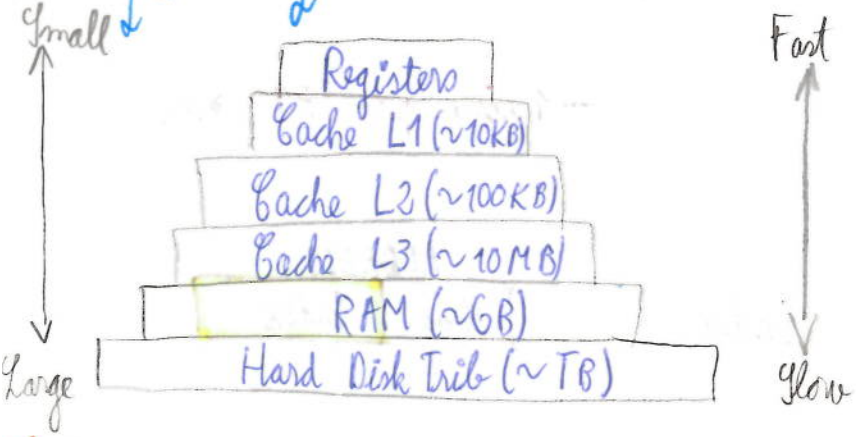
CACHING (fast, but small)

Processor Cache = small amount of very fast memory used as temporary store for frequently used memory locations (both instructions and data)

Role of CPU Cache

- CPU clock frequencies are much faster than memory access time
- CPU would be unable to run at full speed without the addition of the cache

Memory Hierarchy



L2 is { 16x bigger than L1
4x slower than L1
still 10x faster than RAM

L3 is { 2x faster than RAM

Cache → SRAM (needs 6 transistors per bit)

Main Memory → DRAM (needs one)

SSHD (hybrid) = contains a large hard disk drive and and SSD cache to improve performance of frequently accessed data

Caches exploit temporal and spatial locality.

Temporal locality

- memory locations accessed recently are likely to be accessed again within a short time span
- instructions in a loop

Spatial locality

- memory locations located close to recently accessed addresses are likely to be accessed as well
- data in arrays, sequence of instructions

Types of cache

1. Fully Associative Cache (small cache - expensive) - n lines per SET → just (1)
- random access, makes full use of the cache
- cache stores **full address** and **data**
- compare input address with all stored addresses (in parallel)
- if (address found) cache hit
  else cache miss [must go to main memory (slow), cache should be updated]

2. Direct Mapped Cache (based on a hash table) - 1 line per SET → (n)
- fully associative has a lot of expensive comparison operations, so we want to avoid this
- use standard RAM to implement cache functionality
- **address** divided into two parts: **tag** and **index**
- **index** used to address RAM directly
- **tag** is stored and **compared** with incoming tag, if same), data is read
- many addresses map to the same index

(using least significant bit as index exploits principle of spatial locality to minimize displacing recently used data

3. Set Associative Cache (uses a MUX) - way more expensive)
- is simply a small number of direct mapped caches operating in parallel
- replacement strategy more flexible
- hit rate gets better than single DMC
- if 4 DMCs, 4 addresses with the same index but a different tag can be stored

Cache Write Strategy → Cache Hit

- memory writes more complex than reads
- do an address comparison to check if it already exists in cache
- hit, update value in cache, but do not always need to write to memory, long write times

Write Through = every cache write also done to memory (super slow)

Write Through with buffer = buffered, so the write is non-blocking i.e. the processor does not have to wait, but writes may back up

Copy Back (Write) = write is only done to cache (marked as dirty). Only write back to memory when cache entry is replaced

Cache Write Strategy → Cache Miss (fastest strategy => Write Allocate & Copy Back)

Write Allocate = find location, reject if necessary, assign cache location and write value. Write through to RAM or rely on copyback

Write Around (write no-allocate) = just write to RAM, do not write to cache

Valid Bit - indicate meaningful data (it isn't just initialized to 0 or smth.)

→ How to improve "hit rate"

Exploit spatial locality - use a wider cache line (each entry will give more data than just a word). Use the lowest bits to select the word in the cache line. Data is transferred from RAM in bursts equal to the width of the line. Larger line size means less misses, but a line size which is too big means data may not be used or RAM access will take longer

Harvard architecture - separate caches for instructions and for data

Multiple level caches

- big caches are slow and L1 needs to run at processor speed. Put L2 between RAM and L1 → 16 times bigger but 4 x slower than L1. (10x faster than the RAM). L1D (data) and L1I (instructions) both share L2. L3 is large (8MB) its performance is only 2X RAM

Types of cache misses

1. Compulsory misses

- When we first start the computer, the cache is empty, so until the cache is populated, we're going to have a lot of misses

2. Capacity misses (we want to put ALL our elements in the cache)

- since the cache is limited in size, we can't contain **all** of the pages for a program, so some misses will occur

3. Conflict misses - multiple blocks compete for the same set (Not happening in fully associativ

- ~~in a direct mapped or set associative cache, there is competition between memory locations for places in the cache. If the cache was fully associative, then this wouldn't occur~~

Cache performance

- in order to fill a cache from empty, it takes $\frac{\text{cache size}}{\text{line size}}$ (= number of lines) memory accesses
- if we multiply this by the time it takes for a single memory access, then we can work out how long it will take to fill the cache

Cache consistency

- make sure the values stored in the CPU cache are consistent with those in the main memory
there are situations they can **disagree**. (if IO writes / reads directly to RAM)

Solutions

1) Non-cacheable

- make areas of memory that IO can access non-cachable or clear the cache before and after IO takes place

2) IO use data cache

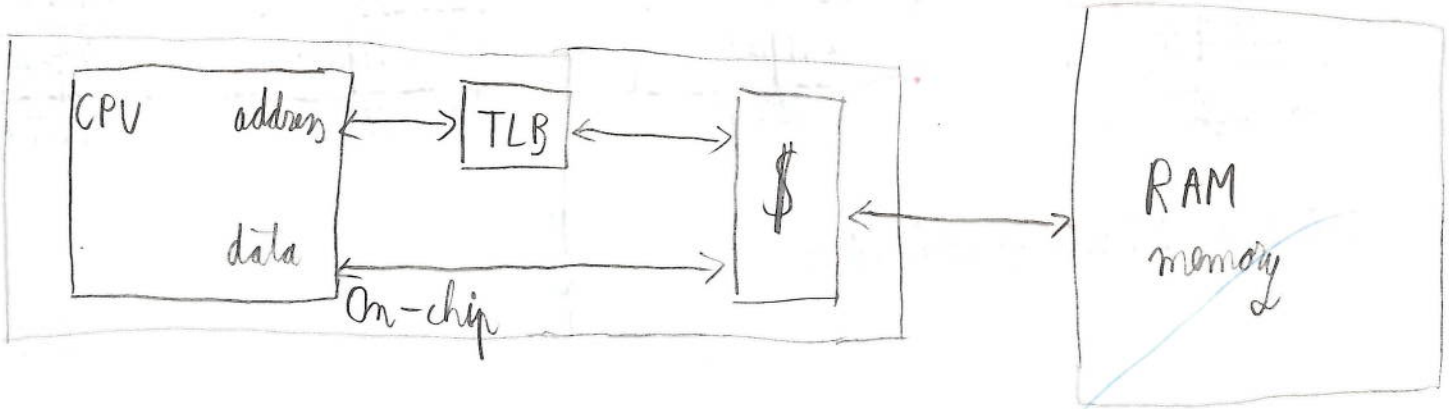
- IO writes first in CPU's L1D (data) cache before accessing the memory
- slows down the cache

3) Snoop on IO activity

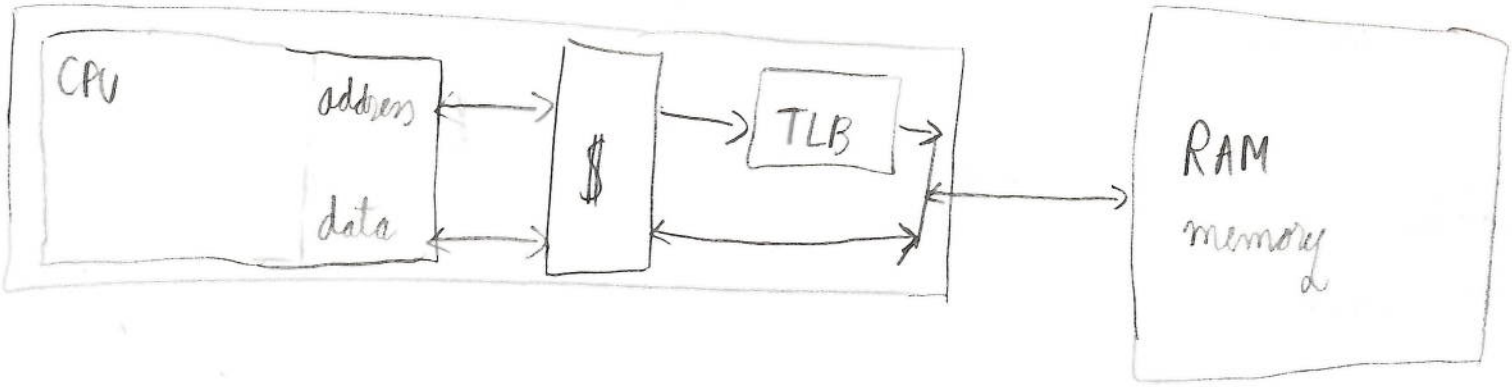
- we could have hardware logic in the cache that will look at the reads and writes to memory from IO and make sure the cache is consistent with memory for those addresses

Virtual Addresses (uses a TLB → Translation - Lookside Buffer)

CPU has **virtual addresses**
Memory has **physical addresses**
} Which address does the cache store? → we have to decide that



- slow, since they must pass through extra logic before hitting the cache



- makes snooping hard to implement along with other functional difficulties

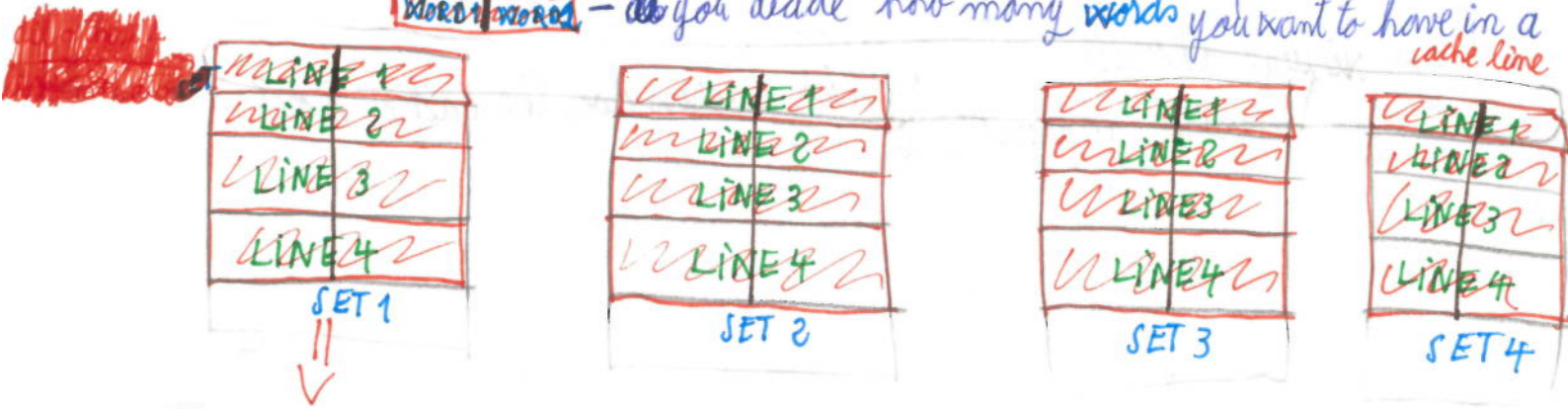
The answer is to have the TLB operate in parallel to the cache, since translation only affects the high order bits of the cache (the low order bits remains the same), and only the tag is changed by address translation.

Example of 4-way set associative (useful for your understanding - 2015 exam)

= 4 • Direct Mapped Caches per 1 SET

[redacted] is a cache line; multiple cache lines form a SET

WORD1 WORD2 - you decide how many words you want to have in a cache line



This is a SET with 4 cache lines

You don't know how many SETS you have. This is just an example. If you know how many sets you have, if you know the cache size and [redacted] set size.

number of SETS = cache size / set size

SET SIZE = number of lines • line size