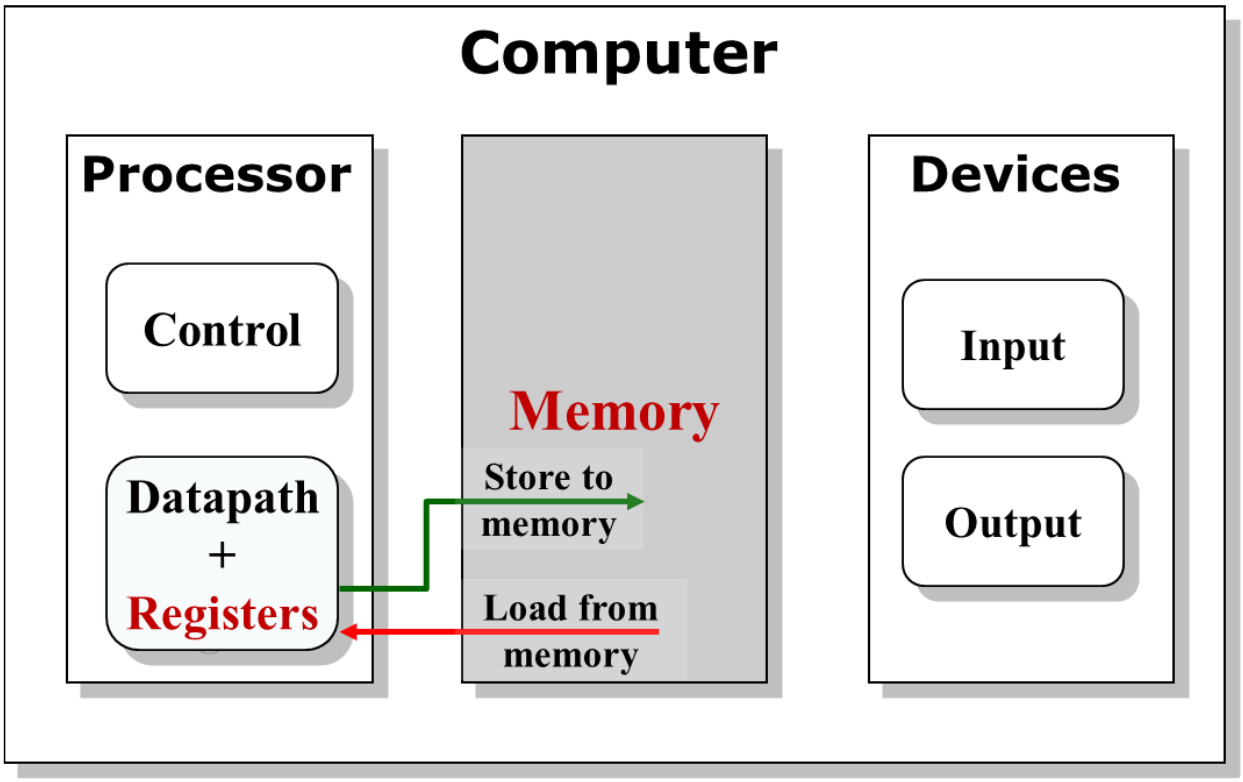
10 - Cache

10.1 Introduction



Registers are in the datapath of the processor. If operands are in memory we have to load them to processor (registers), operate on them, and store them back to memory.

DRAM

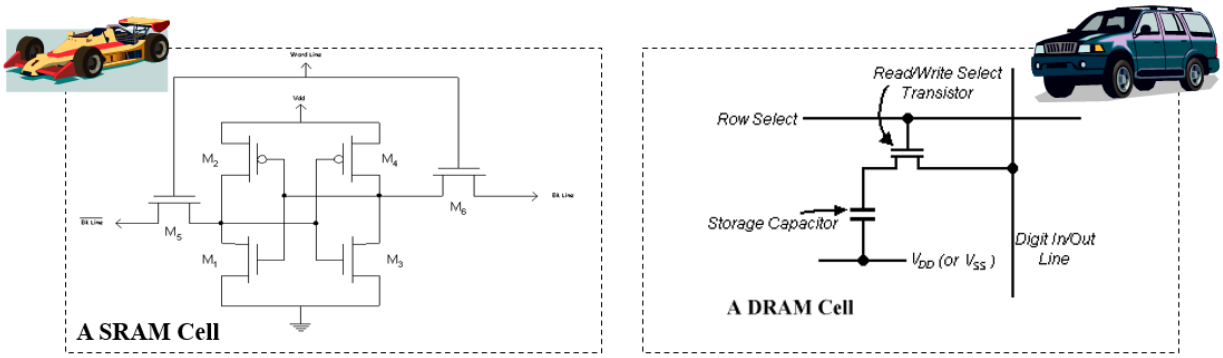
DRAM = DDR SDRAM = Double Data Rate Synchronous Dynamic RAM

Delivers memory on the positive and negative edge of a clock (double rate)

Generations:

1. DDR (`MemClkFreq x 2 (double rate) x 8 words`)
2. DDR2 (`MemClkFreq x 2 (double rate) x 2 x 8 words`)
3. DDR3 (`MemClkFreq x 4 (double rate) x 2 x 8 words`)
4. DDR4 (Lower power consumption, higher bandwidth)

SRAM



SRAM

6 transistors per memory cell

→ **Low density**

Fast access latency of 0.5 – 5 ns

More costly

Uses flip-flops

DRAM

1 transistor per memory cell

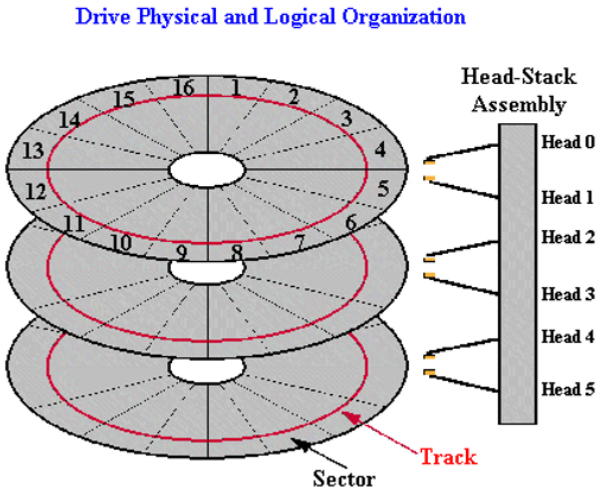
→ **High density**

Slow access latency of 50-70ns

Less costly

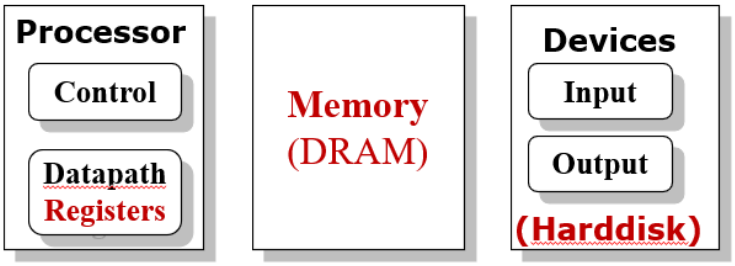
Used in main memory

Magnetic Disk



Typical high-end hard disk:

Average Latency: 4 - 10 ms
Capacity: 500-2000GB



	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	$$$$
SRAM	100s KB	0.5-5 ns	$$$
DRAM	100s MB	50-70 ns	$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	**1 GB**	**1 ns**	**Cheap**

10.2 Cache

Basic Idea

Keep the frequently and recently used data in smaller but faster memory

Refer to bigger and slower memory:

- Only when you cannot find data/instruction in the faster memory

Principle of Locality:

- Program accesses only a small portion of the memory address space within a small time interval

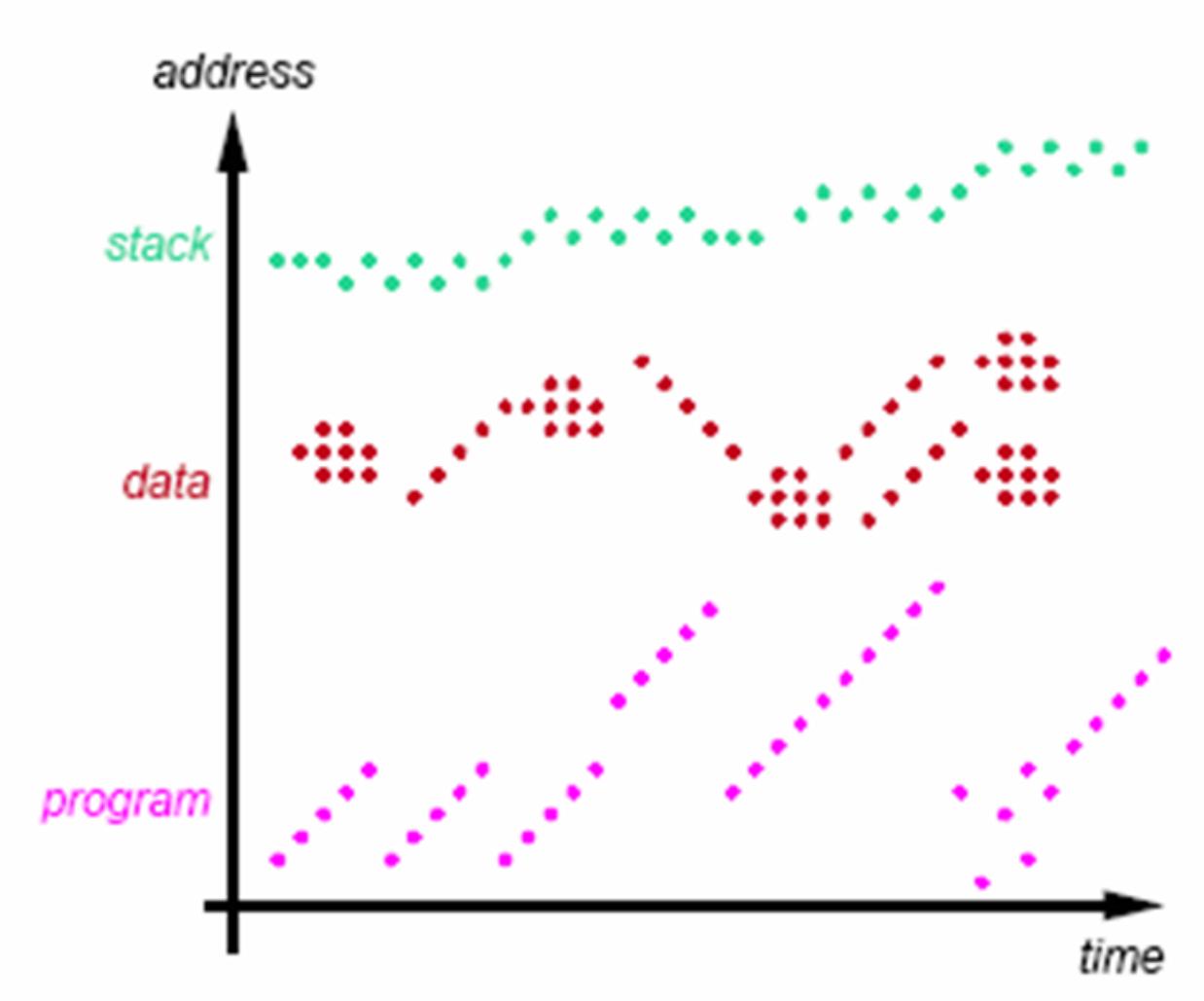
10.2.1 Types of locality

Temporal locality:

- If an item is referenced, it will tend to be referenced again soon

Spatial locality:

- If an item is referenced, the nearby items will tend to be referenced soon



Working Set

Set of locations accessed during Δt

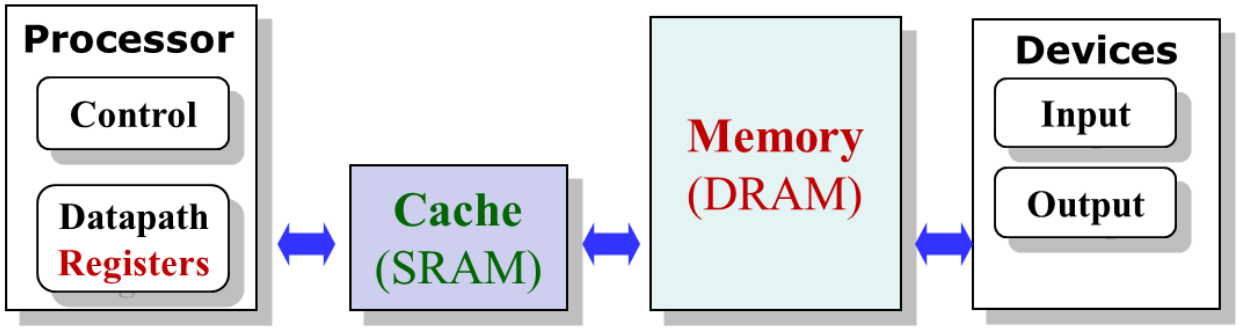
Different phase of execution may use different working sets

Our aim is to capture the working set and keep it in the memory closet to CPU

> ## 工作集 (Working Set)
>
> 工作集是一个动态概念，指的是在一段特定的时间间隔Δt内，程序访问的所有唯一的内存位置（或页面）的集合。这个时间间隔可以是固定的，也可以是基于特定的执行阶段。工作集的大小和内容可能会随着程序执行阶段的不同而变化。
>
> ## 执行阶段和工作集
>
> 程序在其生命周期中可能会经历多个不同的执行阶段，每个阶段可能会使用不同的数据集和代码路径。例如，在初始化阶段，程序可能会加载某些库和数据结构，而在执行计算或响应用户输入的阶段，则可能访问完全不同的内存区域。这些阶段可能具有不同的工作集。

> **为什么工作集重要?**
>
> 1. **性能优化:** 了解程序的工作集对于优化性能至关重要。如果可以将工作集中的数据和代码保留在接近 CPU 的内存（例如，缓存或主内存）中，那么程序的性能可以大幅提高，因为 CPU 访问近处的内存比访问磁盘等远程存储设备要快得多。
> 2. **内存管理:** 操作系统的内存管理器试图优化可用内存的分配，确保最频繁访问的数据（即工作集）位于最快的存储区域。这通常通过页面替换算法来实现，该算法会根据工作集的变化来调整哪些页面应该留在内存中，哪些应该被换出到辅助存储（如磁盘）。
> 3. **预测和调度:** 通过监视工作集的变化，操作系统可以预测程序未来的行为和资源需求，从而更智能地进行任务调度和资源分配

10.2.2 Memory Access



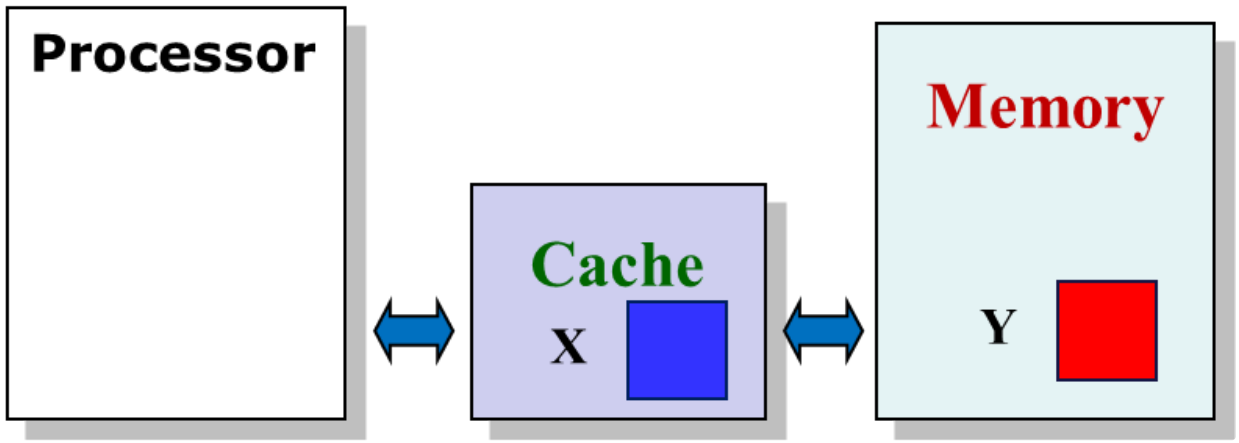
How to make SLOW main memory appear faster?

- Cache - a small but fast SRAM near CPU
- Hardware managed: Transparent to programmer

How to make SMALL main memory appear bigger?

- Virtual memory
- OS managed: Transparent to programmer

Memory Access Time: Terminology



Hit: Data is in cache

- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache

Miss: Data is not in cache

- Miss rate = 1 - Hit rate
- Miss penalty: Time to replace cache block + hit rate

Hit time < Miss penalty

Memory Access Time: Formula

$$\text{Average Access Time} = \text{Hit rate} \times \text{Hit time} + (1 - \text{Hit rate}) \times \text{Miss penalty}$$

Example:

- Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**?
  - Let h be the desired hit rate.

    1 = 0.8h + (1 - h) x (10 + 0.8)
      = 0.8h + 10.8 - 10.8h

    10h = 9.8 -> h = 0.98

    Hence we need a hit rate of **98%**.

10.3 Memory to Cache Mapping

Cache Block/Line:

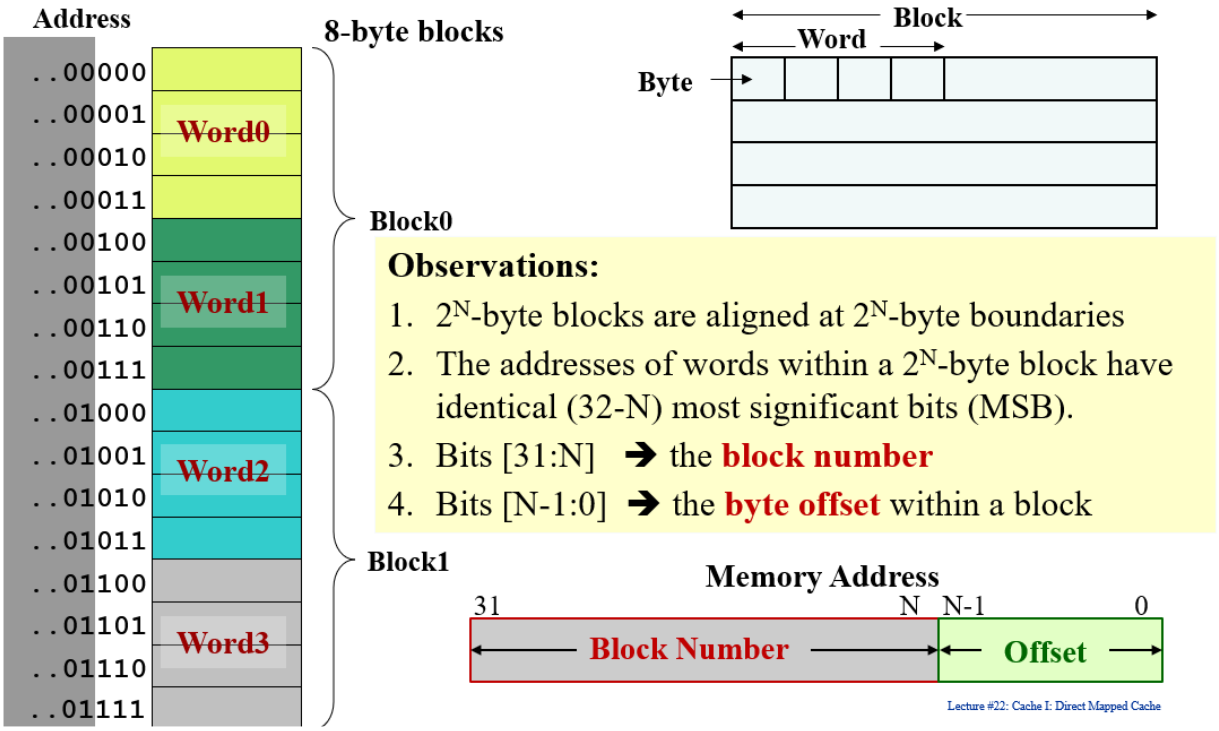
- Unit of transfer between memory and cache

Block size is typically one or more words

- e.g.: 16-byte block ≈ 4-word block
- 32-byte block ≈ 8-word block

> Why the block size is bigger than word size?
>
> 1. **空间局部性 (Spatial Locality)**：
>   - 程序倾向于访问最近访问过的内存位置附近的内存位置。这是由于程序的结构通常是按顺序执行的，并且数据也往往是以数据结构（如数组、结构体等）的形式组织的。
>   - 因此，当一个特定的内存地址被访问时，其附近的地址也很可能很快被访问。将这些数据作为较大的块（而非单个字）一起存储在缓存中，可以预加载这些可能很快就需要的数据，从而减少了未来的缓存未命中。
> 2. **时间局部性 (Temporal Locality)**：
>   - 程序倾向于在短时间内多次访问相同的内存位置。通过在缓存中保存一个比单个字更大的块，系统可以在连续的操作中更有效地使用这些数据，而不是频繁地从主内存中重新加载。
> 3. **传输效率**：
>   - 从主内存到缓存的数据传输是以固定大小的块进行的。传输较大的数据块（而非单个字）更能有效利用内存带宽，因为每次传输都涉及一定的启动（开销）成本。较大的块意味着相对较少的传输，从而减少了总体延迟。
> 4. **降低缓存未命中率 (Cache Misses)**：

- 当CPU访问的数据不在缓存中时，就会发生缓存未命中。较大的缓存块可以减少缓存未命中的可能性，因为更多的相关数据已经预加载到缓存中。



10.4 Direct Mapped Cache

直接映射缓存的工作原理：

在直接映射缓存中，主存储器（main memory）被划分为多个块（blocks），而缓存（cache）则被划分为多个线（lines）或槽（slots）。这些缓存线用于存储来自主存储器的数据块。当CPU需要读取特定地址的数据时，系统会检查这些数据是否已在缓存中。

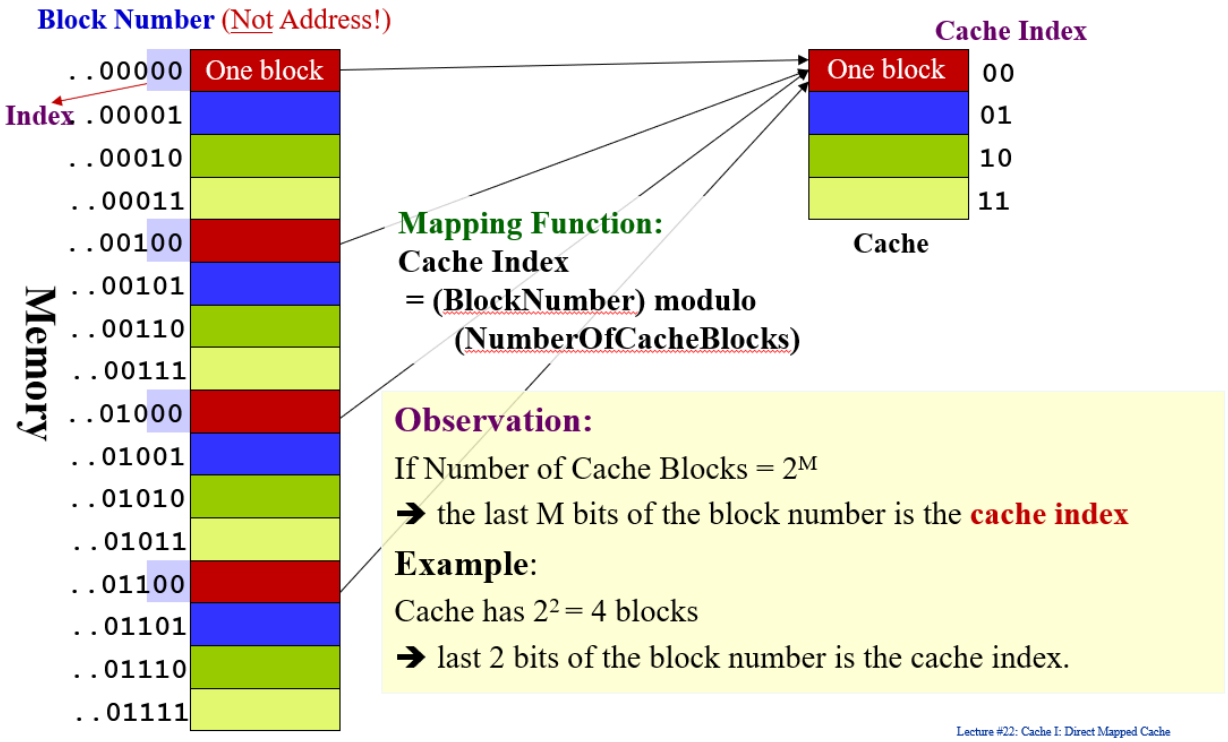
为了决定一个内存块应该存储在缓存的哪个位置，以及如何在缓存中找到这些块，系统会使用一种映射策略。在直接映射缓存中，使用了内存地址的一部分来确定一个特定的缓存线。

缓存索引的作用：

内存地址通常包含以下几个部分：

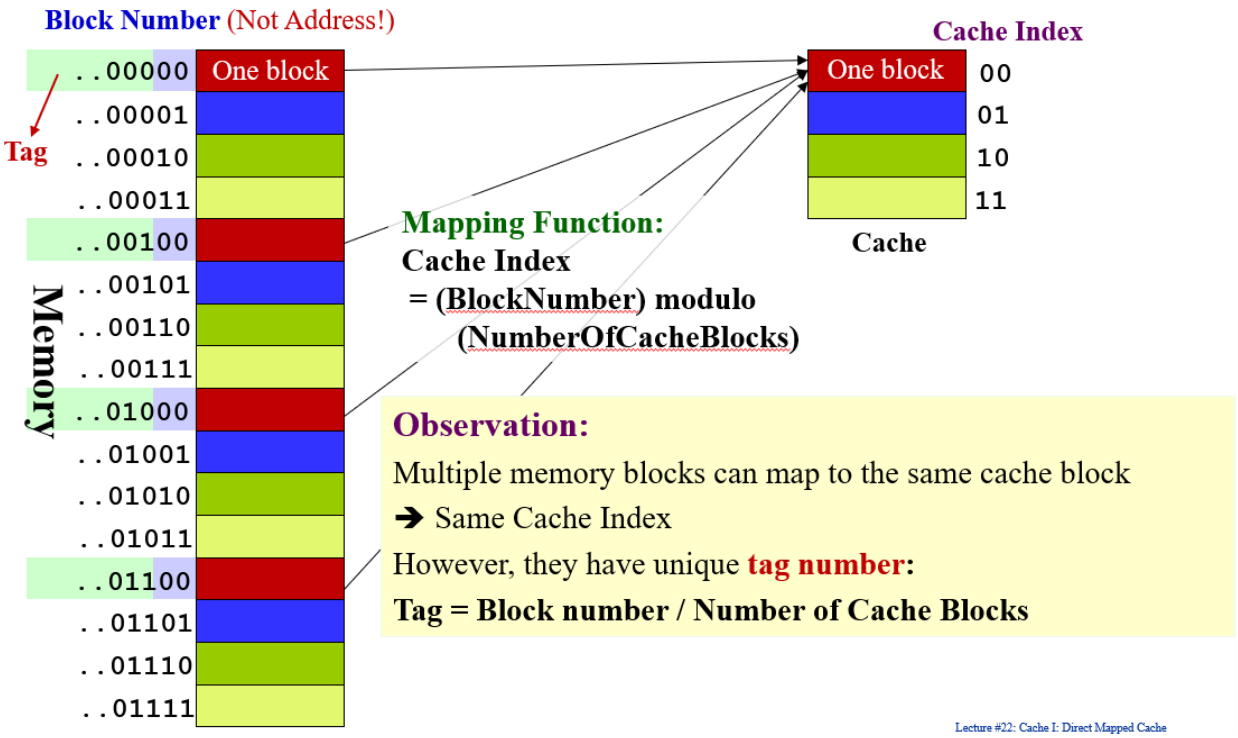
1. **标记（Tag）**：用于唯一标识一个内存块。当缓存检查某个特定缓存线时，它会比较地址标记和缓存线中存储的标记，以确定所需数据是否在该缓存线中。
2. **索引（Index）**：这是直接决定内存块在缓存中位置的部分。系统通过内存地址的索引字段来选择应该使用哪个缓存线。换句话说，索引用于“直接映射”内存块到特定的缓存线。
3. **块内偏移（Block Offset）**：当存储块大小大于一个字（word）时，块内偏移用于在一个块内部定位特定的字。

10.4.1 Cache Index and Tag



计算索引：

- 即我们需要最少的比特位数来表示所有的block。如果一共有2^M个block，那么我们需要M个bit位才能表示所有的block，所以block number的最后M位即位索引位



标记(Tag)

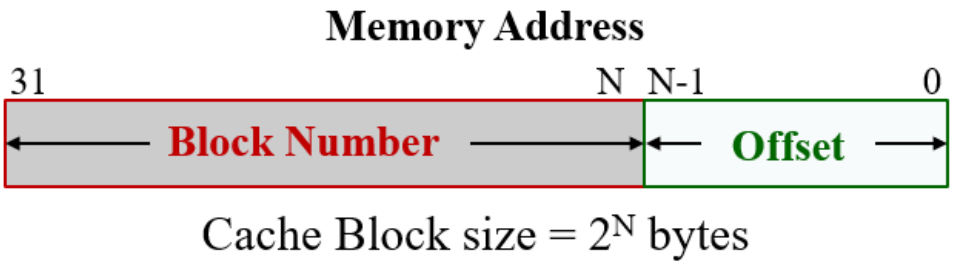
定义与功能:

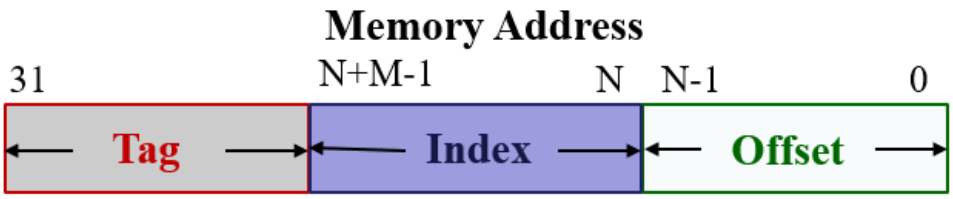
- "标记"是内存地址的一部分，用于在缓存中唯一标识一个数据块。由于缓存容量比整个内存小，因此不可能将所有内存块同时加载到缓存中。标记用于区分不同的内存块，即使它们可能映射到缓存的同一索引位置。
- 当处理器试图访问缓存时，系统会检查所选缓存行中存储的标记与当前请求的内存地址的标记部分是否匹配。如果这两个标记相同，就会发生“缓存命中”；否则，就会发生“缓存未命中”。

计算标记:

- `Tag = Block number / Number of Cache Blocks`

10.4.2 Mapping





Cache Block size = 2^N bytes

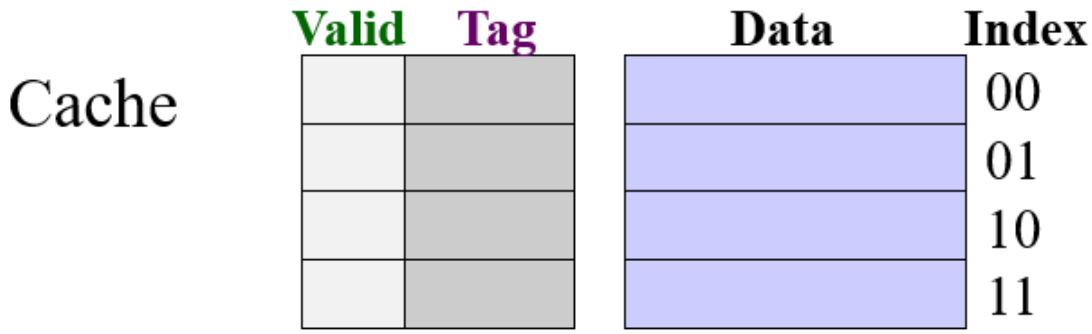
Number of cache blocks = 2^M

Offset = N bits

Index = M bits

Tag = 32 – (N + M) bits

10.4.3 Cache Structure



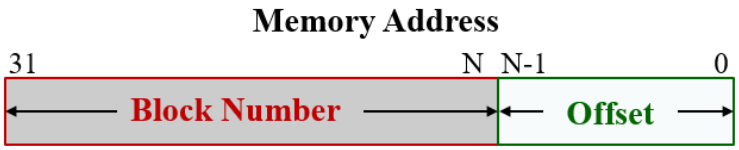
Along with a data block(line), cache also contains the following administrative information

1. Tag of the memory block
2. Valid bit indicating whether the cache line contains valid data

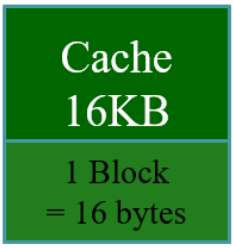
When is there a cache hit?

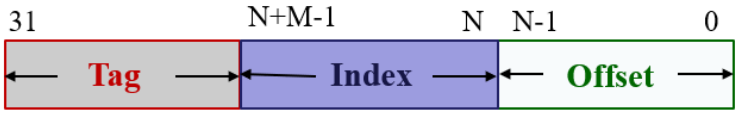
```
Valid[index] == True AND Tag[index] == Tag[memory address]
```

10.4.4 Example

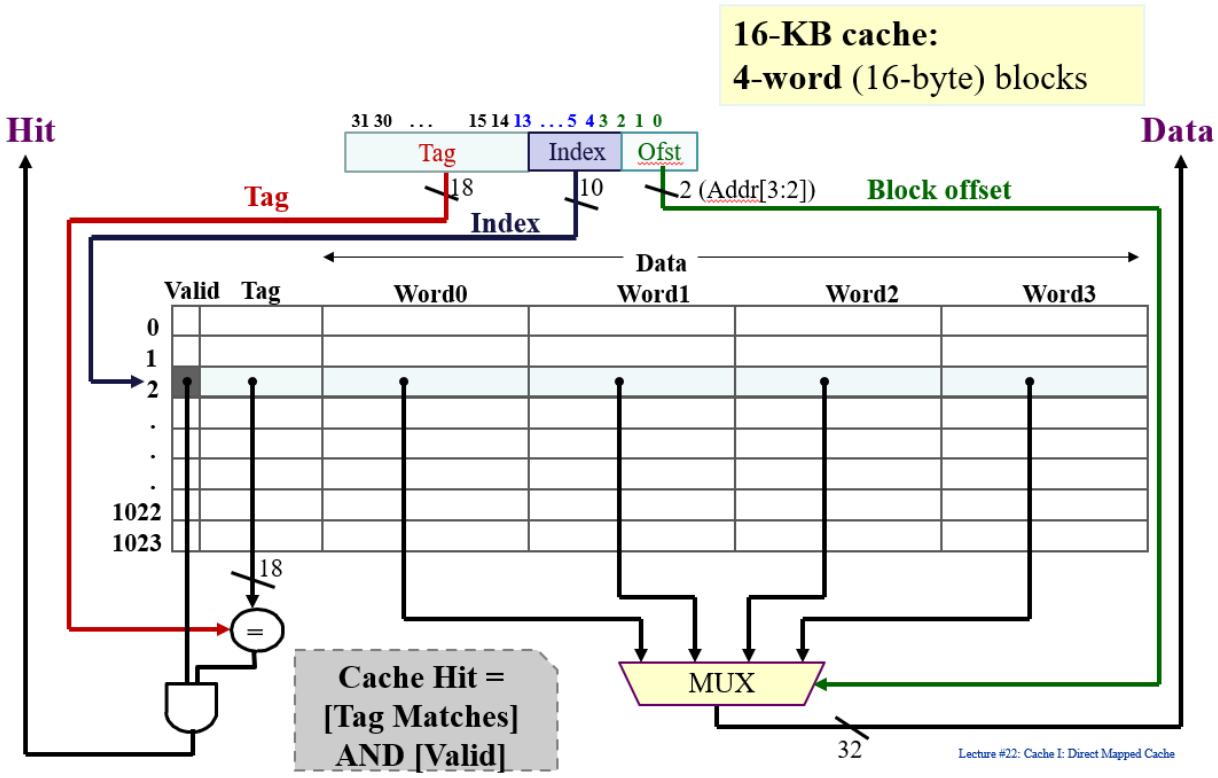


Offset, N = 4 bits
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = 2^{28}



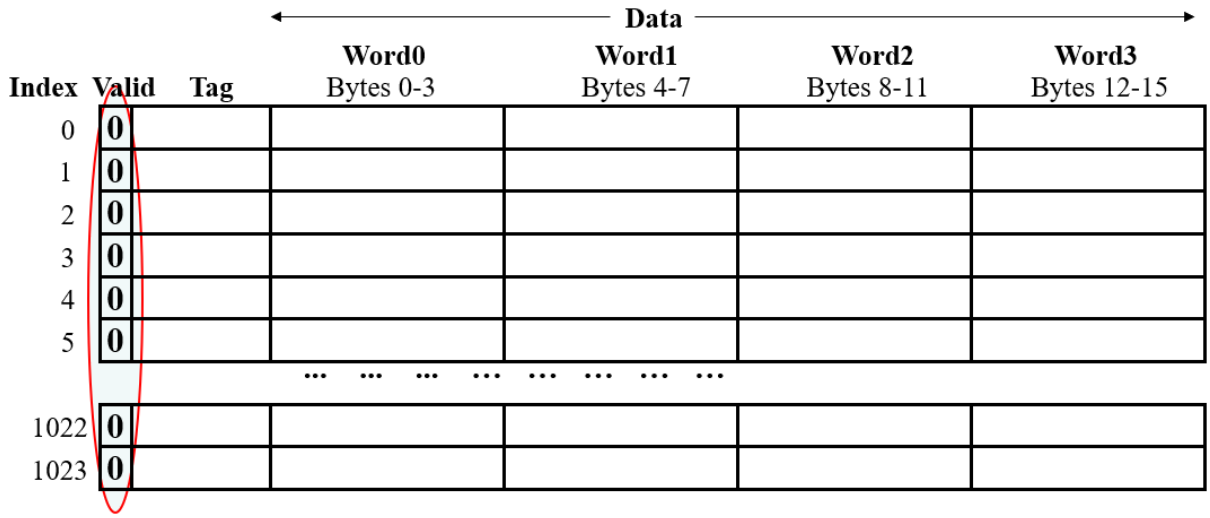


Number of Cache Blocks
= 16KB / 16bytes = 1024 = 2^{10}
Cache Index, M = 10bits
Cache Tag = 32 – 10 – 4 = **18 bits**



10.5 Reading Data

#1 Initial State

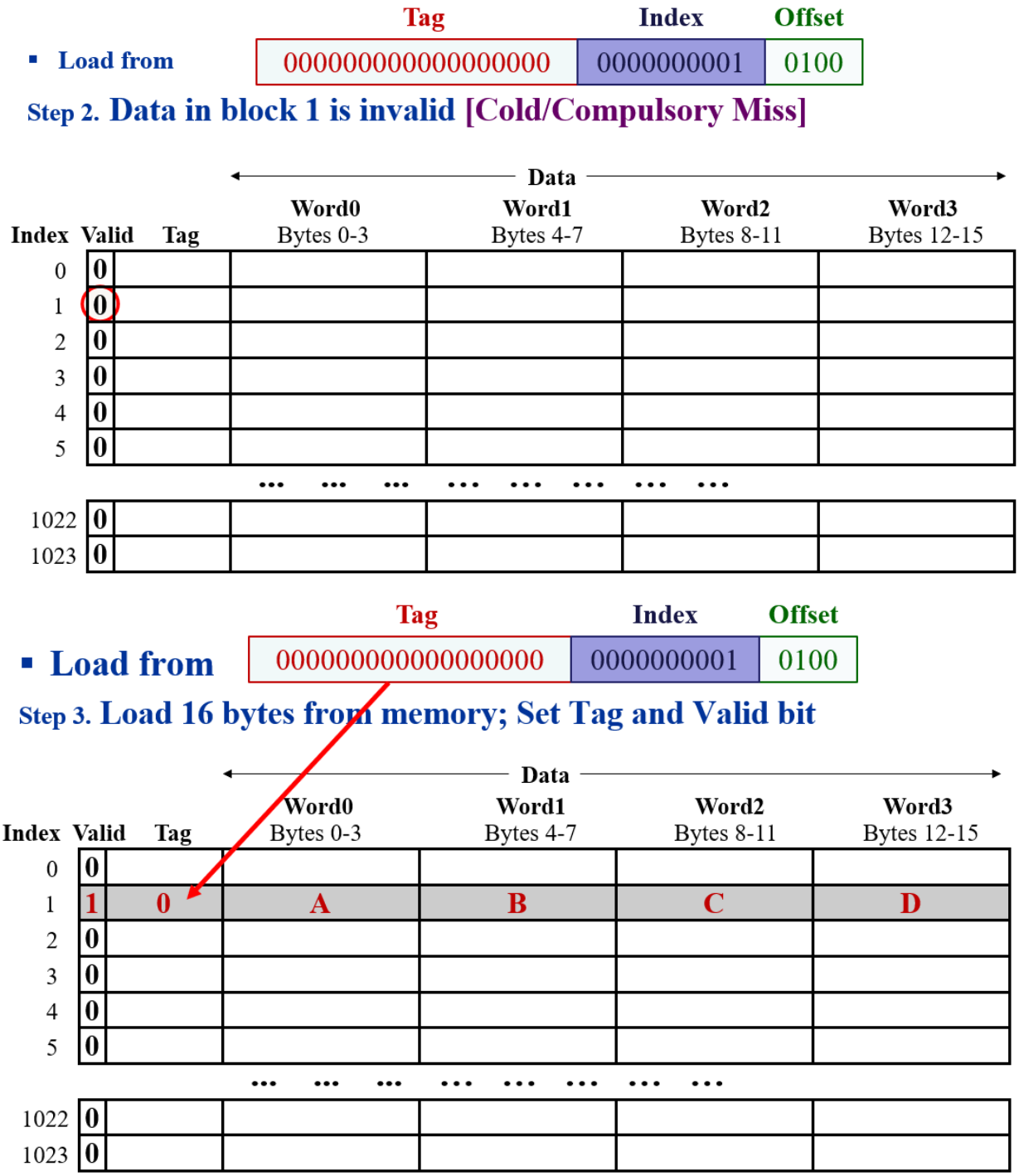


起始状态：

1. 所有的field都是空的
2. 所有的valid bit都为0

> valid bit代表这个slot有没有被写入过数据。在判断缓存命中时需要判断valid bit为1才能正确命中

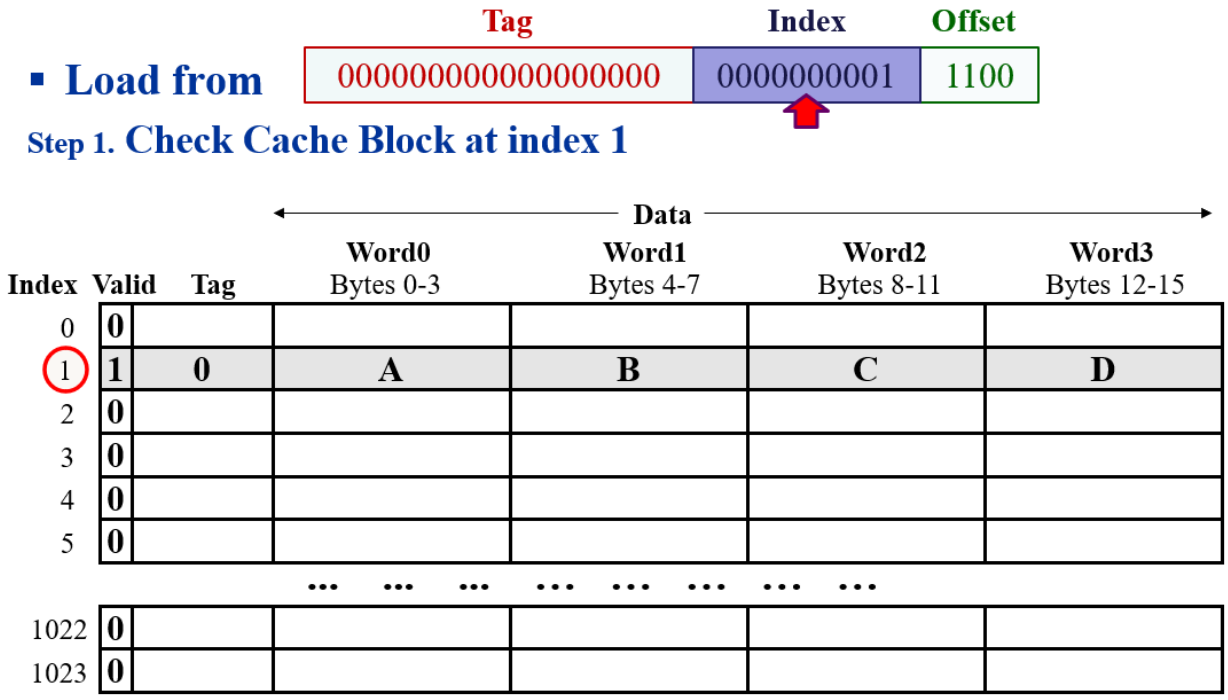
#2 First read data



此处读取了index为1，tag为0，offset为0100的块。

1. 首先检索到缓存索引为1的插槽，其valid bit为0，意味着这个slot没有被写入过。判定为cold/compulsory miss
2. 写入tag，和16 bytes (4 words)的数据。虽然寄存器每次只需要1 word的数据，但是根据时间局部性(Temporal locality)和空间局部性(Spatial locality)，我们写入此word所在的整个block。并将valid bit设置为1
3. 读取offset代表的word，0100即byte 4开始的word，即为word1，传递给寄存器

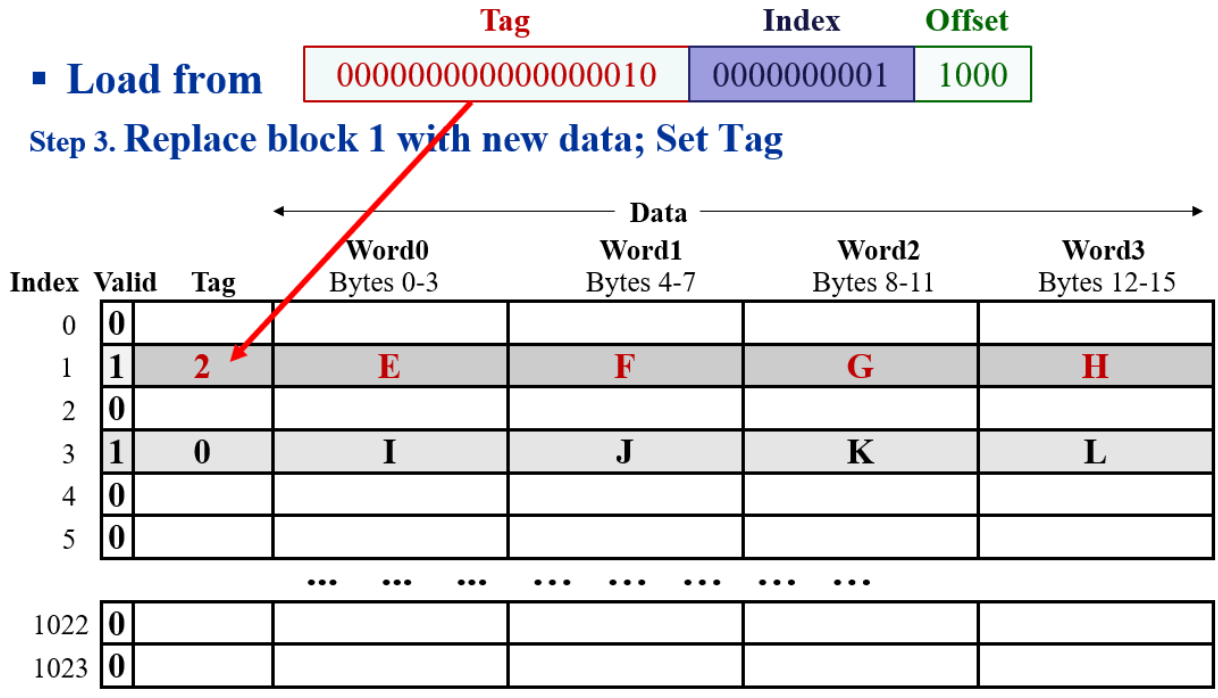
#3 - Cache hit



当寄存器再次需要同一个index但不同offset的word时：

1. 首先检查index索引值，找到所对应的slot
2. 检查valid bit是否为1，以及tag值是否一致。tag作为内存中每个block的特征码是独一无二的
3. 根据offset偏移量确定所需要的word是从1100=12 byte开始的，所以传递word3给CPU寄存器

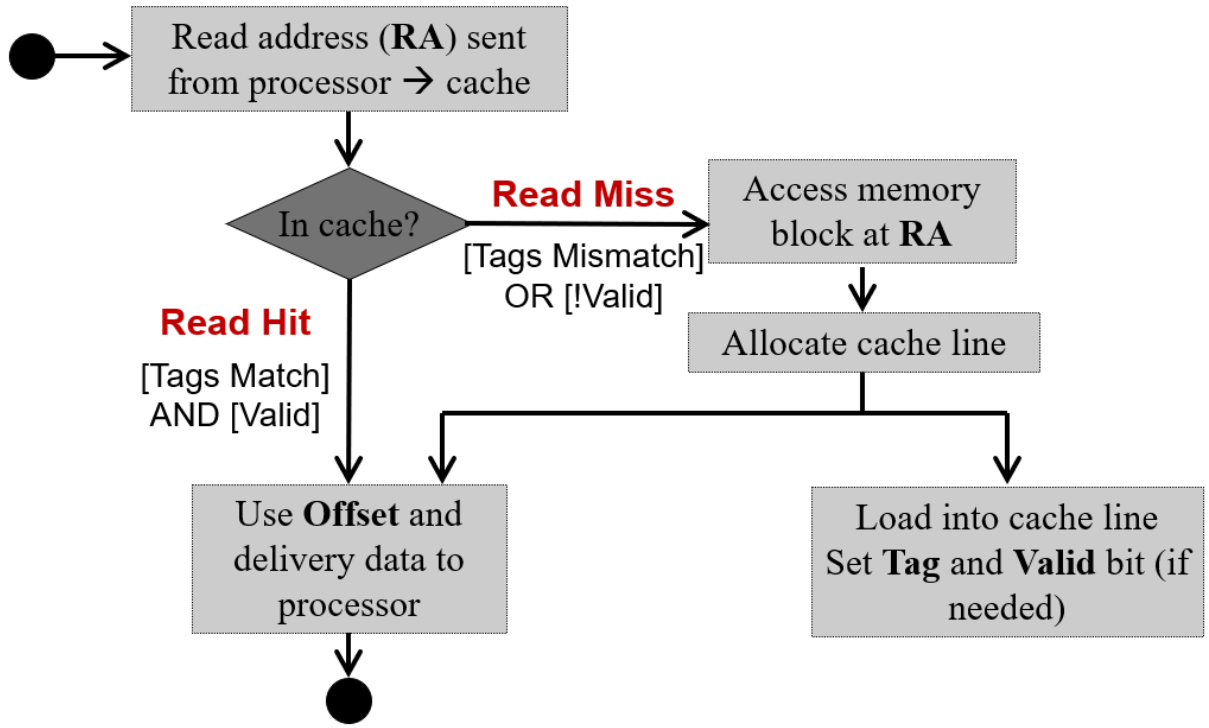
#4 - Different tag



如果index和valid bit能够对应上，但是tag对应不上，依旧被判定为miss (cold miss)

> 需要替换该index对应的slot中的所有数据，包含tag和data

Summary



10.6 Type of Cache miss

Compulsory misses:

- On the first access to a block; the block must be brought into cache
- Also called cold start misses or first reference misses

Conflict misses:

- Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set
- Also called collision misses or interference misses

Capacity misses

- Occur when blocks are discarded from cache as cache cannot contain all blocks needed

> 1. **Compulsory Misses (强制性缺失)**：
>   - 这类缺失发生在对一个数据块的首次访问；因为数据块还没有被加载到缓存中，所以必须从更低一级的内存（如主内存）中取出数据块。
>   - 这也被称为冷启动缺失（cold start misses）或首次引用缺失（first reference misses），因为它们通常发生在程序刚开始运行时，此时缓存未被充分利用。
> 2. **Conflict Misses (冲突缺失)**：

   - 这类缺失发生在直接映射缓存（direct-mapped cache）或组相联缓存（set-associative cache）中，当多个数据块映射到同一个缓存块或集合（set）时。如果新来的数据块与已经在缓存位置的数据块发生冲突，已在缓存中的数据块将被替换出去。
   - 这也被称为碰撞缺失（collision misses）或干扰缺失（interference misses），因为它们是由于不同数据块之间的映射冲突导致的。
3. **Capacity Misses（容量缺失）**：
   - 这类缺失发生在缓存无法容纳所有需要的数据块时。如果程序访问的数据块数量超过了缓存的容量，缓存中的一些数据块将被替换出去，以便为新的数据块腾出空间。当再次需要被替换出去的数据块时，就会发生容量缺失。
   - 这种情况表明，即使缓存中没有冲突，由于缓存容量本身的限制，也无法避免缺失的发生。

10.7 Changing Cache Content: Write Policy

Cache and main memory are inconsistent

- Modified data only in cache, not in memory

Solution 1: Write-through cache

- Write data both to cache and to main memory

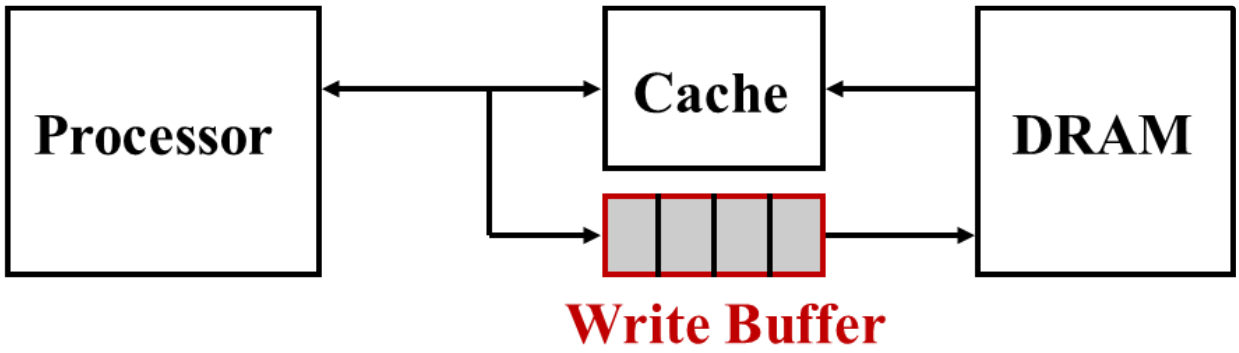
Solution 2: Write-back cache

- Only write to cache
- Write to main memory only when cache block is replaced

当数据被修改后，缓存中的信息和主内存中的信息可能会不一致。为了处理这种不一致性，有两种常见的策略：

1. **写直达（Write-through）缓存**：
   - 在这种策略中，当缓存中的数据被修改时，同时也会将修改的数据写入到主内存中。这样可以确保主内存中的数据始终是最新的，也就是说，缓存和主内存中的数据始终是一致的。
   - 优点是简单、一致性好，当缓存中的数据发生更改时，不需要额外的操作就能保证与主内存的同步。
   - 缺点是每次写操作都需要访问主内存，这会带来额外的时间开销，尤其是当写操作非常频繁时。
2. **写回（Write-back）缓存**：
   - 在这种策略中，修改后的数据仅仅被写回到缓存中，而不是立即写入主内存。只有当缓存中的数据需要被替换，也就是说，当新的数据需要加载到缓存中而缓存已满时，缓存中被修改过的数据（脏数据）才会被写回主内存。
   - 优点是减少了对主内存的写入操作，因为不是每次缓存数据更新都要写入主内存，这可以提高系统的整体性能，特别是在写操作频繁的场景下。
   - 缺点是一致性问题更加复杂。在多核或多处理器系统中，如果不同的处理器缓存中存储了同一主内存位置的不同副本，就需要更复杂的一致性协议来避免数据不一致的问题。

10.7.1 Write-Through Cache



Problem:

- Write will operate at the speed of main memory

Solution:

- Put a write buffer between cache and main memory
  - Processor: writes data to cache + write buffer
  - Memory controller: write contents of the buffer to memory

> 问题：
>
> - 使用写直达策略时，每次写操作都会同时更新缓存和主存（main memory）。由于主存的写速度通常远低于缓存和处理器的速度，因此每次写操作都会被主存的慢速度拖慢，这影响了整个系统的性能。
>
> 解决方案：
>
> - 为了解决这个问题，系统中引入了一个“写缓冲区”（write buffer）。写缓冲区是一种快速存储器，位于缓存和主存之间。
>   - 当处理器执行写操作时，它只需要将数据写入缓存和写缓冲区。由于这两者的速度都很快，处理器不会因为等待写操作完成而闲置，从而可以继续执行后续操作。
>   - 同时，内存控制器（memory controller）会在后台处理写缓冲区中的数据，将其写入较慢的主存中。这一步是异步进行的，不会影响处理器的正常工作。

10.7.2 Write-Back Cache

Problem:

- Quite wasteful if we write back every evicted cache blocks

Solution:

- Add an additional bit (dirty bit) to each cache block
- Write operation will change dirty bit to 1
  - Only cache block is updated, no write to memory
- When a cache block is replaced, only writes back to memory if dirty bit is 1

> 问题：

- 如果我们在替换缓存块时每次都将其写回主存，这将非常浪费资源，因为不是所有被替换的缓存块都包含更新过的数据。在一些情况下，缓存块的数据可能自从被加载到缓存以来并没有被修改过，因此将其写回主存是不必要的。

解决方案：

- 为了更有效地管理缓存内容的写回，系统引入了“脏位”（dirty bit）这一概念。脏位是附加在每个缓存块上的一个额外的位。
  - 当处理器写入缓存时，它只更新缓存块中的数据，并将对应的脏位设置为1，表明该缓存块已被修改，与主存中的相应块内容不一致。这个过程中，并不会有数据写回到主存。
  - 当需要替换缓存块时，系统会检查脏位。如果脏位为1（表示缓存块已被修改），系统才将该缓存块的内容写回主存。如果脏位为0（表示缓存块自加载后未被修改），则无需进行写回操作。

10.7.3 Handling Cache Misses

On a Read Miss:

- Data loaded into cache and then load from there to register

Write miss option 1: Write allocate

- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy (Write-through or Write-back)

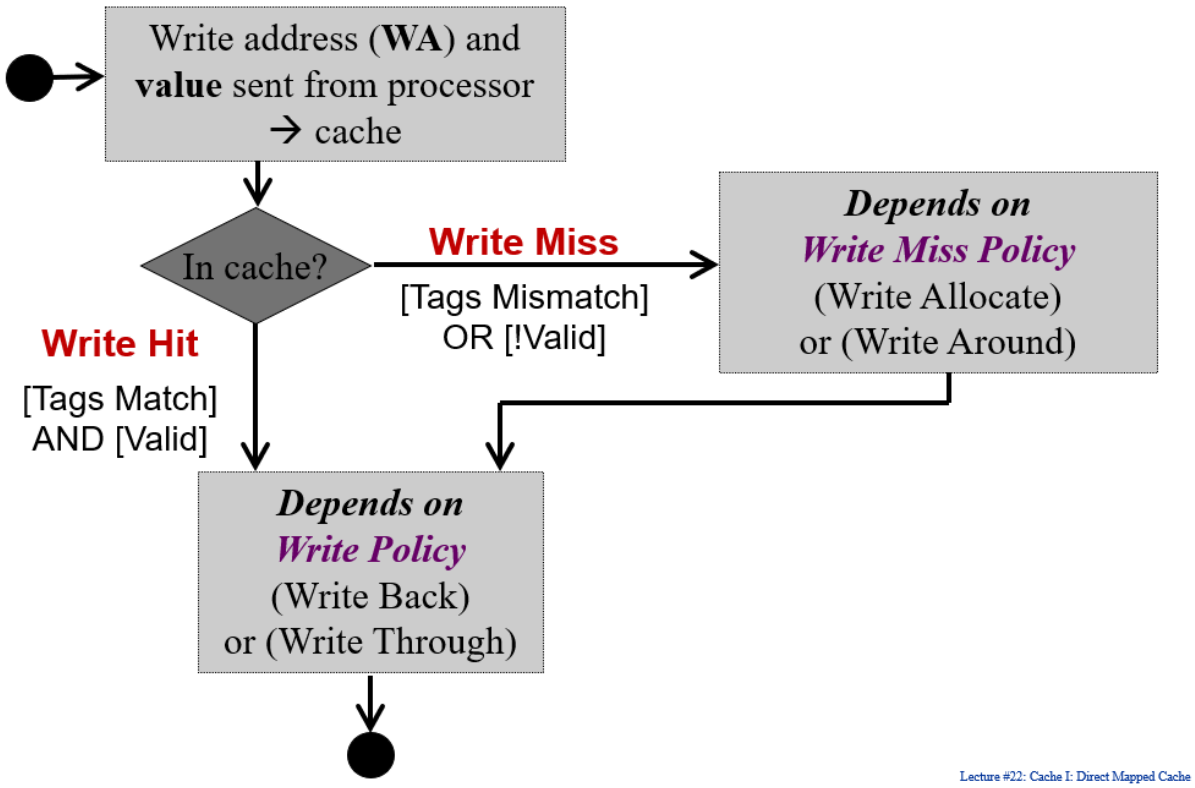
Write miss option 2: Write around

- Do not load the block to cache
- Write directly to main memory only

1. **读缺失（Read Miss）：**
   - 当处理器需要读取的数据不在缓存中时（即缓存未命中或读缺失），系统从主存中加载该数据块到缓存中。然后，数据从缓存传输到需要它的处理器寄存器中。这种方式确保了该数据的后续访问能够更快，因为它现在已经在缓存中了。
2. **写缺失（Write Miss）：** 写缺失发生时，处理器想要写的数据不在缓存中。这时有两种主要的处理策略：
   - **写分配（Write Allocate）：**
     - 当写操作缺失时，首先从主存中将整个数据块加载到缓存中。
     - 接着，处理器只更改缓存中相应的必要数据（单个字或字节）。
     - 至于这个更改过的数据是否立即写回到主存，取决于采用的写策略（写直达或写回）。如果是写直达缓存，数据同时会被写到缓存和主存中。如果是写回缓存，数据更新只发生在缓存中，只有在数据块被替换出缓存时，更改才会写回主存。
   - **绕写（Write Around）：**
     - 在这种策略中，当写缺失发生时，数据不会被加载到缓存中。
     - 相反，更改的数据直接写入到主存中，缓存不参与这次操作。

- 这意味着对这部分数据的后续读取可能会导致缓存缺失，因为数据没有被缓存。这个策略通常用于那些认为不太可能再次用到的数据，或者写操作非常频繁，缓存可能很快就会被新数据填满的情况。

10.7.4 Summary



10.8 Set Associative (SA) Cache

N-way Set Associative Cache

- A memory block can be placed in a fixed number (N) of locations in the cache, where $N > 1$

Key Idea:

- Cache consists of a number of sets:
  - Each set contains N cache blocks
- Each memory block maps to a unique cache set
- Within the set, a memory block can be placed in any of the N cache block in the set

Set Associative (SA) Cache是一种折中的缓存映射策略，结合了直接映射缓存（Direct Mapped Cache）的高效性和全关联缓存（Fully Associative Cache）的灵活性。它试图在这两种策略的优势之间找到平衡，减少缓存未命中的次数，同时保持合理的硬件复杂度和成本。

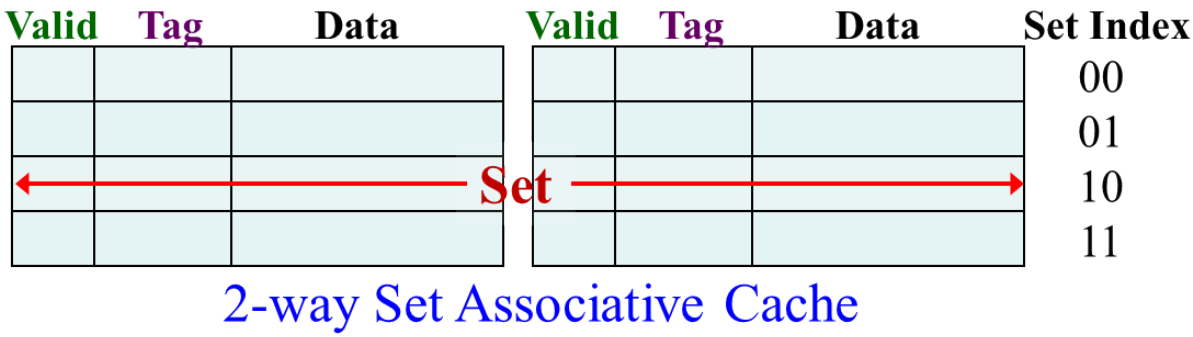
Set Associative缓存的工作原理如下：

1. **集合（Set）**：缓存被划分为多个集合（sets），每个集合包含几个缓存行（cache lines）或块（blocks）。这些块是缓存中可以存储数据的单位。

2. **关联度（Associativity）：** 每个集合中的块数定义了缓存的“关联度”。例如，如果每个集合有四个块，则该缓存是4路组相联的。
3. **映射：** 当主存中的一个块需要被加载到缓存中时，首先会根据该块的地址计算出它应该存储在哪个集合中。这个计算过程通常基于地址的某些位，并且每个集合对应主存中的多个块。然而，一个给定的块只能映射到一个特定的集合。
4. **替换策略（Replacement Policy）：** 如果计算出的目标集合已满（即每个块都已被其他数据占用），缓存必须决定哪个块将被替换以腾出空间。这是通过所谓的替换策略来完成的，常见的替换策略有最近最少使用（LRU）、随机（Random）等。

通过这种方式，Set Associative缓存降低了发生冲突缺失（多个内存地址映射到同一缓存位置）的可能性，因为现在每个内存块有多个潜在的缓存位置可供选择。这增加了一些查找所需数据的复杂性（因为现在必须在一个集合的所有块中查找），但通常能够显著提高缓存的命中率，尤其是在工作集大小适中，且访问模式较为分散的情况下。

10.8.1 Structure



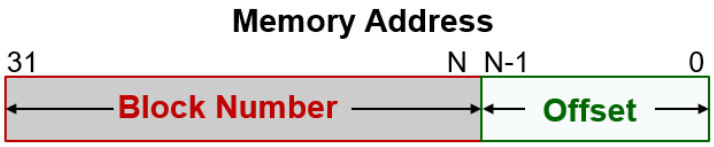
An example of 2-way set associative cache

- Each set has two cache blocks

A memory block maps to a unique set

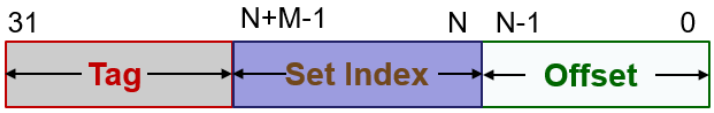
- In the set, the memory block can be placed in either of the cache blocks
- Need to search both to look for the memory block

10.8.2 Mapping



Cache Block size = 2^N bytes

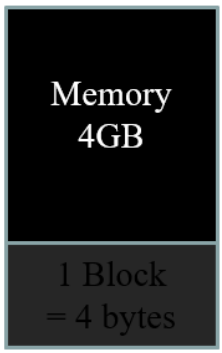
Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)

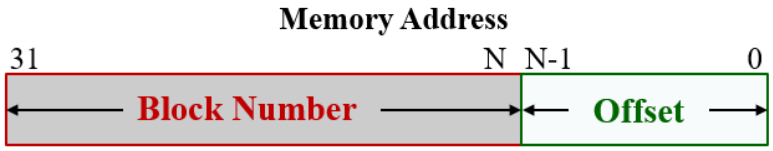


Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = N bits
Set Index = M bits
Tag = 32 – (N + M) bits

Observation:
It is essentially unchanged from the direct-mapping formula

10.8.3 Example

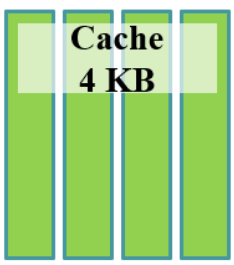




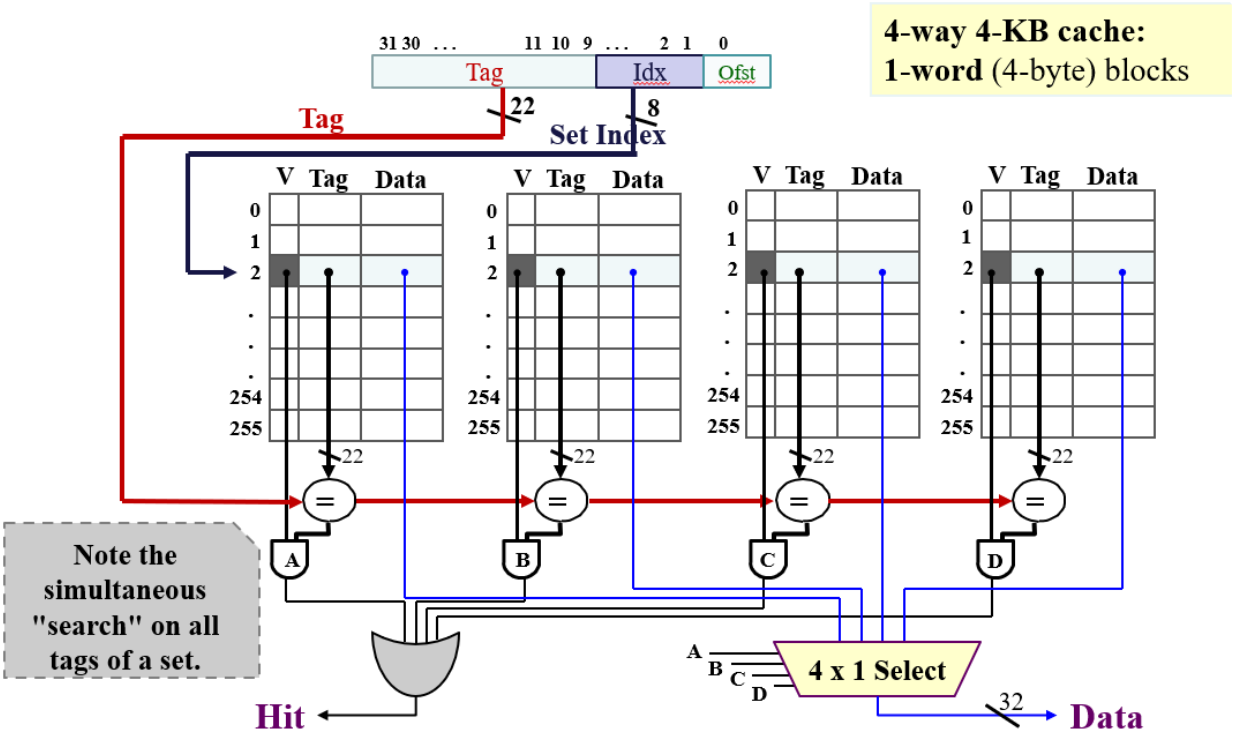
Offset, N = 2 bits
Block Number = 32 – 2 = **30 bits**
Check: Number of Blocks = 2^{30}

Number of Cache Blocks
= 4KB / 4bytes = 1024 = 2^{10}

4-way associative, number of sets
= 1024 / 4 = 256 = 2^8
Set Index, M = 8 bits



Cache Tag = 32 – 8 – 2 = **22 bits**



10.8.4 SA Cache Example

Setup

Given:

- Memory access sequence: `4,0,8,36,0`
- 2-way set-associative cache with a total of four 8-byte blocks (total of 2 sets)
- Indicate hit/miss for each access

Offset, N = 3 bits

Block Number = 32 – 3 = 29 bits

2-way associative, number of sets= 2 = 2^1

Set Index, M = 1 bits

Cache Tag = 32 – 3 – 1 = **28 bits**

Load #1

Load from `4` :

Check: Both blocks in Set 0 are invalid (cold miss)

Result: Load from memory and place in Set 0 - Block 0

> 原先index位表示的是写入哪一个block，现在由于在block上层还有set，所以这里的index是set index

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	~~0~~ 1	0	M[0]	M[4]	0			
1	0				0			

Load #2

Load from `0` :

Result: Valid bit and Tags match in Set 0 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	0				0			

Load #3

Load from `8` :

Check: Both blocks in Set 1 are invalid

Result: Load from memory and place in Set 1 - Block 0

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	~~0~~ 1	0	M[8]	M[12]	0			

Load #4

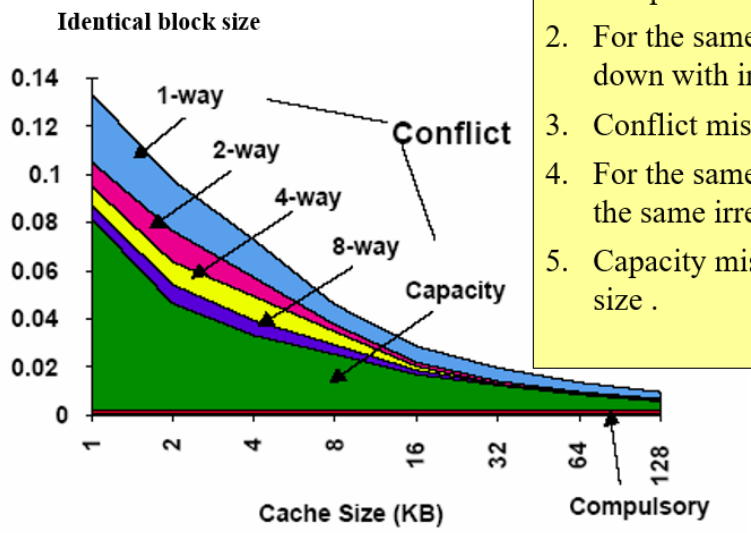
Load from `36` :

Check: Valid is 1 but tag mismatched in Set 0 - Block 0, while Set 0 - Block 1 is invalid

Result: Load from memory and place and place in Set 0 - Block 1

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	~~0~~ 1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

10.9 Cache Performance



Observations:

1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size .

Total Miss = Cold miss + Conflict miss + Capacity miss

Capacity miss (FA) = Total miss (FA) – Cold miss (FA), when Conflict Miss➔0

10.10 Block Replacement Policy

Set Associative or Fully Associative Cache:

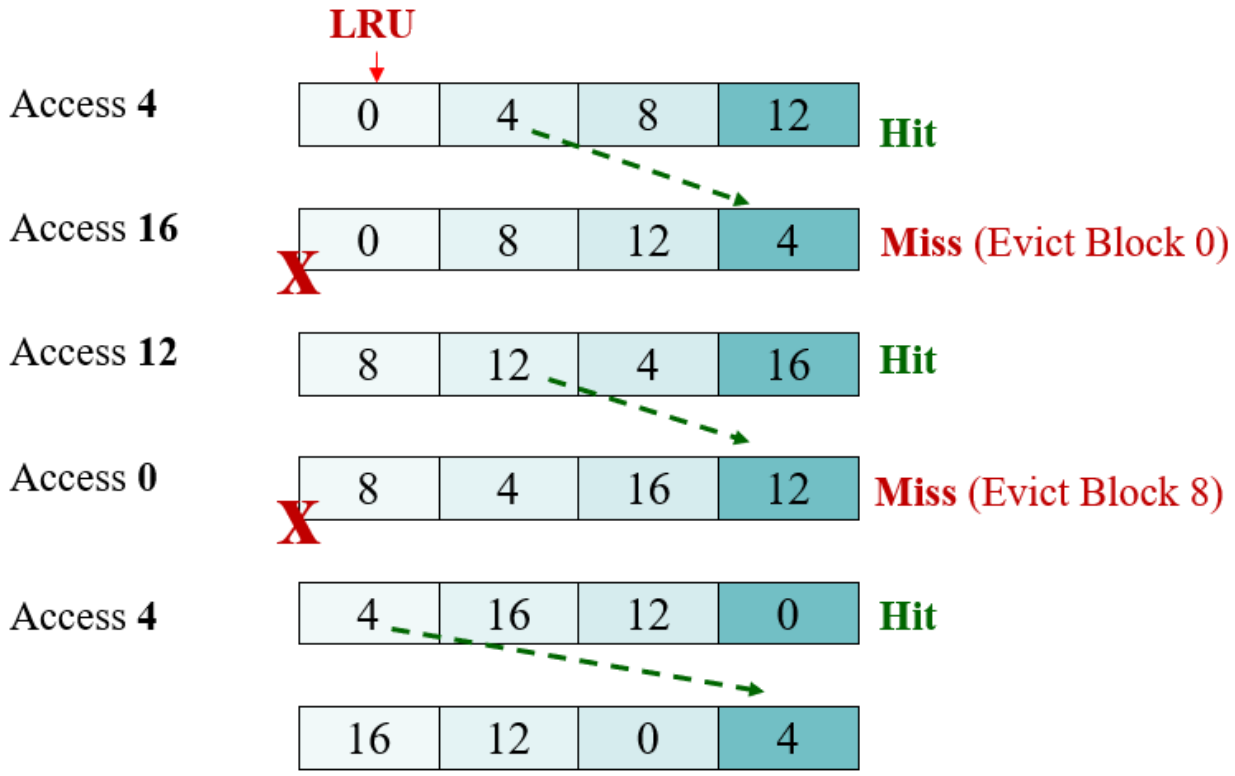
- Can choose where to place a memory block
- Potentially replacing another cache block if full
- Need block replacement policy

Least Recently Used (LRU)

- How: For cache hit, record the cache block that was accessed
  - When replacing a block, choose one which has not been accessed for the longest time
- Why: Temporal locality

LRU policy in action:

- 4-way SA cache
- Memory accesses: `0 4 8 12 4 16 12 0 4`



> Like a heap, the used block moved to the bottom. If a replacement is needed, replace the top block (least use)

Drawback for LRU:

- Hard to keep track if there are many choices

Other replacement policies:

- FIFO
- Random replacement (RR)
- Least Frequently Used (LFU)

10.11 Summary

10.11.1 Cache Organizations

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

10.11.2 Cache Framework

Block Placement: Where can a block be placed in cache?

Direct Mapped:	N-way Set-Associative:
• Only one block defined by index	• Any one of the **N** blocks within the set defined by index

Block Identification: How is a block found if it is in the cache?

Direct Mapped:	N-way Set Associative:
• Tag match with only one block	• Tag match for all the blocks within the set

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:	N-way Set-Associative:
• No Choice	• Based on replacement policy

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate