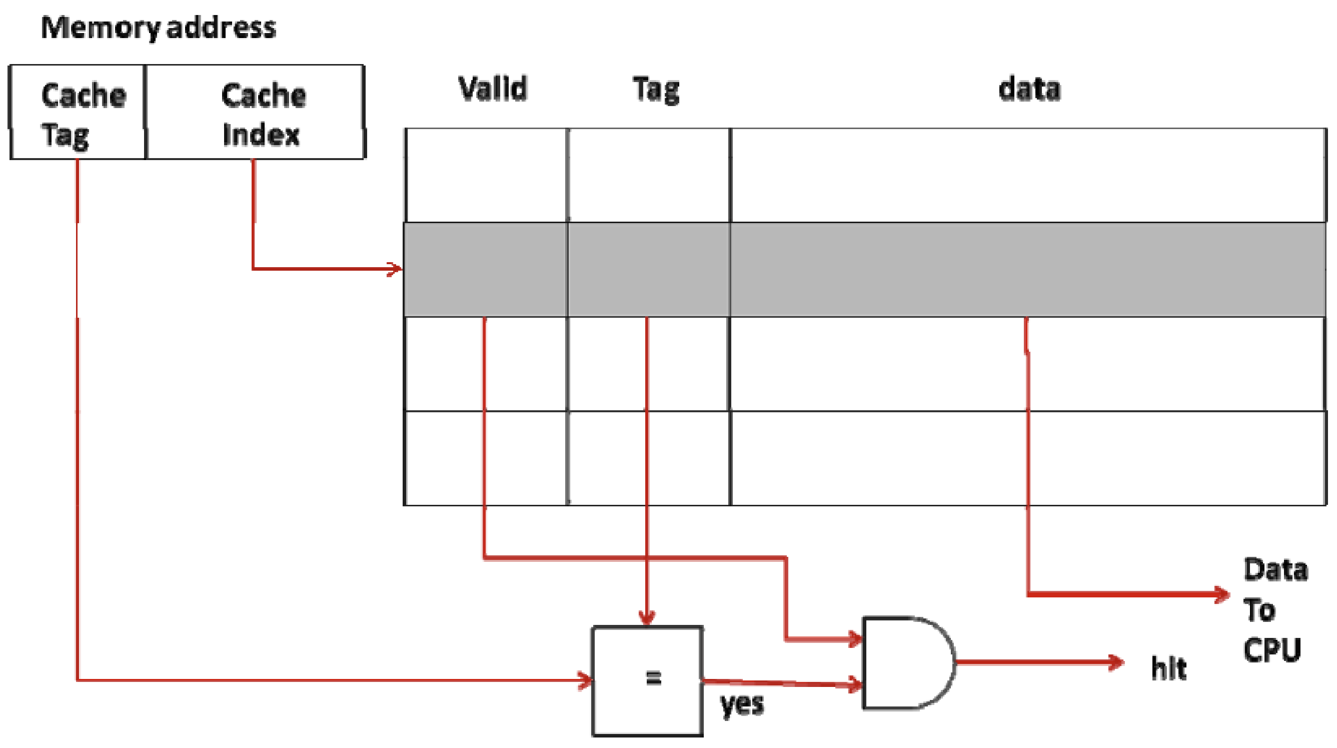
Chapter 9: Caching

Hardware for Direct Mapped Cache



The **index part** of the memory address picks out a unique entry in the cache (the shaded part in the diagram above)

The **comparator** compares the **tag field** of this entry against the tag part of the memory address

If there is a match **and** the entry is valid, then it signals a hit

- Cache supplies the data field of the selected entry on the *cache line* (a.k.a. *cache block* and *cache entry*) to the CPU

The *valid bits*, and *tag fields*, called *meta-data*, are for managing the actual data contained in the cache, and represent a *space overhead*

TODO

Example

Let us consider the design of a direct-mapped cache for a realistic memory system

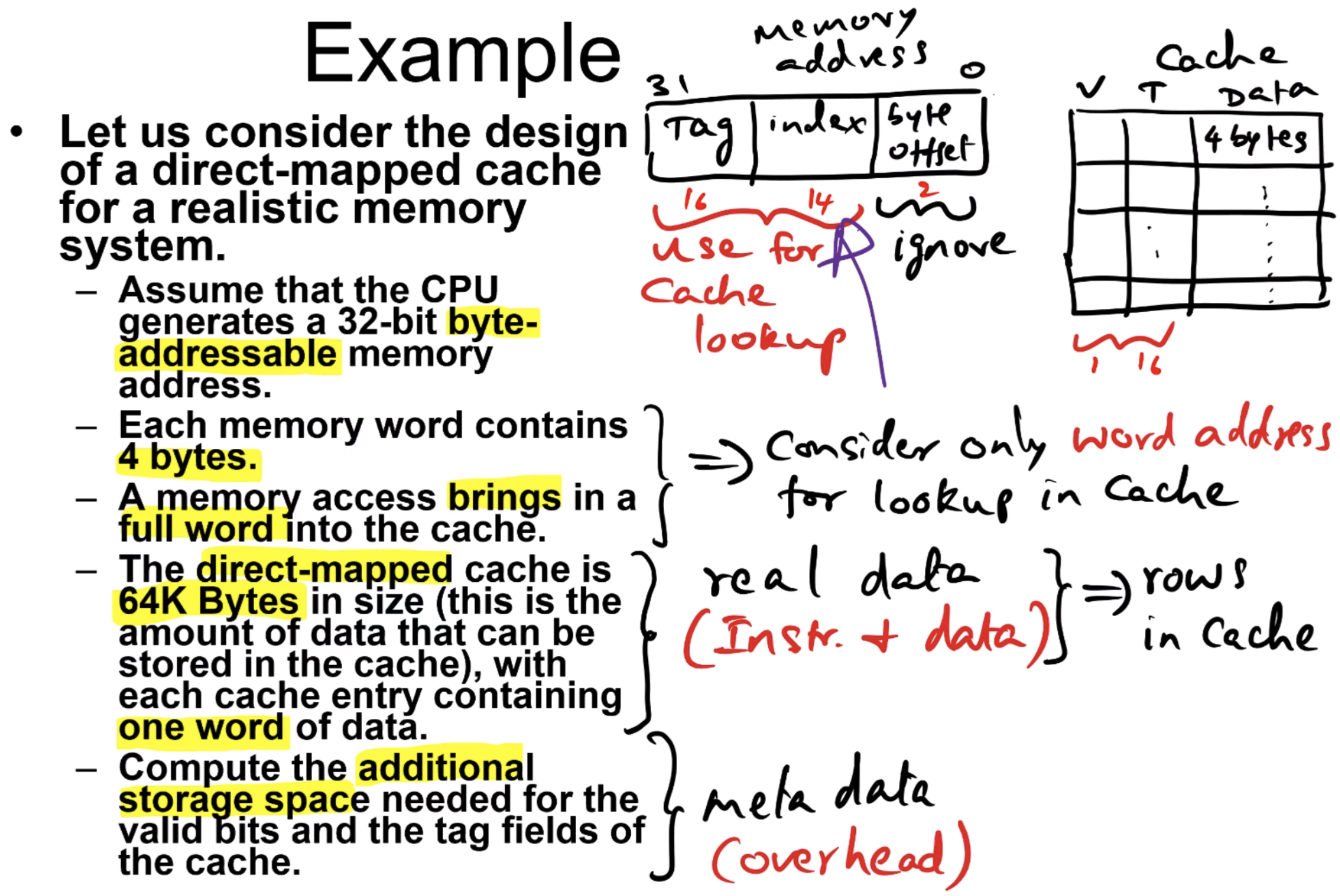
- Assume that the CPU generates a 32-bit byte-addressable memory address

- Each memory word contains 4 bytes
- A memory access brings in a full word into the cache
- The direct-mapped cache is 64K bytes in size (amount of data that can be stored in the cache), with each cache entry containing one word of data
    - (real data = instr + data) => rows in cache
- Compute the additional storage space needed for the valid bits and the tag fields of the cache (a.k.a. the meta data or overhead)

$64K = 2^{16}$

- There are 16K entries which is 14 bits for the index

There are 64K Bytes and each row has 4 bytes, so $64 \div 4 = 16$



Example

- **Let us consider the design of a direct-mapped cache for a realistic memory system.**
  - **Assume that the CPU generates a 32-bit byte-addressable memory address.**
  - **Each memory word contains 4 bytes.**
  - **A memory access brings in a full word into the cache.**
  - **The direct-mapped cache is 64K Bytes in size (this is the amount of data that can be stored in the cache), with each cache entry containing one word of data.**
  - **Compute the additional storage space needed for the valid bits and the tag fields of the cache.**

=> Consider only word address for lookup in cache

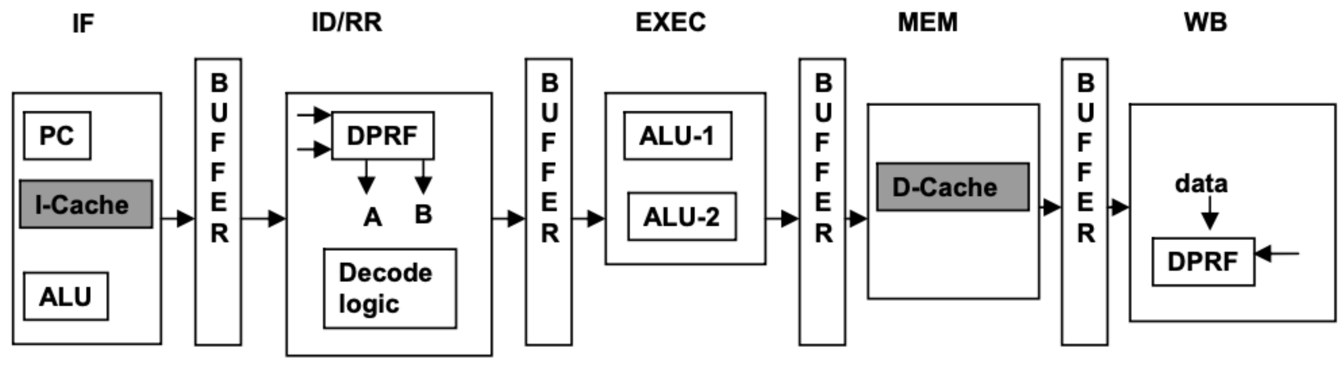
real data (Instr. + data) => rows in Cache

meta data (overhead)

Turningpoint
A direct mapped cache **has a one-to-one mapping between a memory location and a cache location**

In a direct-mapped cache with a t-bit tag **there is a 1-bit tag comparator for the entire cache**

Repercussion on Pipelined Processor



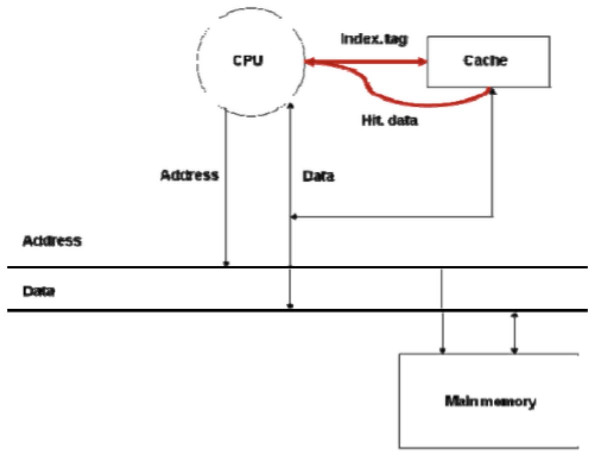
Notice that we have replaced the memories, I-MEM and D-MEM in the IF and MEM stages, by caches I-Cache and D-Cache, respectively

- IF and MEM stages to have comparable cycle times to the other stages of the pipeline

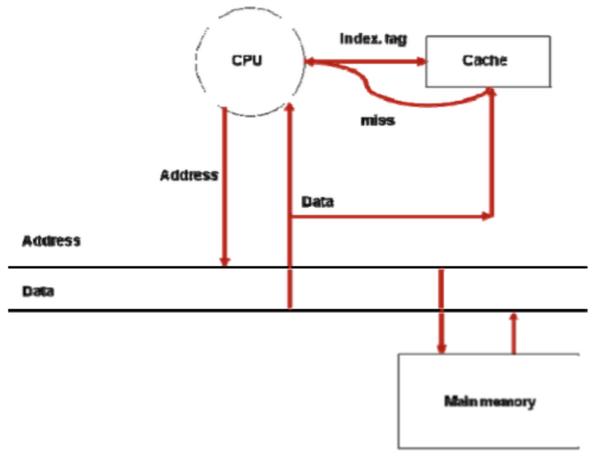
What happens if we have a miss?

1. **Miss in the IF stage**: IF stage sends the reference to the memory to retrieve the instruction. As we know, the memory access time may be several 10's of CPU cycles. Until the instruction arrives from the memory, the IF stage sends NOPs (bubbles) to the next stage
2. **Miss in the MEM stage**: Misses in the D-Cache are relevant only for ***memory reference instructions (load/store)***. Similar to the IF stage, a miss in the MEM stage results in NOPs to the WB stage until the memory reference completes. It also ***freezes*** the preceding stages from advancing past the instructions they are currently working on

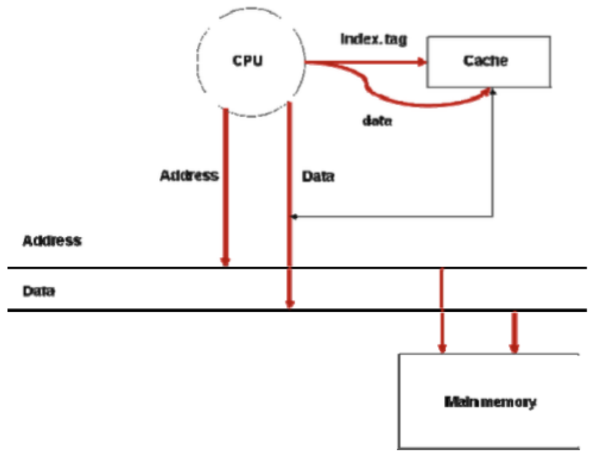
Cache Read/Write Algorithms



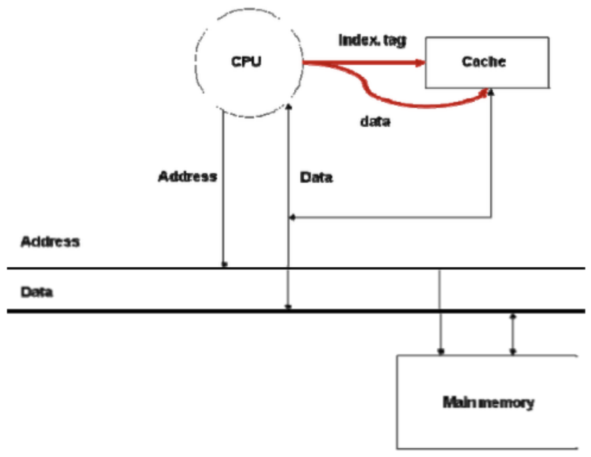
(a) Read Hit



(b) Read miss



(c) Write-through



(d) Write-back

Read Access to the Cache from CPU

CPU needs to read from cache in either the instruction fetch in IF stage or operand fetch in response to a load in MEM stage

1. **Step 1:** CPU sends the index part of the memory address to the cache. Cache does a lookup, and if successful (hit), it supplies data to CPU. If miss, CPU sends the address on the memory bus to the main memory. In principle, all of these actions happen in the same cycle
2. **Step 2:** Upon sending the address to the memory, the CPU sends NOPs down to the subsequent stage until it receives the data from the memory. ***Read stall*** is defined as the number of processor clock cycles wasted to service a read-miss. This could take several CPU cycles depending on the memory speed. The cache allocates a cache block to receive the memory block. Eventually, the main memory delivers the data to the CPU and simultaneously updates the allocated cache block with the data. The cache modifies the tag field of this cache entry appropriately and sets the valid bit

Write Access to The Cache From CPU

This could happen for storing a memory operand in the MEM stage. There are a couple of choices for handling processor write access to the cache: write through and write back.

Write Through Policy

Update the cache and the main memory on each CPU write operation

1. **Step 1**: On every write (store instruction in LC-2200), the CPU simply writes to the cache. No need to check the valid bit or the cache tag – the cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
2. **Step 2**: Simultaneously, the CPU sends the address and data to the main memory. This of course is problematic in terms of performance since memory access takes several CPU cycles to complete. To alleviate this performance bottleneck, it is customary to include a write-buffer in the datapath between the CPU and the memory bus. The write-buffer is a small hardware store (similar to a register file) to smoothen out the speed disparity between the CPU and memory. As far as the CPU is concerned, the write operation is complete as soon as it places the address and data in the write buffer. Thus, this action also happens in the MEM stage of the pipeline without stalling the pipeline.
3. **Step 3**: The write-buffer completes the write to the main memory independent of the CPU. Note that, if the write buffer is full at the time the processor attempts to write to it, then the pipeline will stall until one of the entries from the write buffer has been sent to the memory. Write stall is defined as the number of processor clock cycles wasted due to a write operation (regardless of hit or a miss in the cache).

Effect of memory stalls due to cache misses on pipeline performance

Updated Equations

Execution time $=$ (Number of instructions executed $\cdot$ (CPIAvg $+$ Memory $-$ stallsAvg)) $\cdot$ clock cycle time

Effective CPI $=$ CPIAvg $+$ Memory $-$ stallsAvg

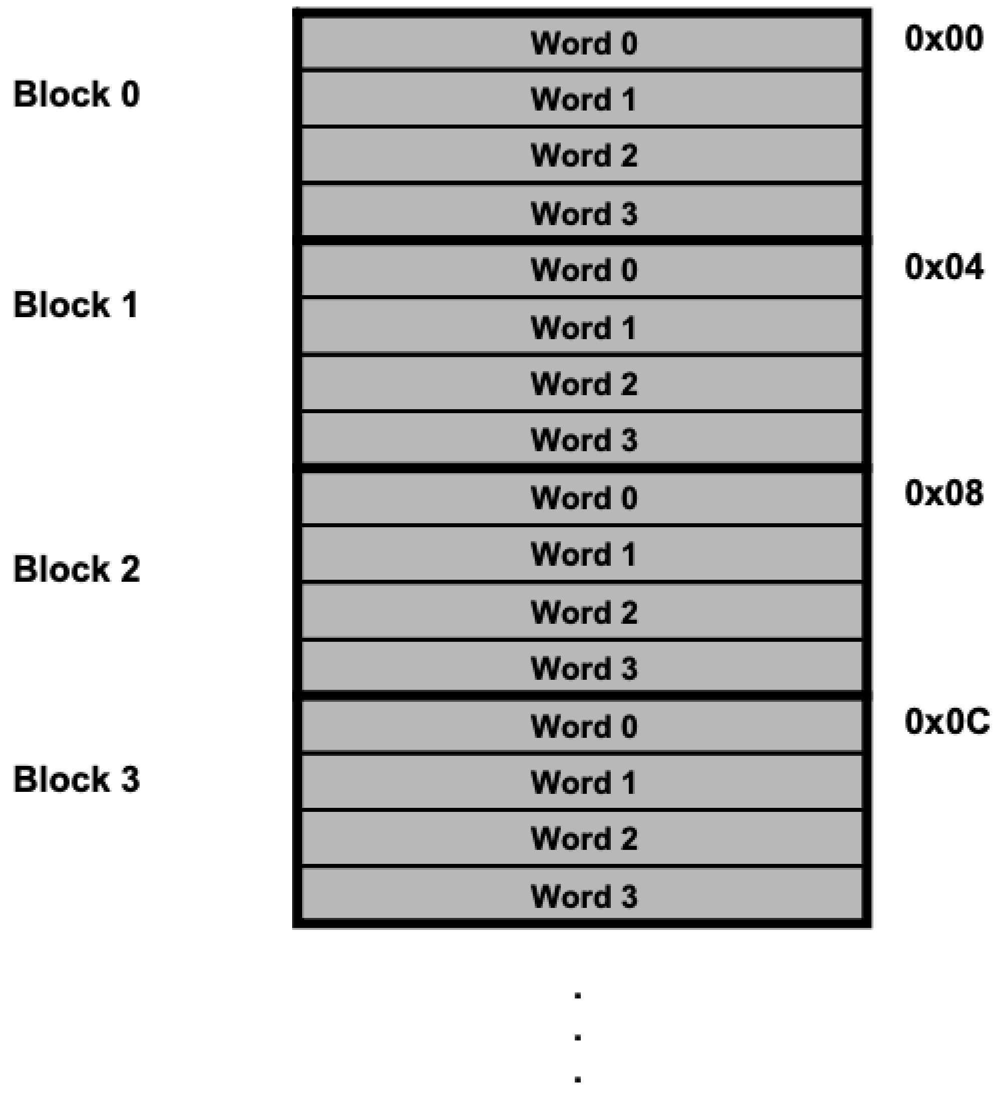
Total memory stalls $= \text{Number of instructions} \cdot \text{Memory} - stalls_{Avg}$

$\text{Memory} - stalls_{Avg} = \text{misses per } instruction_{Avg} * \text{miss} - penalty_{Avg}$

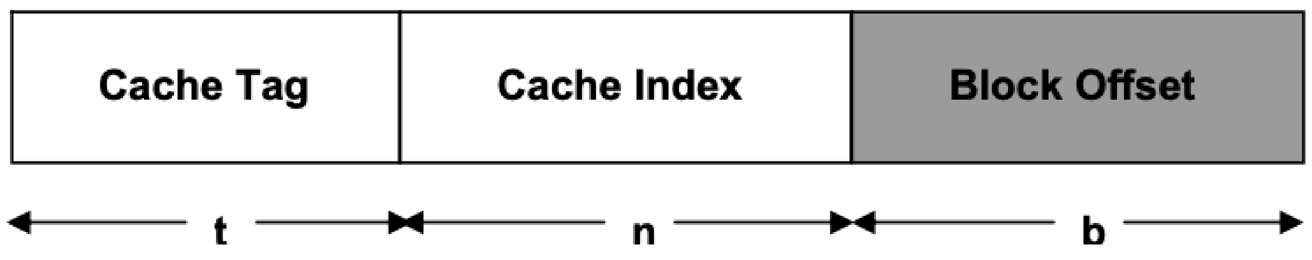
Exploiting Spatial Locality to Improve Performance

The basic idea is to bring adjacent memory locations into the cache upon a miss for a memory location i

- To exploit spatial locality in the cache design, we decouple the *unit of memory access* by an instruction from the *unit of memory transfer* between the memory and the cache
- We refer to the unit of transfer between the cache and the memory as the ***block size***
- Upon a miss, the cache brings an entire *block of block size* bytes that contains the missing memory reference



In the example above, if we miss on location 0x01,
the cache brings in all 4 memory words in Block 0



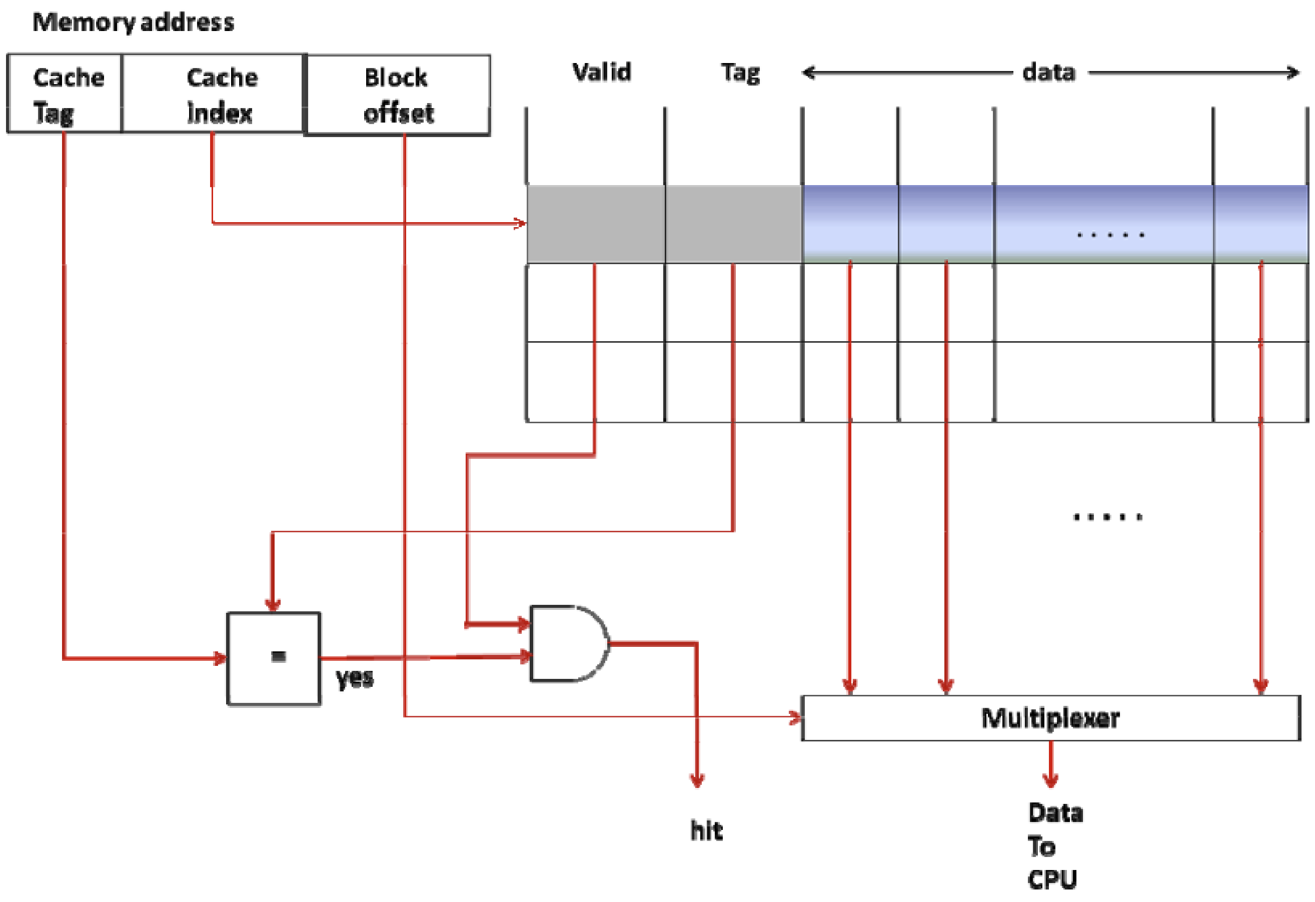
Block Offset is the # of bits necessary to enumerate the set of contiguous memory locations contained in a cache block

if the block size is 64 bytes, then the block offset is 6-bits

- $2^{16} = 65,536$

If L represents the number of lines in a direct-mapped cache of size S and B bytes per block, with b representing the number of least significant bits of the memory field, t representing the number of most significant bits of the memory field, n representing the middle bits, and a representing the total number of bits, then

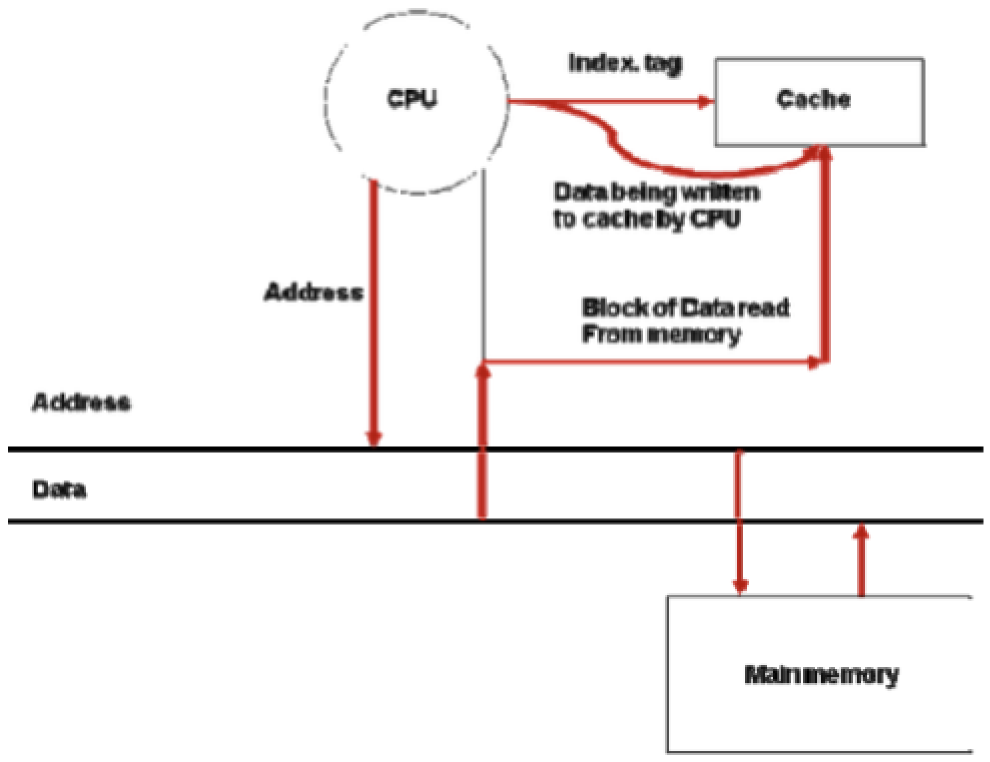
$$b = log_2 B$$
$$L = S \div B$$
$$n = log_2 L$$
$$t = a - (b + n)$$



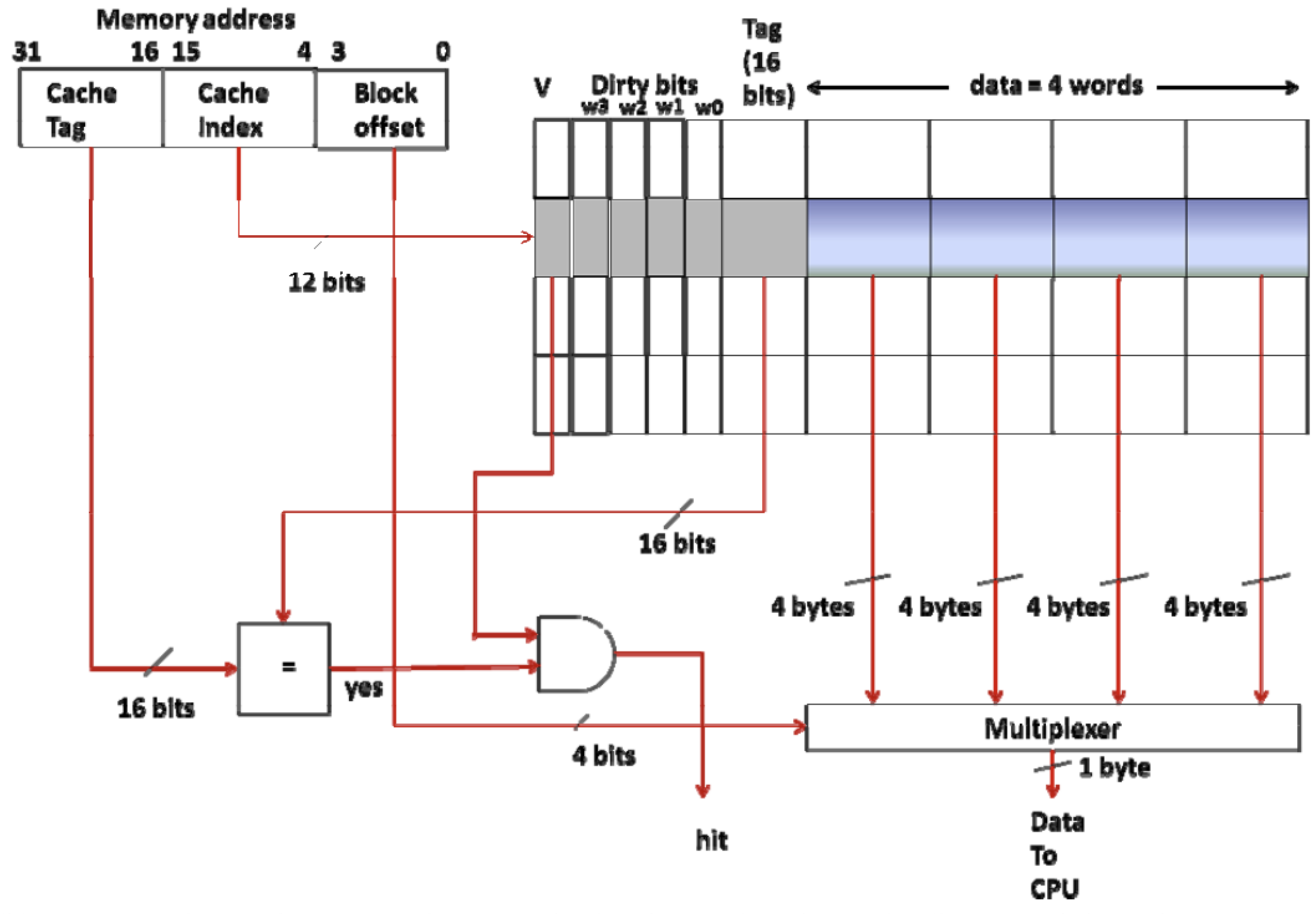
1. **Lookup**: The index for cache lookup comes from the middle part of the memory address. The entry contains an entire block (if it is a hit as determined by the cache tag in the address and the tag stored in the specific entry). The least significant b bits of the address specify the specific word (or byte) within that block requested by the

processor. A multiplexer selects the specific word (or byte) from the block using these b bits and sends it to the CPU.

2. **Read**: Cache brings out the entire block corresponding to the cache index. If the tag comparison results in a hit, then the multiplexer selects the specific word (or byte) within the block and sends it to the CPU. If miss then the CPU initiates a block transfer from the memory
3. **Write**: Modify the write algorithm since there is only 1 valid bit for the entire cache line. Similar to the read-miss, the CPU initiates a block transfer from the memory upon a write-miss


CPU
Index. tag
Cache
Data being written to cache by CPU
Address
Block of Data read From memory
Address
Data
Main memory

Example

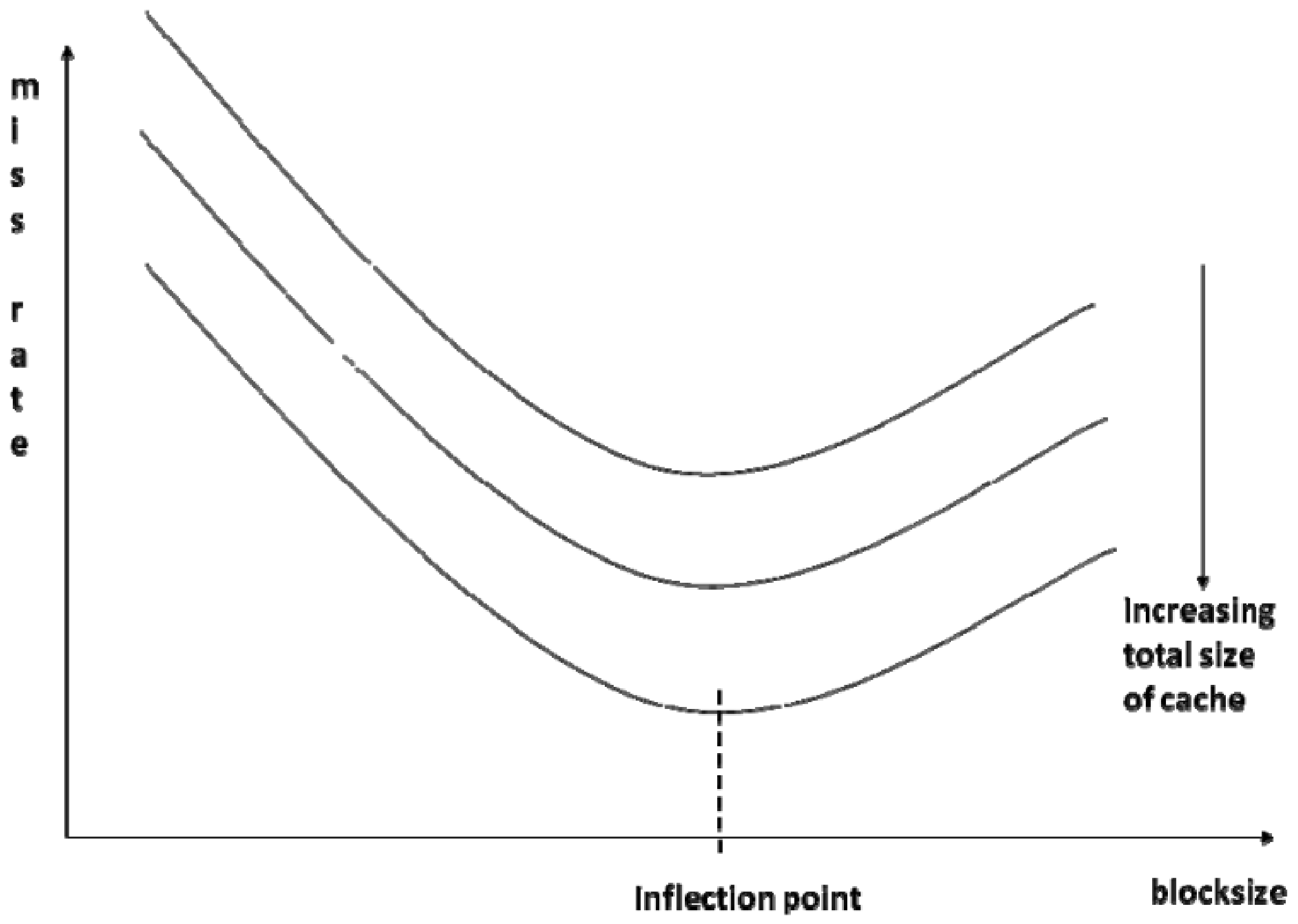


TODO

Performance Implications of Increased Blocksize

For a given total size of the cache, we would expect the miss-rate to decrease with increasing block size

- Will miss rate keep decreasing forever? ***NO***
- Will the overall performance of the processor go up as we increase the block size? ***DEPENDS***



Expected behavior of miss-rate as a function of blocksize

Beyond the inflection point, the processor starts incurring more memory stalls thus degrading the performance. However, the downturn in processor performance may start happening much sooner than the inflection point

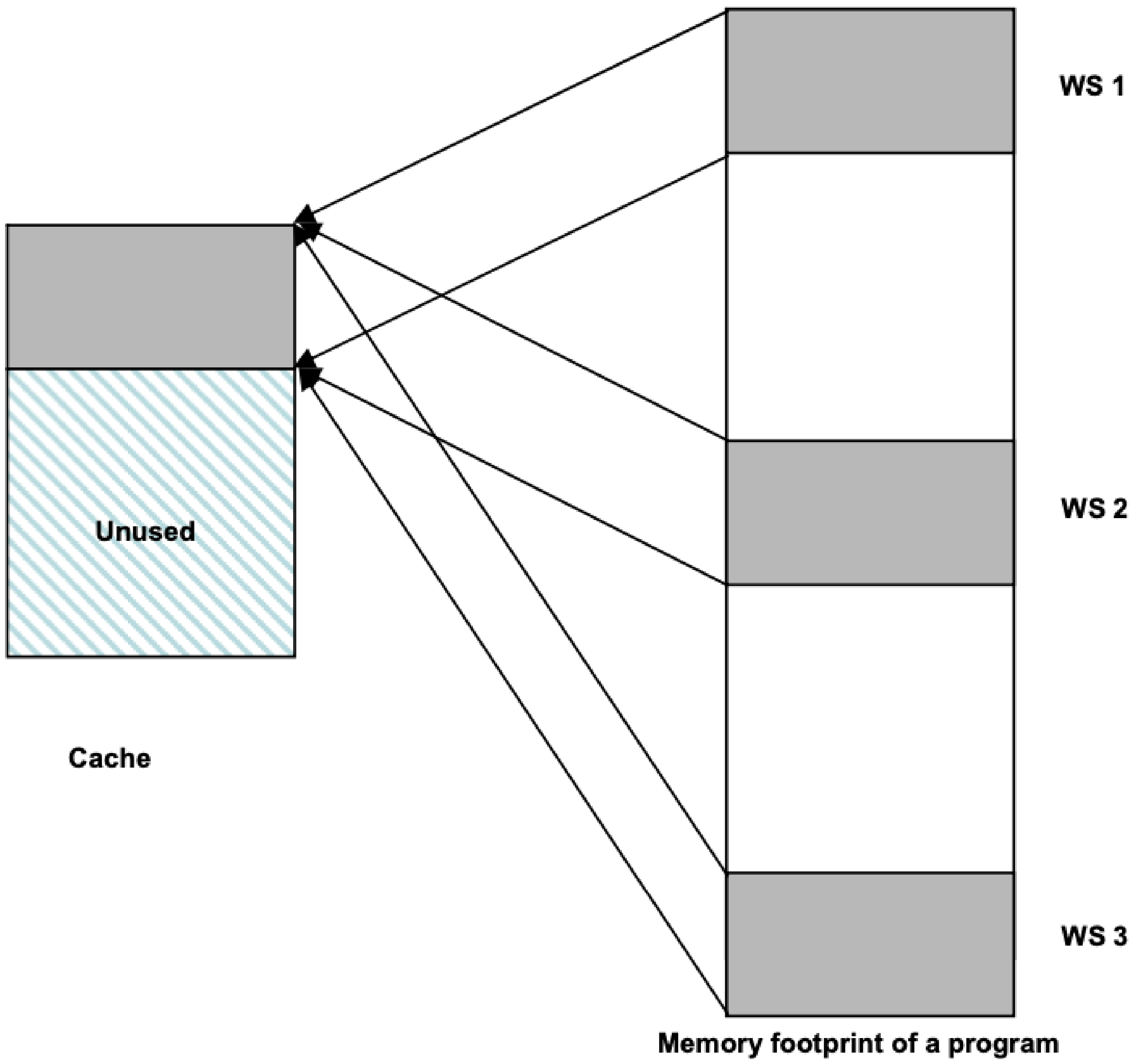
- Increased block size could negatively interact with the miss penalty
- The larger the block size, the larger the time penalty for the transfer from the memory to the cache on misses, thus increasing the memory stalls

Choice of block size affects the balance between latency for a *single instruction* (that incurred the miss) and throughput for the program as a whole, by reducing the potential misses for other *later instructions* that might benefit from the exploitation of spatial locality

Flexible Placement

In a direct-mapped cache, there is a one-to-one mapping from a memory address to a cache index

- unable to place a new memory location in a currently unoccupied slot in the cache



Different working sets of a program occupying the same portion of a direct-mapped cache

The program frequently flips among the three working sets (WS1, WS2, and WS3)

- Due to the rigid mapping, the working sets displace one another from the cache resulting in poor performance
- We would want all three working sets of the program to reside in the cache so that there will be no more misses beyond the compulsory ones

Fully Associative Cache

No unique mapping from a memory block to a cache block

- A cache block can host any memory block

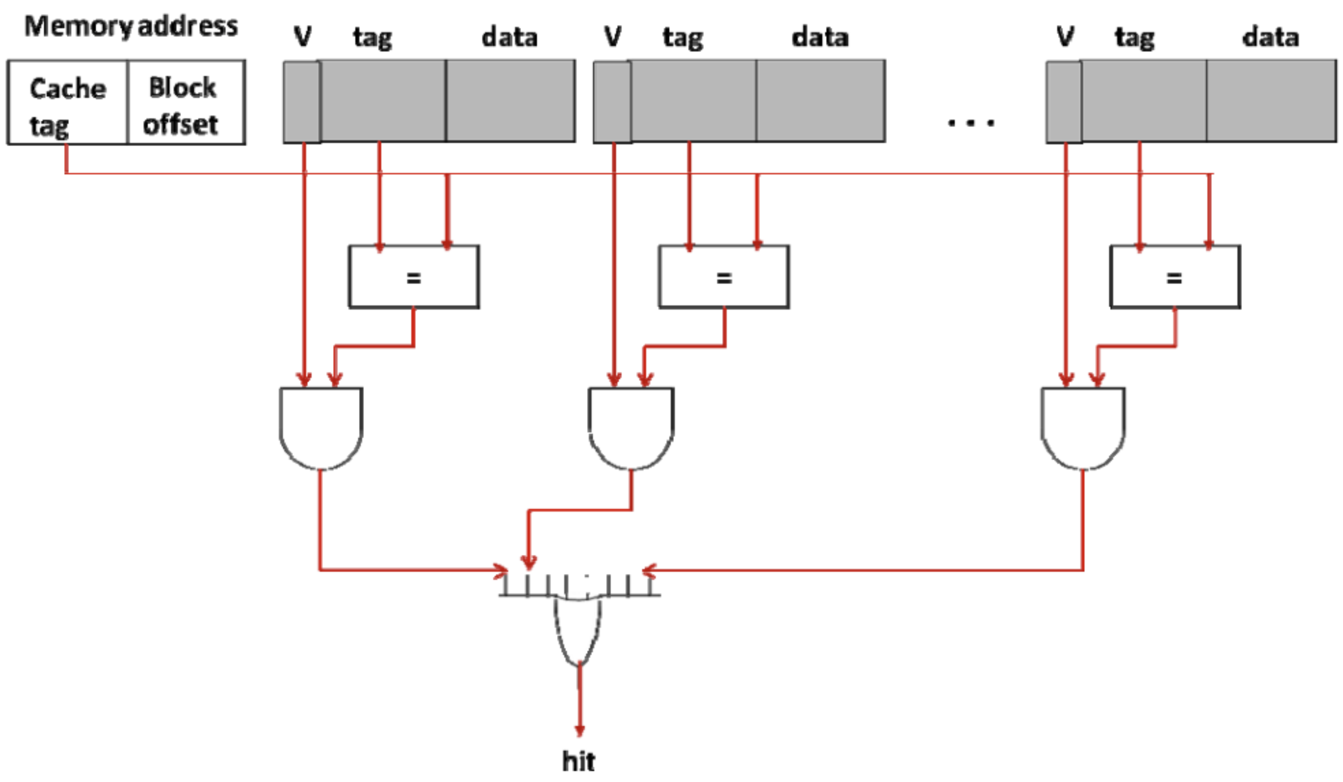
- ***Compulsory*** and ***capacity*** misses are the only kind of misses encountered with this organization

Cache Tag	**Block Offset**

Memory address interpretation for a fully associative cache

To perform a look up, the cache has to search through **all the entries** to see if there is a match between the cache tag in the memory address and the tags in any of the valid entries

- Search through sequentially? Too slow
- Add comparator for everything instead



Each block of the cache represents an independent cache. Tag matching has to happen in parallel for all the caches to determine hit or a miss

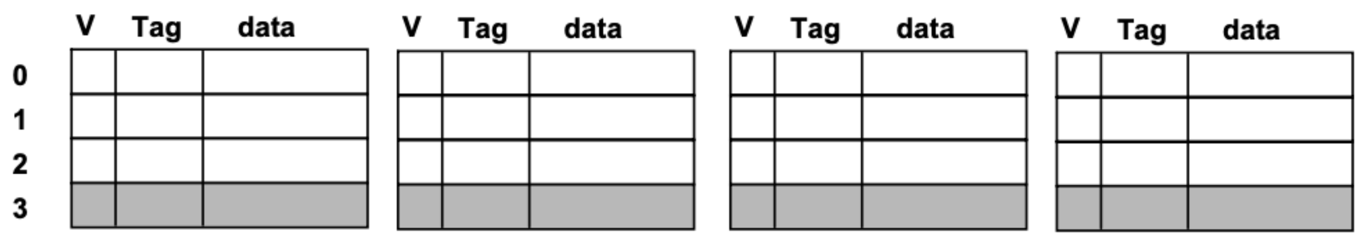
Turningpoint
In a fully associative cache with 64K bytes of data, 64 bytes per block and with a t-bit tag, **there are 1K t-bit tag comparators**

- 64K bytes of data / 64 bytes per block = 1K

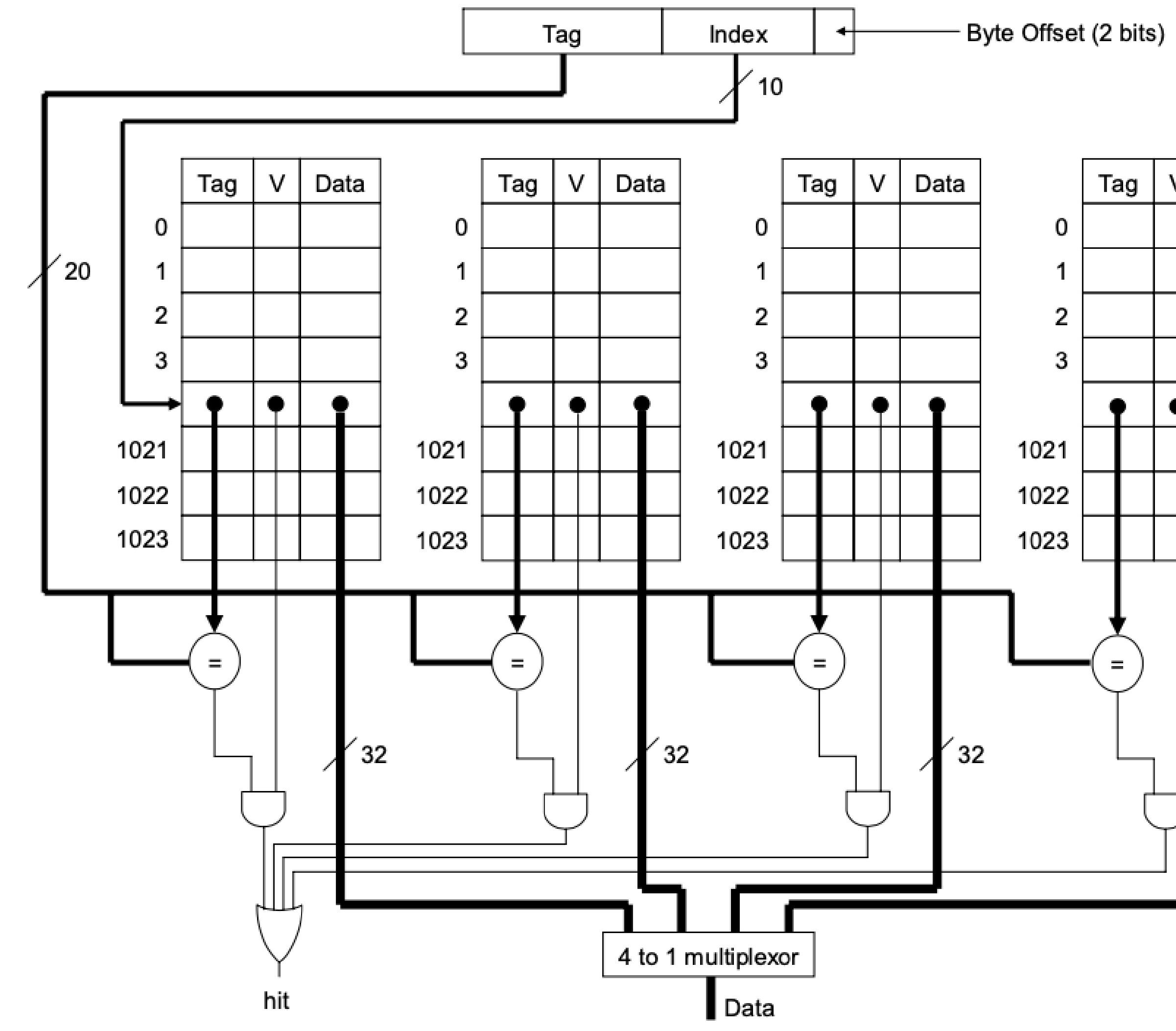
Set Associative Cache

Memory block can be associated with a set of cache blocks

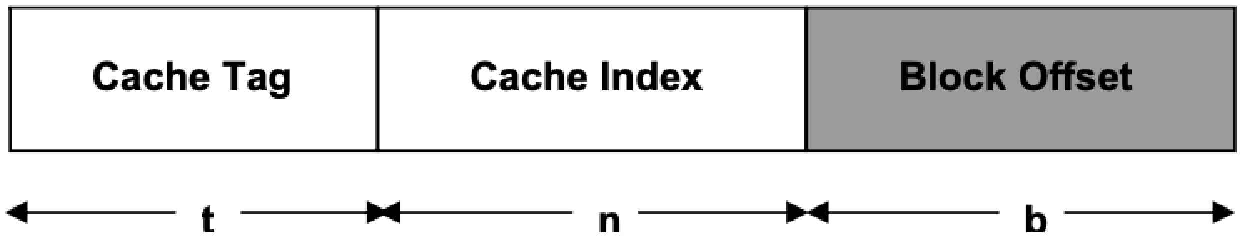
- Ex: a 4-way set associative cache gives a memory block one of four possible homes in the cache
- The ***degree of associativity*** is defined as the number of homes that a given memory block has in the cache



4-way associative cache



4-way set associative cache organization



Similar to direct-mapped cache

Turningpoint

In a 4-way set associative cache with 64K bytes of data, 64 bytes per block and with a t-bit tag, **there are four t-bit tag comparators**