライベート) : 二级缓存，通常是每个 CPU 核心独有 (Private)。
 - L3 (チップ上のコアが共有) : 三级缓存，通常是芯片上所有 CPU 核心共同使用 (Shared)。
 - メモリチップ (内存芯片): 即主内存 (RAM)。

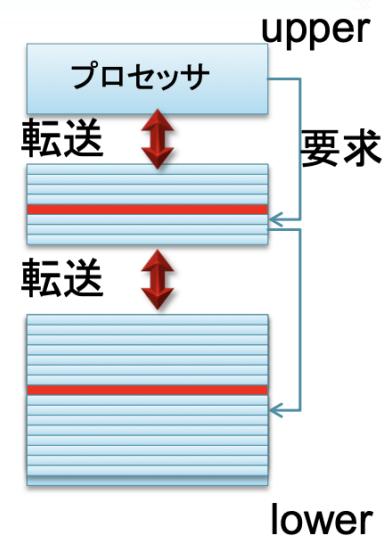
階層記憶のモデル

1. Consist of several levels
2. 層間の最小転送単位: Block/Line e.g. 64B

3. Data request: Hit/Miss

■ 階層記憶のモデル

- 数レベルで構成
- 階層間の最小転送単位
 - ブロック または ライン
 - 例 64B
- データリクエスト時
 - 上位階層から検索
 - 見つかれば「ヒット」
 - なければ「ミス」 ⇒ さらに下位を検索



キャッシュの場所 (Cache 的位置)

- (在过去, 通常是处理器芯片外部的 SRAM 内存)
- 現在は一般にキャッシュはオンチップ、メモリはオフチップ (现在一般 Cache 在片上, 内存 (主存) 在片外)
- (也有将大型 LLC 放在片外, 或将内存也集成到芯片内, 或放在芯片封装内等各种实现方式)
- オフチップにするとレイテンシは大きく増加 (如果放在片外, 延迟会大幅增加)
- 核心原则: Cache 越靠近 CPU 核心 (即越在片上), 其访问延迟就越小, 性能就越好。

キャッシュアクセスの基本

アクセスに要する時間 (访问所需时间)

1. 命中时 (ヒット時 - Hit)
 - ヒットミス判定の時間 (判断命中/不命中所需的时间)
 - その階層から読み出す時間 (高速) (从该层级读取数据所需的时间 (高速))
2. 不命中时 (ミス時 - Miss)
 - ヒットミス判定の時間 (判断命中/不命中所需的时间)
 - 下位階層から読み出す時間 (低速) (从下层存储器读取数据所需的时间 (低速))
 - 下位階層でさらにミスすることもある (在下层存储器也可能再次不命中)

- その階層から読み出す時間 (从该层级读取数据所需的时间)
3. 命中与不命中时的时钟周期差异
- ヒット時とミス時でバブルの数に違い (命中时和不命中时, Bubble/Stall 的数量不同)
 - 正因如此, CPU 的设计不必以最坏情况下的延迟为基准运行。只要通过利用局部性, 保证极高的命中率, 就可以让系统的平均访问时间接近于高速缓存的访问时间

連想検索の利用 (关联搜索的应用-Associative Search)

1. 核心問題 キャッシュ内容はメインメモリのサブセット (Cache 内容是主内存的子集)
2. あるアドレスへのメモリリクエスト (对某个地址的内存请求) we need to answer:
 - キャッシュ内に該当データはあるのか? ないのか? (Cache 中是否存在对应数据?)
 - あるとしてキャッシュの何処にしまってあるのか? (如果存在, 它被存放在 Cache 的哪个位置?)
 - 如果不能高速地决定这两个问题, 即使 Cache 本身是一个小容量、高速的设备, 也会因为查找时间过长而失去性能优势。
2. 关联搜索的定义与作用
- 定义: 关联搜索是一种机制, 它使得系统仅凭内存地址 (メモリアドレス) 就能快速判断对应数据是否存在于 Cache 中以及存在于哪个位置。
3. 实现关联搜索的常见机制
 - 一般的な知見: ハッシュを使う (一般的经验: 使用哈希/Hash)
 - 对于缓存来说, 为了避免冲突, 低位比特的信息更为重要: 简单地提取低位比特
 - 原因: 程序通常会连续访问相邻的内存地址。这些相邻地址的高位可能相同, 但低位是不同的。利用地址的低位作为 Cache 行号, 可以确保相邻的内存块分散到 Cache 的不同位置, 从而避免它们相互竞争同一个 Cache 行, 提高效率。

Cache Block/Cache Line

1. Cache 的存储单位 (キャッシュの格納の単位)
 - ブロック単位でキャッシュへ読み込み (以块为单位读取到 Cache 中)
 - ブロック単位でキャッシュから破棄 (以块为单位从 Cache 中丢弃)
 - キャッシュ 1 エントリに 1 キャッシュブロックを格納 (Cache 的一个条目Entry存储一个 Cache 块)
2. Cache 块的大小 (キャッシュブロックの大きさ) Cache 块的大小是一个关键的设计参数, 需要进行权衡
 - 一般に数ワード、32B、64B、128B など (一般来说是数个字、32 字节、64 字节、128 字节等)
 - ある程度の大きさがあると良い (需要有一定的大小)
 - 标签数组 (タグアレイ) 的容量可以减少: Cache 需要存储每个块对应的**标签 (Tag) **来标识它来自主内存的哪个地址。如果块很大, Cache 中块的总数就会减少, 从而减少存储标签所需的总容量, 节约 SRAM 空间。
 - 大きすぎると、同じ容量のキャッシュでもエントリ数が減ってしまい、競合增加 (如果太大, 即使 Cache 容量相同, 条目数也会减少, 导致竞争增加)
 - 条目数减少意味着更多的主内存地址会映射到相同的 Cache 条目上, 导致竞争 (Collision/Conflict) 增加, 从而降低命中率。

基本的なキャッシュの作り方 (Cache 的基本构建方法：映射方式)

Cache 映射方式是解决“主内存的一个块应该放在 Cache 哪里？”这个问题的关键机制。

1. 直接映射 (ダイレクト・マップ / Direct-Mapped): 主内存中的任何一个地址，在 Cache 中都只能存储在一个固定的、唯一的块位置上。
2. 组相联映射 (セット・アソシエイティブ / Set-Associative)-主内存中的一个地址可以存储在 Cache 的一个固定集合 (Set) 内的任意一个块位置。
- 例如 例如：2-way 组相联（在一个 Set 内有 2 个块位置可选），4-way 组相联（有 4 个块位置可选）。
3. 全相联映射 (フル・アソシエイティブ / Fully Associative): 主内存中的任何一个数据块可以存储在 Cache 中的任意一个空闲块位置。
- 实现最复杂、成本最高、速度最慢：由于需要同时查找所有块，硬件开销大且耗电。通常只用于容量非常小的 Cache 层级，如 TLB (Translation Lookaside Buffer) 或 L1 Cache 中非常小的部分。

ダイレクトマップ方式 (直接映射方式) 解读

1. 存储位置的固定性 (キャッシュの格納場所が一つ定まる):
 - メモリアドレスに対してキャッシュの格納場所が一つ定まる (对于一个内存地址，它在 Cache 中的存储位置是唯一固定的)
 - モジュロを取る (取模运算): 确定位置的方法就是对主内存块号进行取模运算。例如，如果 Cache 有 N 个块，内存块 M 只能映射到 Cache 块 $M \bmod N$ 。
2. 多个内存地址对应一个 Cache 条目 (複数のメモリアドレスが対応)
 - 一つのキャッシュエントリには複数のメモリアドレスが対応 (一个 Cache 条目对应多个内存地址)
 - 見分けるために、インデックスに使わなかった上位ビットをタグとしてブロック内容と一緒に記憶 (为了区分它们，将未用于索引的**上位比特 (高位) **作为 标签 (Tag) 与块内容一起存储)
 - 机制：当一个内存块被加载到 Cache 中时，除了块数据本身，Cache 还需要存储地址的高位部分，即 Tag。
 - 完整地址组成：内存地址被分割成三个部分：

$$\text{内存地址} = [\text{Tag}] + [\text{Index}] + [\text{Offset}]$$

- リクエスト時、タグが一致すればヒット、一致しなければミス (请求时，如果 Tag 一致则命中，不一致则不命中)

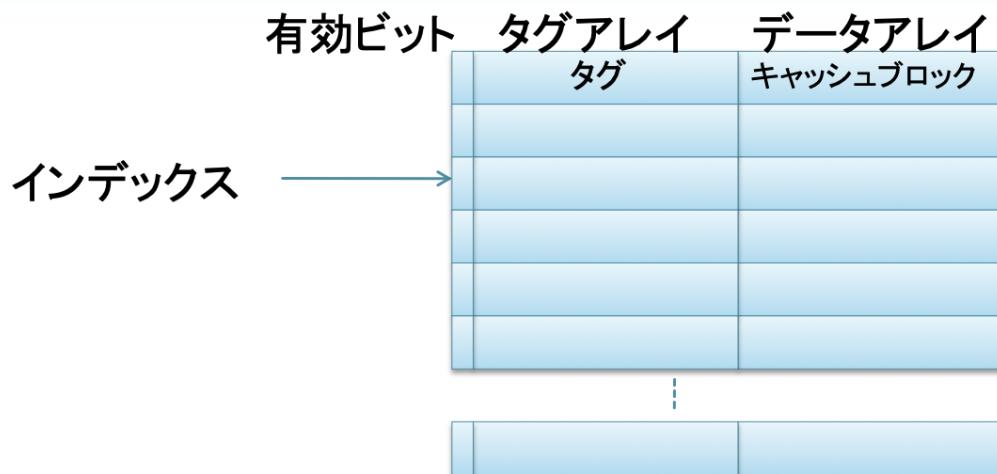
总结，查找流程：

1. CPU 提供地址
2. 使用地址中的 Index 部分找到 Cache 中的固定行 (取模运算等)
3. 如果 Tag 一致，则命中 (Hit)；如果 Tag 不一致，则不命中 (Miss) (称为冲突 Miss)。

直接映射的优点是实现简单，速度极快，但缺点是易发生冲突。

ダイレクトマップ・キャッシュ (直接映射 Cache) 结构图示解读

ダイレクトマップ・キャッシュ



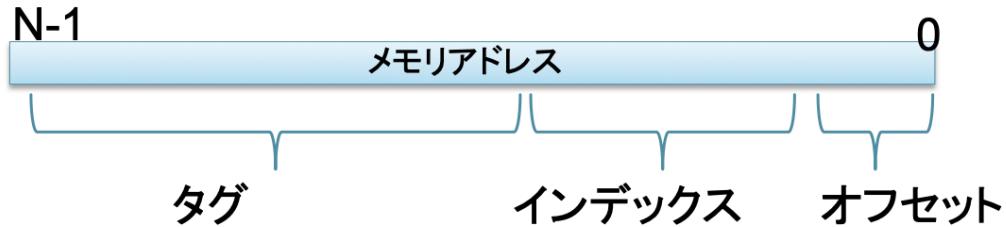
这张图展示了直接映射 Cache 的核心组成部分：索引、有效比特、标签数组和数据数组。

1. 索引 (インデックス / Index)
 2. Cache 内部的三个主要列
 - 有効ビット (有效比特 / Valid Bit)
 - タグアレイ (标签数组 / Tag Array)
 - データアレイ (数据数组 / Data Array)
 3. 查找流程总结 (直接映射)
 - Index: 根据 A 的中间比特位确定 Cache 的行号 i 。
 - Valid Bit: 检查第 i 行的有效比特是否为 1
 - Tag Match: 将 A 的高位 Tag 部分与第 i 行存储的 Tag 进行比较。
 - Hit / Miss

直接映射 (Direct-Mapped) 机制的精确数学描述

这张图展示了在直接映射 Cache 中，一个 N 位内存地址（从 N-1 位到 0 位）的逻辑分割。

■ アドレスからインデックスの決定 (ダイレクトマップ、ブロック長1ワード)



- 下位($\log_2(\text{ブロックサイズ (バイト数)})$)がオフセット
- 次の下位 ($\log_2(\text{エントリ数})$) ビットがインデックス
 - キャッシュの格納位置
- 残りの上位ビットがタグ

地址分割的数学原理：

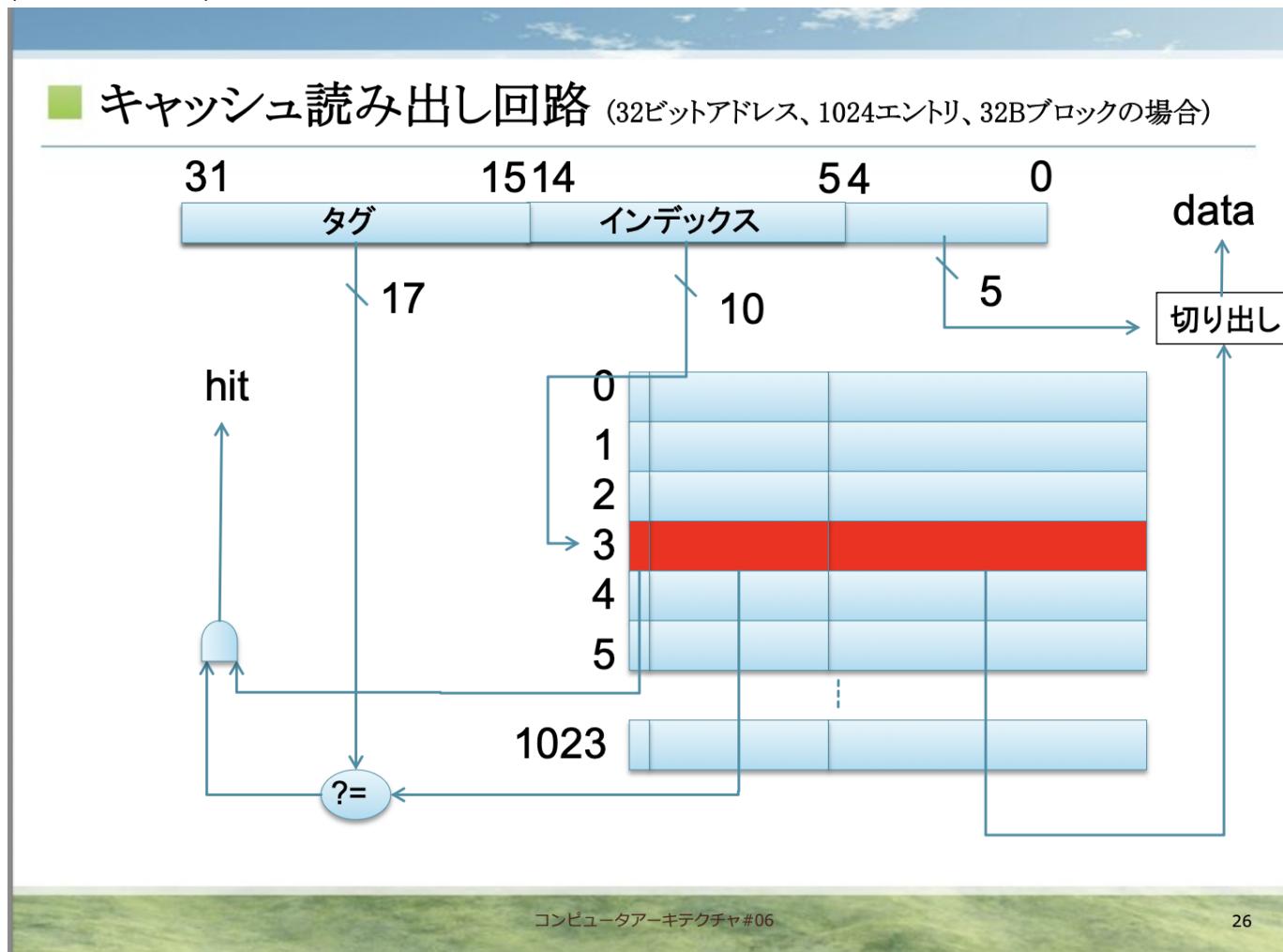
如果使用 M 代表内存地址的总位数，Cache 块大小为 B 字节，Cache 总条目数为 E ，则：

$$\text{Offset 位数} = \log_2 B$$

$$\text{Index 位数} = \log_2 E$$

$$\text{Tag 位数} = M - (\log_2 B + \log_2 E)$$

(Cache 读取电路):



キャッシュラインの衝突 (Cache 行的衝突)

- 冲突的起因: Cache Miss: キャッシュに該当アドレスのブロック/ラインが存在しない場合 (当 Cache 中不存在对应地址的块/行时)
- Miss 后的数据获取: 下位メモリから該当ブロックを読みだして格納 (从下层存储器读取对应数据块并进行存储)
- 冲突的解决: 数据替换: 格納する場所がすでに占められていた場合はキャッシュから追い出して新しいブロックを書き込み (如果存储位置已经被占据, 则将原块从 Cache 中驱逐/淘汰, 然后写入新的数据块)
- 替换的关键: 写入策略 (隐含信息, 注意脏位)

Cache 块配置的灵活性 (ブロック配置の柔軟性)

- 灵活性的一端: 直接映射 (ダイレクトマップ方式 / Direct-Mapped) 直接映射代表了配置灵活性最低, 但访问速度最快、硬件成本最低的一端。
- 灵活性的另一端: 全相联映射 (フル・アソシティブ方式 / Fully Associative) 全相联映射代表了配置灵活性最高, 但访问速度最慢、硬件成本最高的一端, アクセス時間やハードが高価 (访问时间长, 硬件成本高昂)。这是因为需要进行并行搜索, アクセスの時はすべてのエントリを探す (访问时必须搜索所有的 Cache 行)。
- 两极端的权衡 (兩極端)

- 直接映射：以牺牲配置灵活性（高冲突）为代价，换取最快的访问速度和最简单的硬件。
 - 全相联映射：以增加访问时间和硬件成本为代价，换取最大的配置灵活性（最低冲突）。
4. 组相联映射 (Set-Associative)

组相联映射 (Set-Associative) 正是介于这两种极端之间的折衷方案

セット・アソシアティブ方式 (组相联映射方式)

总结组相联映射的关键信息，以表格形式单独输出。

🤝 组相联映射 (Set-Associative) 总结

元素	作用/特点	查找机制
设计定位	中间设计（主流设计），平衡了速度、容量和冲突。	-
Way (路数 \$n\$)	Set 内的可选存储位置数量，例如 2 Way, 4 Way, 8 Way 等。	内存块在 Set 内有 \$n\$ 个位置可选。
Set (组)	Cache 的基本查找单位，由 \$n\$ 个 Way 构成。	Index 唯一确定 Set 号（像直接映射）。
访问流程	结合了两种映射方式的优点。	1. 外部：Direct-Mapped 确定 Set。2. 内部：Fully-Associative 搜索 Set 内的 \$n\$ 个 Way。

联想度带来的差别

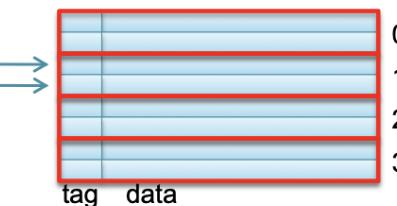
■ 联想度の違い

• 8ブロックのキャッシュ

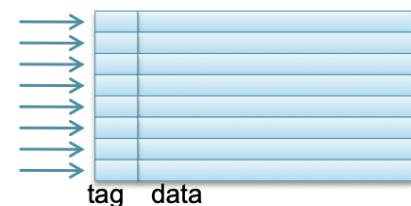
ダイレクトマップ
 (= 1ウェイ)



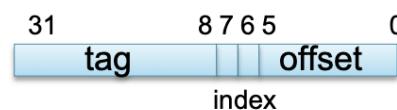
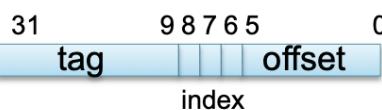
2ウェイ・
セットアソシアティブ



フル・アソシアティブ
 (= 8ウェイ)



1ライン= 64B, 32bit アドレスの場合



- 直接映射的限制 (ダイレクトマップの場合) :

- 同じエントリを使用する別アドレス ⇒ 衝突、競合 (使用相同行 (Entry) 的不同地址 ⇒ 冲突、竞争)
- キャッシュの他エントリが埋まっていなくても追い出し発生 (即使 Cache 的其他行没有被占满，也会发生驱逐)

2. 提高相联度的优势 (N Way 的情况)

- N ウェイであれば N 個の競合まで発生可 (如果是 N 路, 则最多可以容忍 N 个竞争)
- 同じセット内の開いているエントリに入れられる (可以放入同一个 Set 内的空闲行)

3. 性能总结：利用效率与饱和点

- キャッシュ利用効率の向上 (Cache 利用效率的提高)
- 連想度を増していくと効果は飽和 (提高相联度，效果会趋于饱和 (Saturation))
- 饱和点はプログラムによって異なる (饱和点因程序而异)

设计原则：现代 Cache 设计的目标是找到 N 的最佳值，在达到性能提升的饱和点之前，用最小的硬件成本获得最大的命中率收益。

Cache 替换的关键机制——驱逐算法（追い出しアルゴリズム, Replacement Algorithm）。

当 Cache 发生 Miss，需要将数据块从下一级存储器加载到 Cache 中时，如果目标位置已满，就需要使用替换算法来决定淘汰谁。

1. 触发驱逐的条件 (新しく読み込んできたライン)

- 入れようとしたセットの全てのウェイが既に valid (要放入的 Set 中的所有 Way 都已经是 Valid/有效状态)
- どれを追い出す？ (驱逐哪一个？)

2. 经典的驱逐算法：LRU (定番：LRU)

- 最後にアクセスされた時間が最も古いラインを追い出す (驱逐最后一次访问时间最久的 Cache 行)。

驱逐算法是 Cache 性能优化的最后一步，它确保了在有限的 Cache 空间中，总能存放最可能被 CPU 需要的数据

キャッシュミスの 3C (Cache Miss 的 3C 分类) 解读

1. 强制性 Miss (初期参照ミス / Compulsory Miss)

- 定义：そのラインへの最初のアクセス (对该 Cache 行的第一次访问)。

2. 冲突性 Miss (競合ミス / Conflict Miss)

- 同じセットへのアクセスによって追い出されたラインへのアクセス (对同一 Set 的访问导致原 Cache 行被驱逐，之后再次访问被驱逐的行)。
- フルアソシアティブであれば防げるミス (如果是全相联映射，则可以避免的 Miss)。

3. 容量性 Miss (容量性ミス / Capacity Miss)

- キャッシュ容量がたりないために追い出されたラインへのアクセス (Cache 容量不足导致原 Cache 行被驱逐，之后再次访问被驱逐的行)。

- 容量が十分であれば防げるミス (如果容量足够, 则可以避免的 Miss)。

🔍 Cache Miss 的 3C 分类总结

Miss 类型 (日文/英文)	发生原因	本质/定义	改进方法
初期参照ミス (Compulsory Miss)	对某个数据块的第一次访问。	数据最初不在 Cache 中 (Cold Start)。	**预取 (Prefetching) **技术缓解。
競合ミス (Conflict Miss)	多个常用块映射到同一个 Set, 相互竞争驱逐。	相联度低所导致的、可避免的 Miss。	提高相联度 (如从 Direct-Mapped 到 Set-Associative)。
容量性ミス (Capacity Miss)	程序的工作集大于 Cache 的总容量。	Cache 存储空间不足所导致的 Miss。	增大 Cache 容量。

Cache 与主内存之间的数据一致性

1. 数据一致性问题

- キャッシュに存在するデータはメモリのコピー (Cache 中存在的数据是内存的副本)
- コピーへの変更には注意が必要 (对副本的更改需要注意)
- メモリデータとの違い (与内存数据的不一致/分歧)

1. 写入策略: 两种主要方式

- Write-Through 方式 (ライトスルー方式 / 写穿/写通): 机制: キャッシュへの書き込みと同時にメモリへも書き込み (在写入 Cache 的同时, 也写入主内存)
- Write-Back 方式 (ライトバック方式 / 写回): 机制: 書き込みラインが追い出されるときにメモリへ書き込み (当被写入的行被驱逐/替换时, 才写入主内存)
 - 过程: CPU 写入 Cache 时, 只修改 Cache, 并将该行标记为“脏 (Dirty)”。只有当该“脏行”需要被替换出 Cache 时, 才将其内容写回到主内存。

📝 Cache 写入策略的权衡

策略 (日文/英文)	机制 (如何写入)	优点/性能特点	缺点/复杂性
ライトスル 一方式 (Write-Through / 写穿)	写入 Cache 的同时, 也同步写入主内存。	Cache 和内存数据始终保持一致。	每次写操作都访问内存, 写操作慢; 总线流量大。
ライトバッ ク方式 (Write-Back / 写回)	仅写入 Cache, 并将该行标记为“脏”; 当该行被驱逐/替换时, 才写入主内存。	写操作速度快; 可以减少对内存的写入次数 (多次修改只需最后写回一次)。	处理复杂, 需要脏比特来追踪修改; 需要复杂的一致性协议; 在写回前, 内存数据与 Cache 不一致。

写回策略虽然复杂，但由于其显著提高了写操作性能和减少了总线带宽占用，因此现代高性能 CPU 的 L1/L2 Cache 大多采用 Write-Back 策略。

■ Thinking Time 7.3

- ・ 基本CPIが1の（インオーダー）プロセッサ
- ・ メモリステージのレイテンシ：
 - キャッシュヒット時：1サイクル（バブルなしとして良い）
 - キャッシュミス時：100サイクル
 - ストア命令はヒット、ミスに関わらず1サイクル
- ・ ロード命令が20%の割合で含まれるプログラム
 - キャッシュヒット率が95%のとき
 - キャッシュヒット率が80%のとき

性能差は？

解答：

$$\frac{20 \times 0.95 + 80 + 0.05 \times 20 \times 100}{100}$$
$$= 1.99$$

Cycle per instruction

$$\frac{20 \times 0.8 + 80 + 0.2 \times 20 \times 100}{100}$$

$$= 4.99$$

$$4.99$$

$$\frac{1.99}{4.99} \approx 2^{49}$$

Cache hit rate is very important to performance