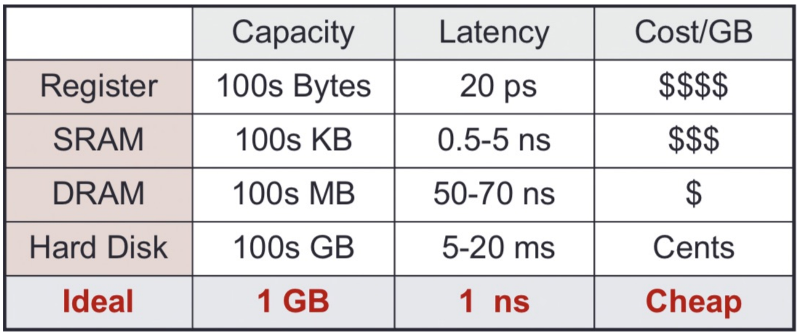
Cache

- SRAM
  - 6 transistors per memory cell.
  - Low density
  - Fast access latency of 0.5 - 5 ns
  - More costly.
  - used in flip-flops, cache.

- DRAM
  - 1 transitor per memory cell
  - High density
  - Slow access latency of 50 - 70 ns
  - less costly
  - used in main memory.



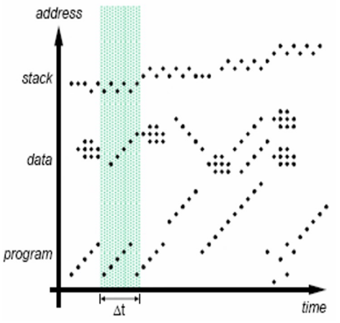
	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	$$$$
SRAM	100s KB	0.5-5 ns	$$$
DRAM	100s MB	50-70 ns	$
Hard Disk	100s GB	5-20 ms	Cents
Ideal	**1 GB**	**1 ns**	**Cheap**

└ Big and Fast

Cache

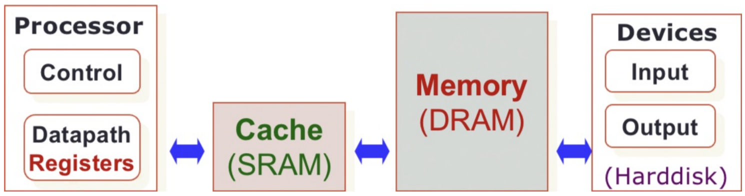
- Idea: keep frequently and recently used data in smaller but faster memory.
- Priciple of Locality: Program accesses only a small portion of the memory address space within a small time interval.

- Temporal Locality (Time)
  - If an item is referenced, it will tend to be referenced again soon.
- Spatial Locality (Space)
  - If an item is referenced, nearby items will tend to be referenced soon.
- Different locality for instructions and data.



Capture the working set and keep it in the memory closest to CPU.

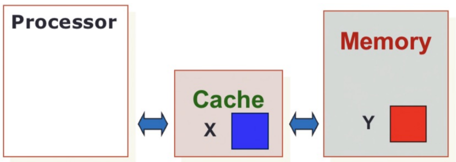
Cache Memory



Small, but fast SRAM near CPU

Hardware managed, transparent to programmer.

make slow main memory appear faster.



- **Hit**: Data is in cache (e.g., **X**)
  - Hit rate: Fraction of memory accesses that hit
  - Hit time: Time to access cache (very fast)
- **Miss**: Data is not in cache (e.g., **Y**)
  - Miss rate = 1 – Hit rate (opposite of Hit rate)
  - Miss penalty: Time to replace cache block + hit time
- Hit time < Miss penalty

Average Access Time

= Hit rate x Hit Time + (1-Hit rate) x Miss penalty

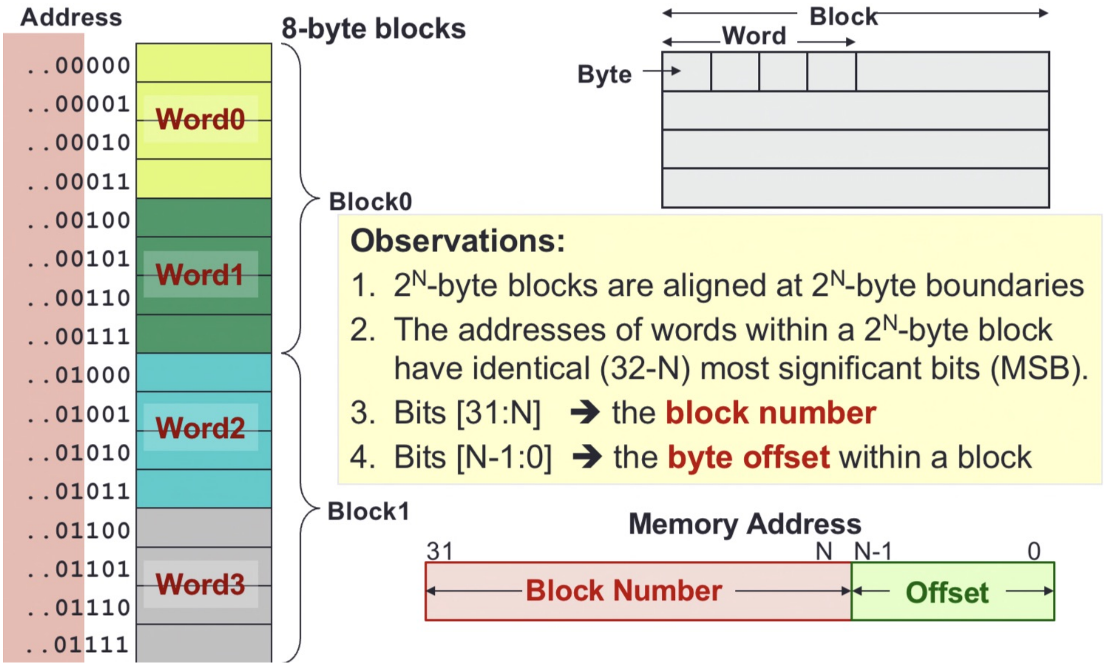
Example:

- 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.
$1 = 0.8h + (1 - h) \times (10 + 0.8)$
$= 0.8h + 10.8 - 10.8h$
$10h = 9.8 \rightarrow h = 0.98$
Hence we need a hit rate of **98%**.

Memory to cache mapping

- Cache Block/Line – unit of transfer between memory and cache.
  – typically one or more words
    - 16-byte block ≈ 4-word block.
    - 32-byte block ≈ 8-word block.



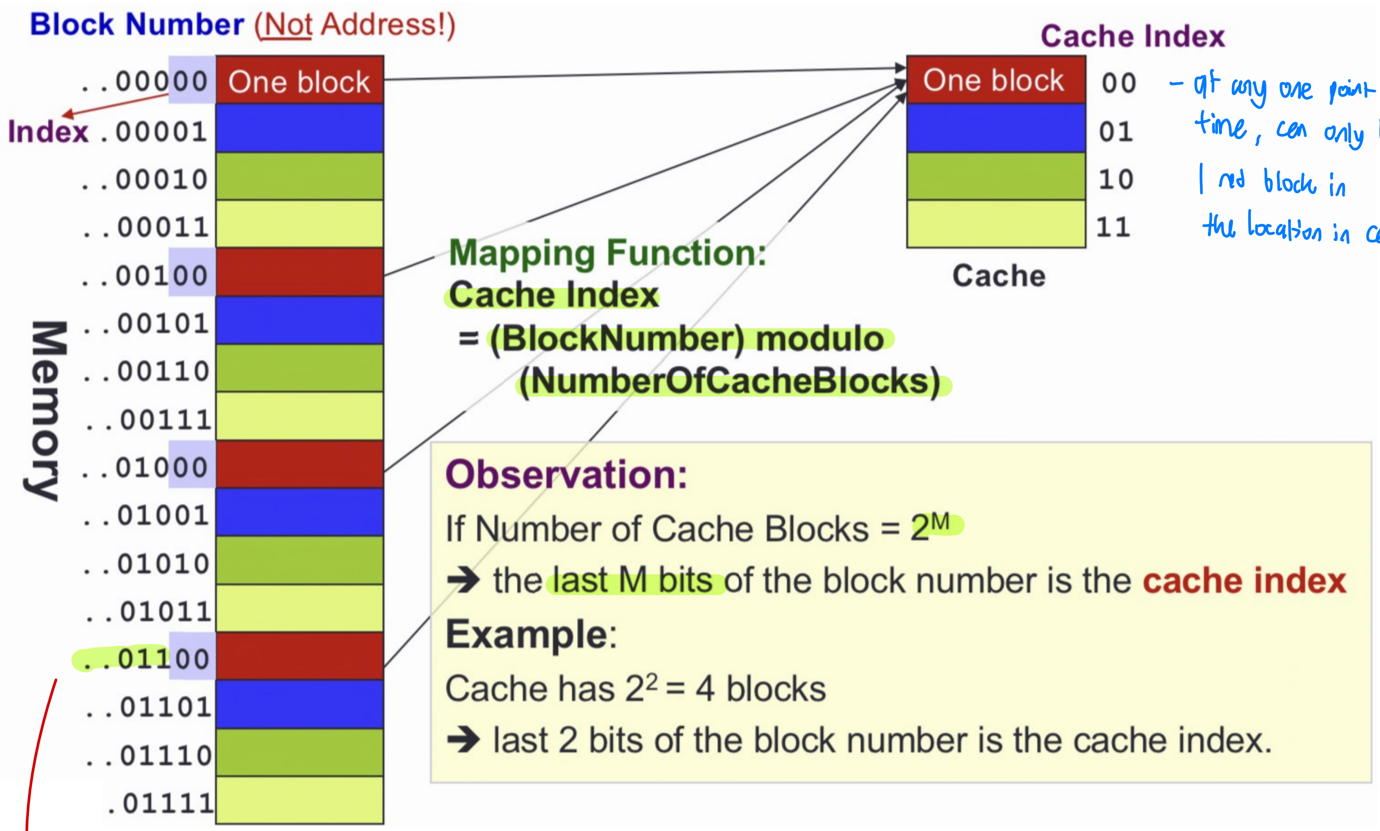
Observations:

1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. The addresses of words within a 2^N-byte block have identical (32-N) most significant bits (MSB).
3. Bits [31:N] ➔ the **block number**
4. Bits [N-1:0] ➔ the **byte offset** within a block

32-N MSB. "Block number"

byte-offset within the block.

· Direct Mapped Cache.



Mapping Function:
Cache Index = (BlockNumber) modulo (NumberOfCacheBlocks)

Observation:

If Number of Cache Blocks = 2^M

➔ the last M bits of the block number is the **cache index**

Example:

Cache has 2^2 = 4 blocks

➔ last 2 bits of the block number is the cache index.

- at any one point in time, can only have 1 red block in the location in cache

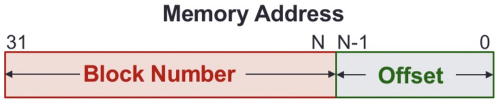
2 LSBs → 4 slots in cache.

Multiple memory blocks can map to the same cache block

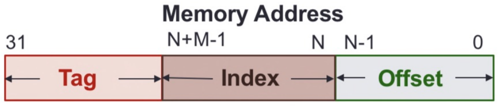
→ same cache index.

However, they have unique tag number.

Tag = Block number / Number of Cache Block.

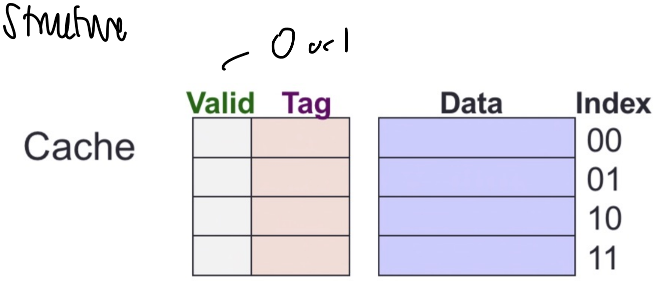


Cache Block size = 2^N bytes



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

Cache Structure



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

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

When is there a cache hit?

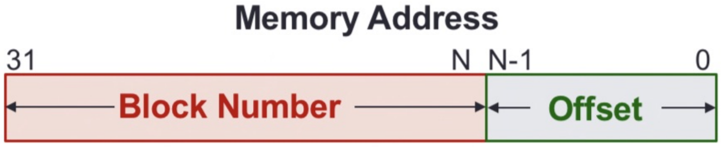
(Valid[index] = TRUE) **AND** is the slot filled ?

(Tag[index] = Tag[memory address])

Eg.

Memory 4GB

1 Block = 16 bytes



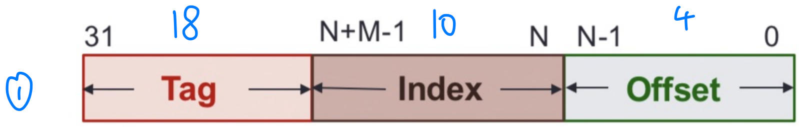
Offset, N = 4 bits

Block Number = 32 – 4 = **28 bits**

Check: Number of Blocks = $\mathbf{2^{28}}$

Cache 16KB

1 Block = 16 bytes



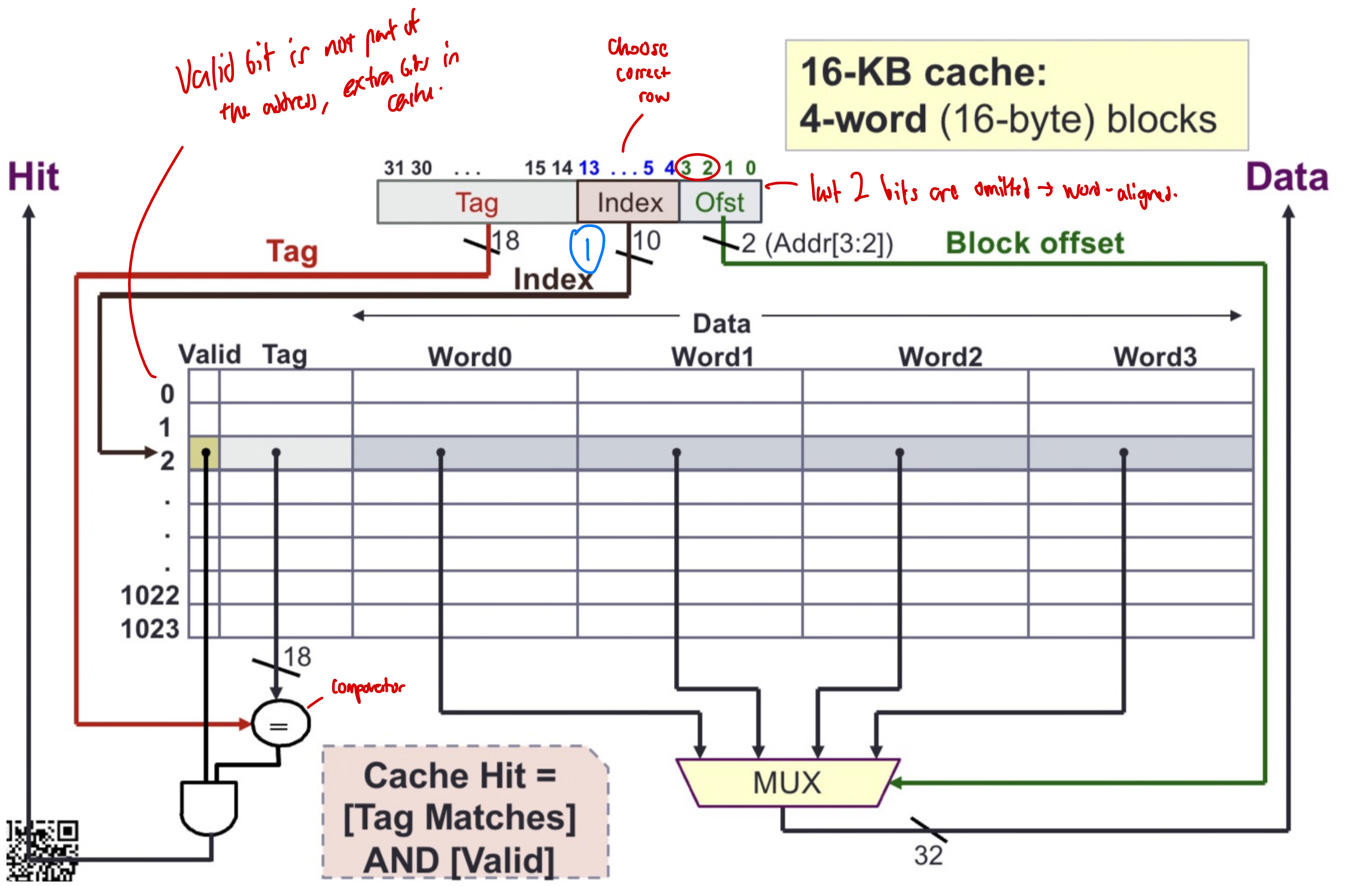
Number of Cache Blocks

= 16KB / 16bytes = 1024 = $\mathbf{2^{10}}$

Cache Index, M = 10bits

Cache Tag = 32 – 10 – 4 = **18 bits**

16-KB cache:
4-word (16-byte) blocks



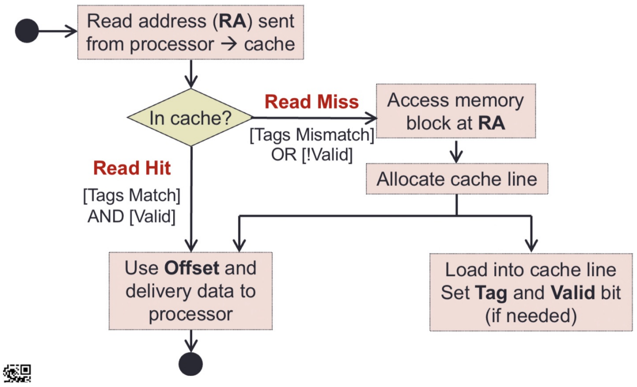
Valid bit is not part of the address, extra bits in cache.

Choose correct row

last 2 bits are omitted → word-aligned.

Comparator

Cache Hit = [Tag Matches] AND [Valid]


Read address (**RA**) sent from processor → cache

In cache?

Read Miss [Tags Mismatch] OR [!Valid]

Access memory block at **RA**

Allocate cache line

Read Hit [Tags Match] AND [Valid]

Use **Offset** and delivery data to processor

Load into cache line Set **Tag** and **Valid** bit (if needed)

* Initially cache is empty

Steps: (Load)

1. Check Cache Block at specified index
2. Data in specified block is invalid (Cold / Compulsory Miss)
3. Load 16 bytes from memory; Set Tag and Valid bit.
   ↳ To populate the "words" slot. ↳ To 1.
4. Return (offset 2MSB) Word to Register

(2nd time)
1. Check Cache Block at specified index
2. [Cache Block is Valid] AND [Tags match] → Cache hit!
3. Look at offset to get the specified Word. [Spatial Locality].

(Overriding)
1. Check Cache Block at specified index
2. Cache block is Valid but Tags mismatched [Cold miss]
3. Replace block 1 with new data; Set Tag.
4. Check offset, return specified Word to Register.

hit if block in mem == block in cache.

Cache Misses

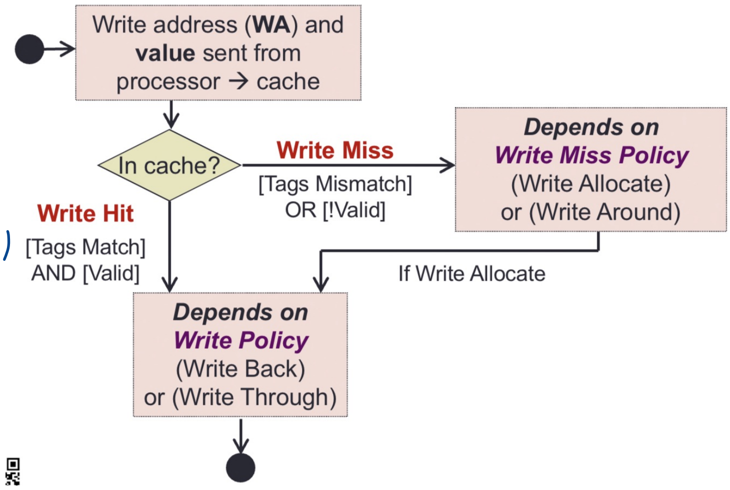
1. Compulsory / Cold Misses
   - On first access to a block; the block must be brought into the cache.
   - Aka Cold Start misses or first reference misses.
2. Conflict Misses
   - Occur in direct mapped cache / set associative cache, when several blocks are mapped to the same block/set. (Same other blocks)
   - Aka Collision misses / interference misses.
3. Capacity Misses
   - Occur when blocks are discarded from cache as cache cannot contain all blocks needed.

Handling Cache Misses

- On Read Miss:
  - Data located 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 the cache
  - Write to main memory depends on write policy. (WB/WT)
- Write Miss option 2: Write around
  - Do not load the block to cache
  - Write directly to main memory only.

Steps: (Write)

1. Check index, offset
2. Check if Cache is Valid and Tags match → Cache hit!
3. Replace chosen offset with new Word. (Changed data in cache only)

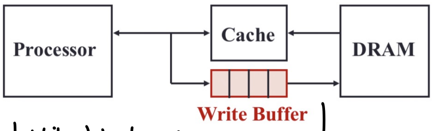


Write address (WA) and value sent from processor → cache
In cache?
Write Miss
[Tags Mismatch] OR [!Valid]
Depends on Write Miss Policy (Write Allocate) or (Write Around)
If Write Allocate
Write Hit
[Tags Match] AND [Valid]
Depends on Write Policy (Write Back) or (Write Through)

⇓

Write Policy

- Cache and main memory are inconsistent.
- Solution 1: Write-through Cache
  - Write data both to cache and main memory.
    - └ Slow - will only operate at the speed of main memory.
    - └ Buffer btw Cache and main memory (program could carry on, no need wait)



└ Writes data to cache and write buffer.

- Memory Controller: write contents of buffer to memory.

- Solution 2: Write-back cache
  - only write to cache
  - Write to main memory only when cache block is replaced (evicted).
  - More complicated to implement, wasteful if write back every evicted cache block.
    - └ Add an additional bit (Dirty bit) to each cache block — indicate if data is changed by write operation.
      - Write operation will change dirty bit to 1
        - Only cache block is updated, no write to memory.
      - When a cache block is replaced:
        - Only write back to memory if dirty bit is 1.