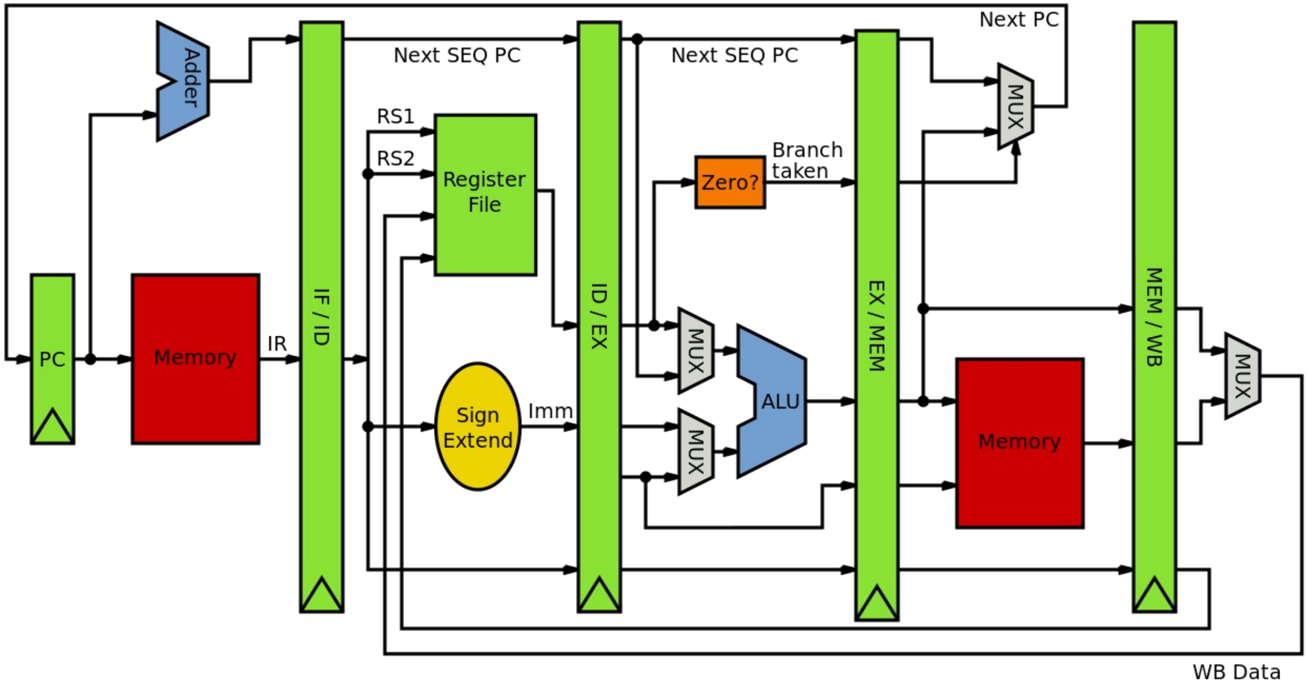
Computer Memory Organization

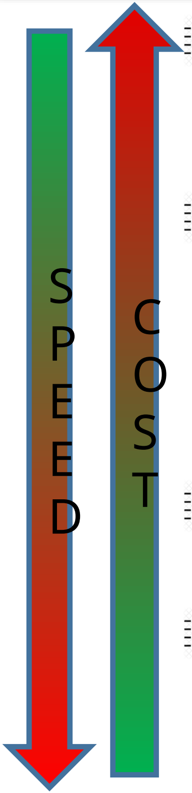
Instruction Fetch	Instruction Decode Register Fetch	Execute Address Calc.	Memory Access	Write Back
IF	ID	EX	MEM	WB



Module Outline

- ***Part 1: The Dream Memory***
  - Large+Fast+ Cheap
- ***Part 2: Direct Mapped Memory Design***
  - Understanding the basics
- ***Part 3: Speeding Memory Performance***
- ***Part 4: Virtual Memory***
  - The large memory illusion

Memory Technologies



SRAM

- uses more transistors per bit and keeps the data as long as it is powered

DRAM

- keeps the charge for few milli-sec → needs periodic refresh (Dynamic!)
- Organized in banks that can be accessed simultaneously
- DDR (Double Data Rate) [Transfer on rising and falling clock edges]

Flash memory [EEPROM]

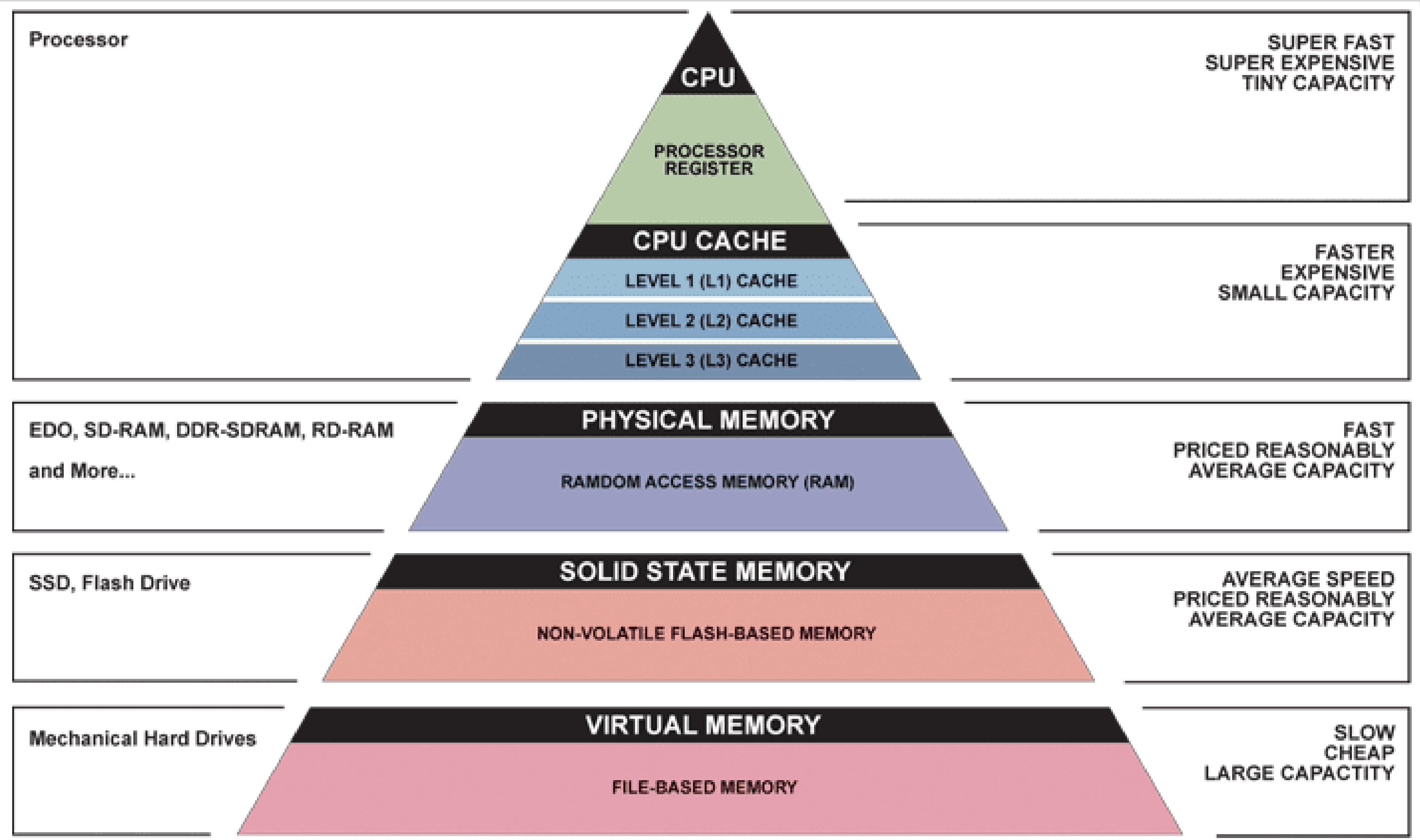
- Avoid wears by distributing dispersing the written blocks

Magnetic disc

- Platters having tracks split into sectors
- Large access time (seek delay + rotational delay + transfer time)

Memory Hierarchy

Cache makes slow MAIN MEMORY appear faster



https://www.google.com/url?sa=i&source=images&cd=&ved=2ahUKEwj7q47ClNXiAhWiVBUIHb7HBs8QjRx6BAgBEAQ&url=https%3A%2F%2Fsites.google.com%2Fsite%2Fcachememory2011%2Fmemory-hierarchy&psig=AOvVaw1IETBmqESPM4ztRnnpso-B&ust=1559920945166571

Virtual Memory makes small MAIN MEMORY appear bigger

Memory Dream

Hierarchal Memory Organization

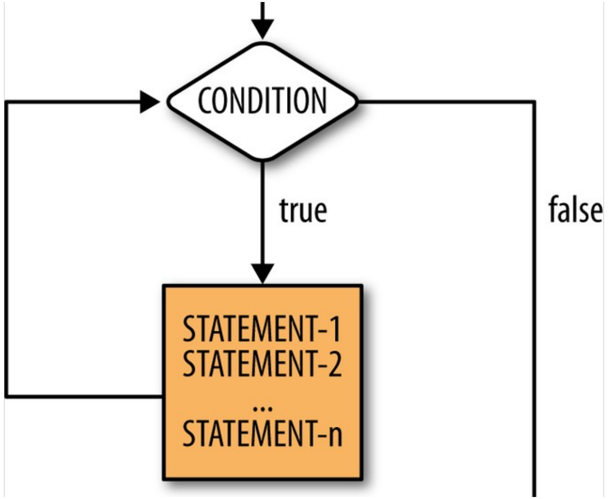
Why does memory Hierarichy work?

Principal of locality

- Programs access a small proportion of their address space at any time

Temporal locality
Items accessed recently are likely to be accessed again soon; e.g., instructions in a loop, induction variables

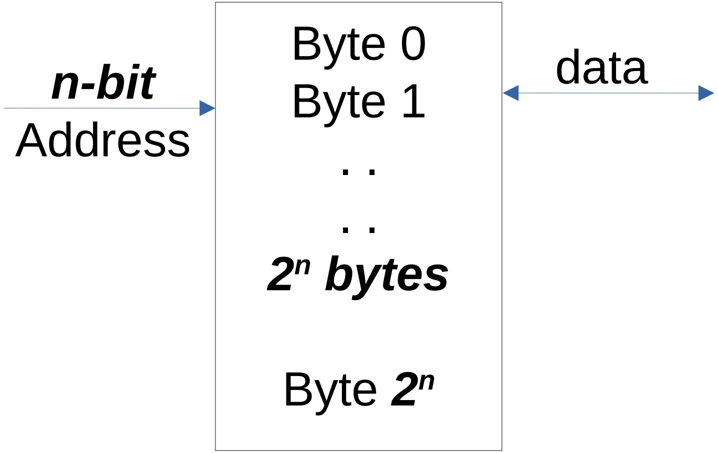
Spatial locality Items near those accessed recently are likely to be accessed soon; E.g., sequential instruction access, array data



Memory Organization & Performance

- ***Memory Hierarchy is transparent to programmer!***
  - Machine AUTOMATICALLY assigns locations, depending on runtime usage patterns
  - Programmer doesn’t (cannot) know where the data is actually stored!
- ***Memory organization impact the computer performance***
  - Organization refers not only to the hieraricy but also to the involved read and write operations to different parts of this hierarchy

Remember: Memory



Memory Blocks

Main memory Blocks

- contains the code and data of active applications stored in units called ***blocks*** (some text call them lines)

Cache Blocks

- contains a subset of the main memory blocks
- data transfer between cache and main memory is based on entire **blocks**

Processor only reads and writes to cache memory

0	1	2	3	4	5	6

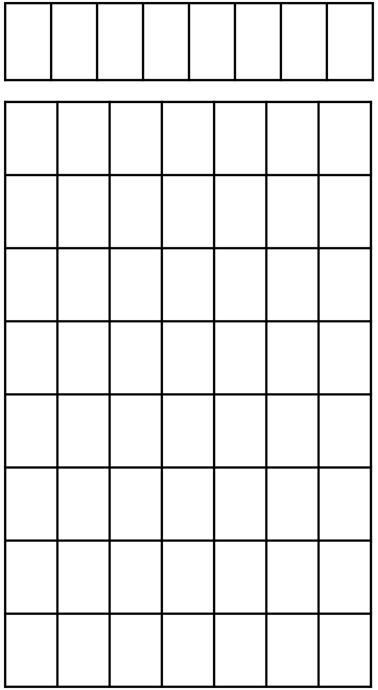
Big picture

The processor requests **data/code words** using their **physical memory address**

The physical memory address depends on the processor architecture (e.g., 32-bit address in MIPS 32, can be smaller or larger for other archs.)

The physical (n-bit) address has two parts

Memory block Identifier	**Block offset**



Example 1A

- *Consider a processor with* ***16-bit address*** *lines using a* ***block size of 16 words****.*
- ***How many lines are used for block offset and identifier?***

16 words/block = 64 bytes/block = 2^6 bytes/block

Block offset = 6 bits

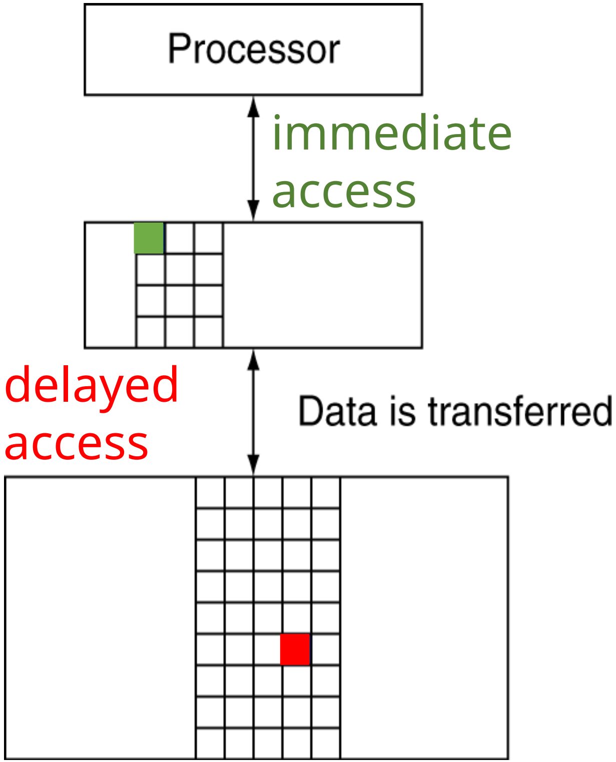
Memory block identifier = address lines - block offset lines

= 16 - 6 = 10 bits

10 bits (Block ID)	**6 bits (Offset)**

Memory Dynamics

- *The processor needs to access data in block X*
  - ***Memory hit:*** processor finds block X in the cache
  - ***Memory miss:*** block X is not currently in the cache
    - Block X should be copied from the main memory to the cache to continue operation (delay penalty)



What happens on cache misses?

1. Stall the CPU pipeline
    - Send original PC value to memory i.e., current PC -4
2. Fetch block from lower level of hierarchy
    - Instruct memory to perform a read and wait for memory to complete access
    - Cache update
        - Put data from memory into data portion of entry
        - Update some cache control bits (later)
3. Resume execution
    1. ***Instruction cache miss:*** restart instruction fetch
    2. ***Data cache miss:*** complete data access [read or write]

Memory Performance Measures

- ***Hit/miss ratio***
  - percentage of memory accesses that results in a memory hit/miss
  - hit ratio α
- ***Average memory access time (t_a)***
  - $t_a = \alpha\, t_h + (1-\alpha)\, t_m$
    - A high hit ratio is desirable
    - A small hit time (t_h) is desired
    - A smaller miss penalty (t_m) is also desirable

Example

Consider a MIPS processor with 64-block cache with 16 bytes/block

What is the cache size?

*How many address lines are used for **block offset**?*

*How many address lines are used for **memory block index**?*

To which memory block does address 1202 belong?

***Memory Block** = Byte address/ Bytes per block*

$$= \lfloor 1200/16 \rfloor = \lfloor 75.125 \rfloor = 75$$

$$1200 = \boxed{1001011}\boxed{0010}_2$$

Part 1: The Dream Memory

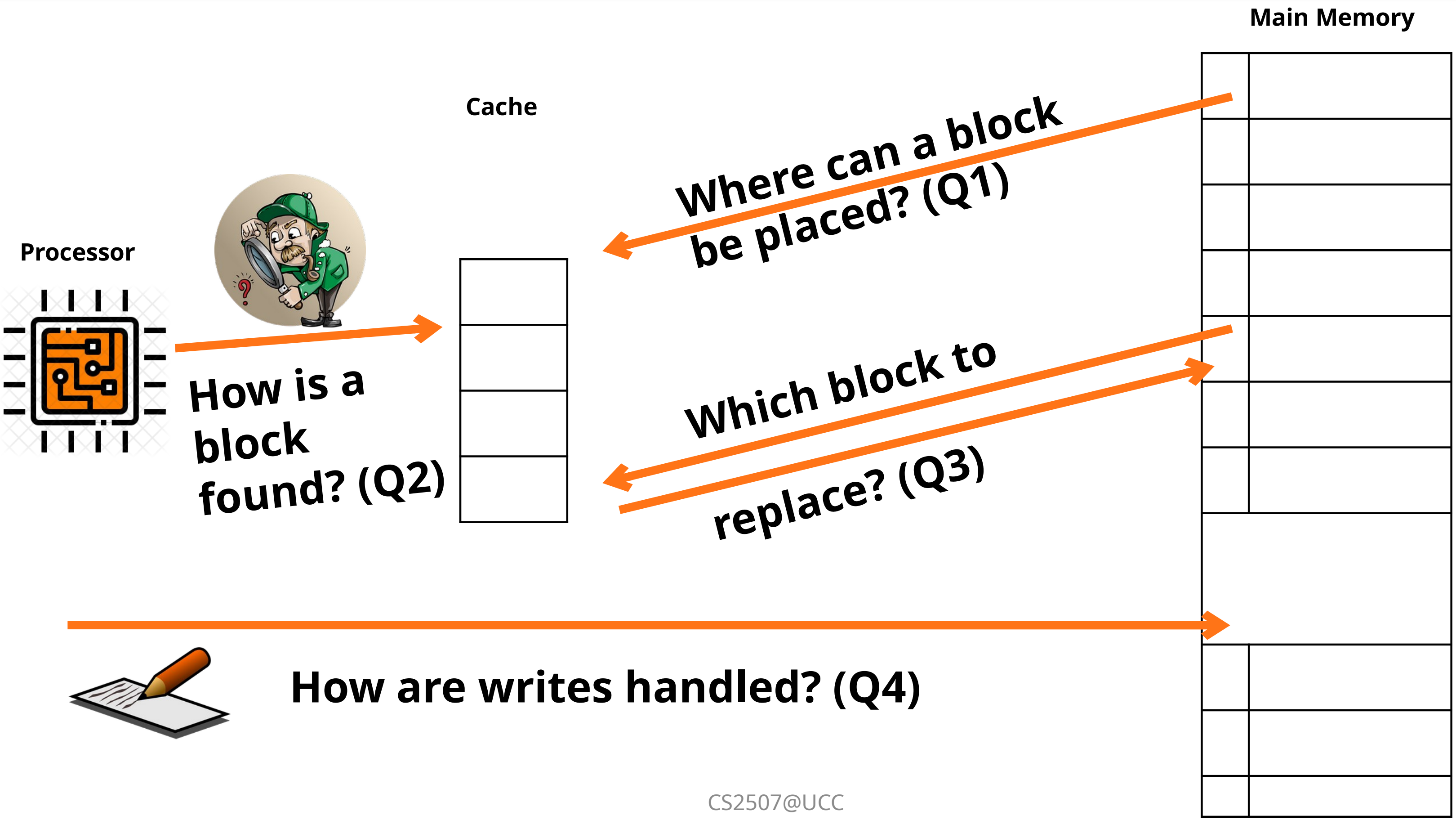
- ***Memory technologies***
- ***Memory Hierarchy***
- ***Memory Operation***
- ***Measuring memory performance***

Sections 5.1-5.2

Part 2: Direct Mapped Memory Organization

Sections 5.3

Key Design Questions



Where can a block be placed? (Q1)

How is a block found? (Q2)

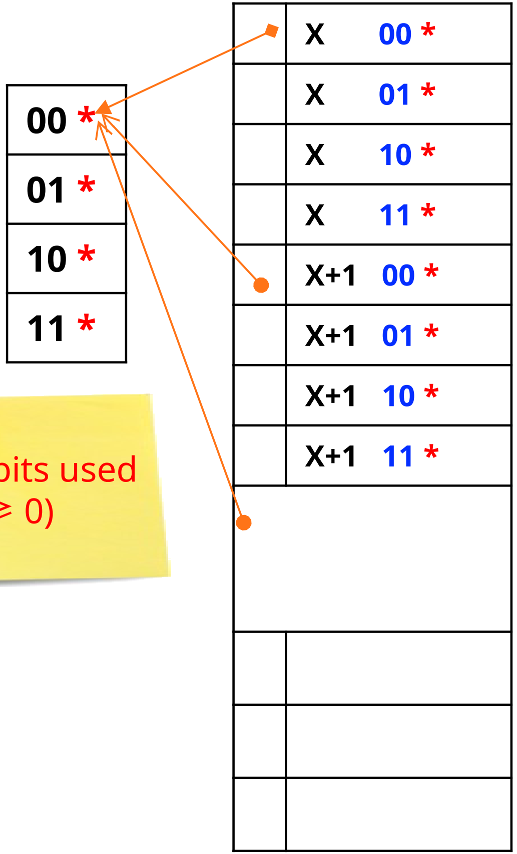
Which block to replace? (Q3)

How are writes handled? (Q4)

Direct Mapped Cache Organization

Where can a block be placed? (Q1)

- The physical address --> cache block location
- Each physical address is mapped to only ONE cache location
- Physical address is split to ***3 parts***



TAG	Cache block Identifier	Block offset

Mem Block ID

* represents the bits used for block offset (⩾ 0)

Tag is used to determine which memory block is actually in the cache

Example: Larger Block Size

Consider a 64-block cache with 16 bytes/block

*To which **cache** block does address 1200 belong?*

Method #1:

- ***Memory Block** = Byte address/ Byte per block = $\lfloor 1200/16 \rfloor$ = 75*
- ***Direct-mapped Cache Block** = 75 modulo 64 = **11***

Method #2:

- *64 blocks = 2^6 blocks → **n = 6***

  16 byte/block = 2^4 bytes/block

  MIPS TAG size = 32 – (6+4) = 20 bit

1200 =

0..01 **001011** **0000**

TAG Block Index Block Offset

Example:

- *Consider a processor with a max memory size of 64 MB, 4KB cache, and block size of 64 bytes. What are the cache block index and TAG values for the following memory addresses?*

    - 500

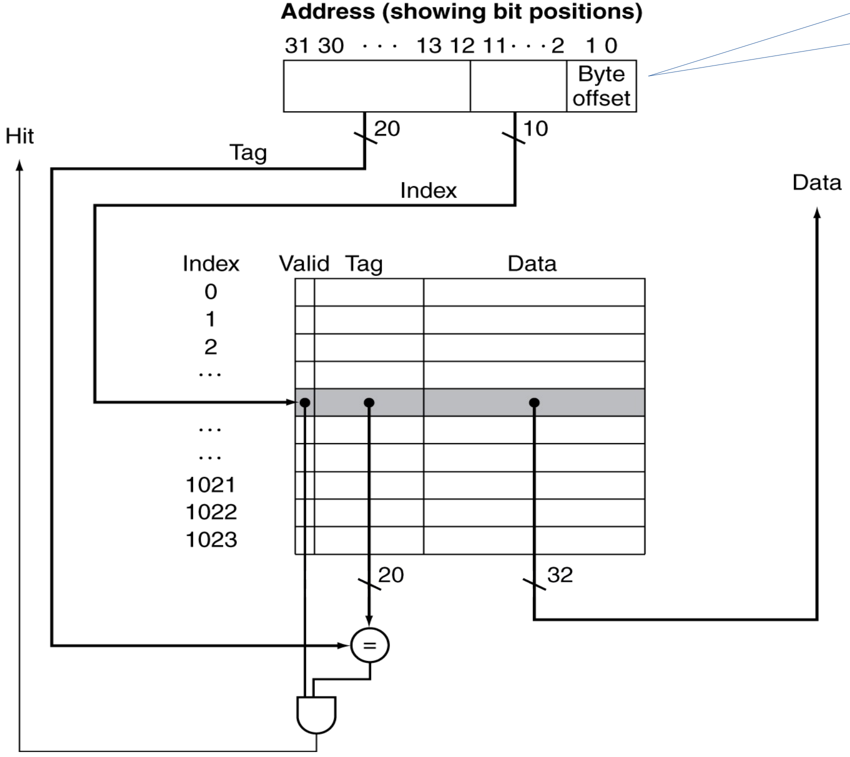
    - 0x 07 d0

Direct Mapped Cache Block Search

- ***How is a block found? (Q2)***
  - **store** the tag of memory blocks in cache
  - **check** the tag to know if the block exists in the memory or no
- ***How do we know the cache block has valid data?***
  - ***Valid bit:*** 1 = present, 0 = not present
  - Initially 0 (when the machine starts)

The claimed cache size refers to the data space without the additional control overhead such as TAG bits and VALID bit

Direct Mapped Cache HW



Calculated by the processor

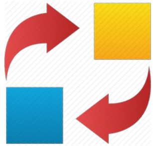
If tag and upper 20 are equal and valid bit set to 1 then we have a hit

Direct Mapped Memory Cache Size

- ***Cache stores [data + Tag + valid bit] for every block***
  - Cache size = number of blocks * (block size + tag size + valid field size)
  - Tag size = Address size - Index bits - OFFSET bits
- ***E.g.: MIPS with 16-byte blocks and 1Kb cache***
  - 16 bytes/block --> 4-bit offset field
  - # cache blocks = $2^{10}/2^{4} = 2^{6}$ --> 6-bit Block Index
  - tag size = 32 - 6 - 4 = 22 bits
  - Cache size = …….

Block Replacement Strategies

- ***What block is replaced on a miss? (Q3)***
  - In Direct-mapped memory, there is only one possible place for every block.
  - A trivial question



Handling Cache Writes (Q4)

write hit (Block in cache)

- ***Write-through*** technique: Update **both** cache and next lower level memory hierarchy

- ***Write-back*** technique: Update slower memory level when the entire block is replaced

When to update the main memory? Consistent vs. speed

write miss (Block is not in cache)

Write-allocate: load the block in the cache and update in the cache

No write allocate: update the written address using write through without loading the block

Write-through **technique**

Consistent ++

slow (hmm?)

Example: if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles

- Effective CPI = 1 + 0.1×100 = 11

- ***Speeding write-through using*** a ***write buffer***
  - A buffer holds data waiting to be written to memory
  - CPU continues immediately
    - ***Only stalls on write if write buffer is already full (not bad at all! :)***

Write-Back

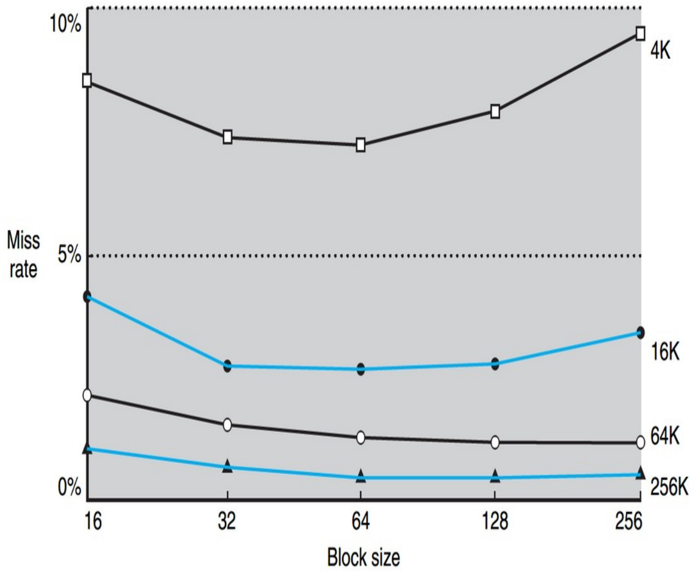
- **Fast writes** but **a higher miss penalty**
  - Write every block back to main memory
  - Doubles a miss penalty, why?
- Reducing miss penalty
  - A ***dirty bit*** is set whenever a cache block is updated
  - When an updated (dirty) block is replaced
    - Write it back to memory
    - Can use a ***write buffer*** to allow replacing block to be read first

Write back is a good option with frequent block writes and/or slower memory

Which is better large or small block size?

Larger blocks

- should reduce miss rate due to spatial locality
- Larger miss penalty
- higher miss rate (fewer blocks in a fixed cache size)



Revision

Memory Hierarchy **Principle of Locality**

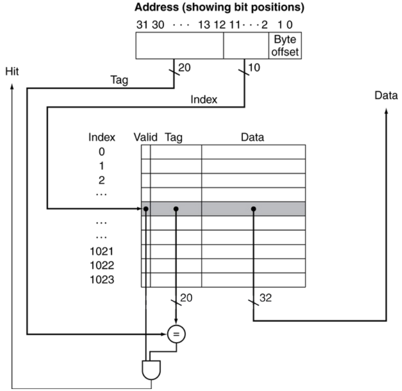
Block

Placement: Where can a memory block be placed in the cache? (Q1)
Search: How does the CPU knows the block is in cache? (Q2)
Replacement: Which block is replaced on a miss? (Q3)
Write: How are writes handled? (Q4)

CPU needs byte at address 2500
Given: **64-byte Block** & **16-block Cache**
Direct mapped
1- MEM block ID? 2500/64 = 39.0626
2- Cache block 39 mod 16 = 7
search: hit or miss

Direct Mapped Cache
A1: one-to-one mapping for address
A2: fast search HW
A3: trivial
A4: writing strategies

2500 =0b100111000100



Measuring Cache Performance

Cache Performance Analysis

CPU time = (# CPU execution cycles + CPU stall cycles) * cycle period

$$\text{Memory-stall clock cycles} = (\text{Read-stall cycles} + \text{Write-stall cycles})$$

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Can be ignored with sufficiently deep buffer and/or fast writing procedures

Measuring Cache Performance

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

Example:

- Instruction-cache miss rate = 2%
- Data-cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI (ideal cache) = 2
- Load & stores are 36% of instructions

Stall cycles

- I-cache: 0.02 × 100 = 2
- D-cache: 0.36 × 0.04 × 100 = 1.44

- Actual CPI = 2 + 2 + 1.44 = 5.44
- Ideal CPU is 5.44/2 =2.72 times faster

What if the CPU CPI is 1 (faster processor)?

Performance Concerns

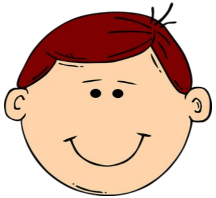
- ***What happens when CPU performance increased?***
  - Decreasing base CPI
    - Greater proportion of time spent on memory stalls
  - Increasing clock rate
    - Memory stalls account for more CPU cycles
- ***When CPU performance increased***
  - Miss penalty becomes more significant
  - Can’t neglect cache behavior when evaluating system performance

Average Memory Access Time

- Hit time is also important for performance
- Average Memory Access Time (AMAT)
  - AMAT = Hit time + Miss rate × Miss penalty
- Example
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Good memory performance

1. *A small hit time*
2. *A low miss ratio*
3. *A small miss penalty*



Revision

Memory Hierarchy **Principle of Locality**

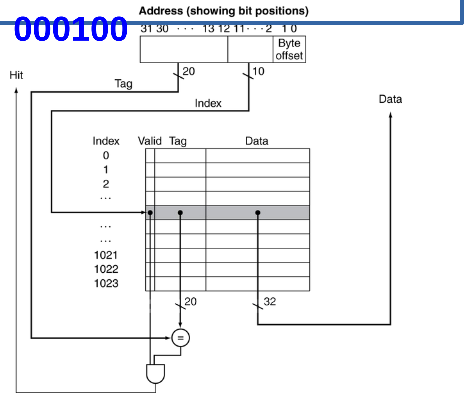
Block

Placement: Where can a memory block be placed in the cache? (Q1)
Search: How does the CPU knows the block is in cache? (Q2)
Replacement: Which block is replaced on a miss? (Q3)
Write: How are writes handled? (Q4)

Direct Mapped Cache
A1: one-to-one mapping for address
A2: fast search HW
A3: trivial
A4: writing strategies

CPU needs byte at address 2500
Given: **64-byte Block** & **16-block Cache**
Direct mapped
1- MEM block ID? 2500/64 = 39.0626
2- Cache block 39 mod 16 = 7
search: hit or miss

2500 =0b10 0111 **000100**



Good memory performance
A small hit time
A low miss ratio
A small miss penalty

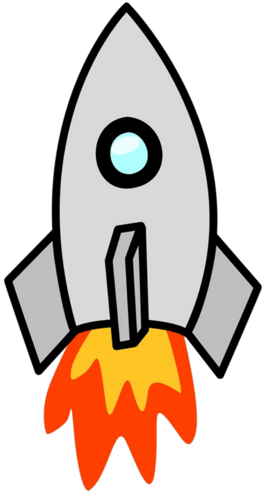
]

Improving Cache Performance

Reduce miss rate (??)

Reduce miss (time) penalty (??)

Speed hit access time!



Example: Direct Mapped Cache

- Consider a system with a **4-block** cache using Direct mapped memory organization
  - **Block access sequence**: 0, 8, 0, 6, 8
  - Initially, we assume all cache entries are empty

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

Reducing Miss Rate

- *Flexible block location to reduce the competition for cache blocks → **Associative Caches***

Fully associative caches

✓ Any memory block can go to any cache entry (less competition)

Problem: Where is my block?

✓ Search all entries at once to reduce hit time (HW solution **$$$**)

n-way set-associative caches

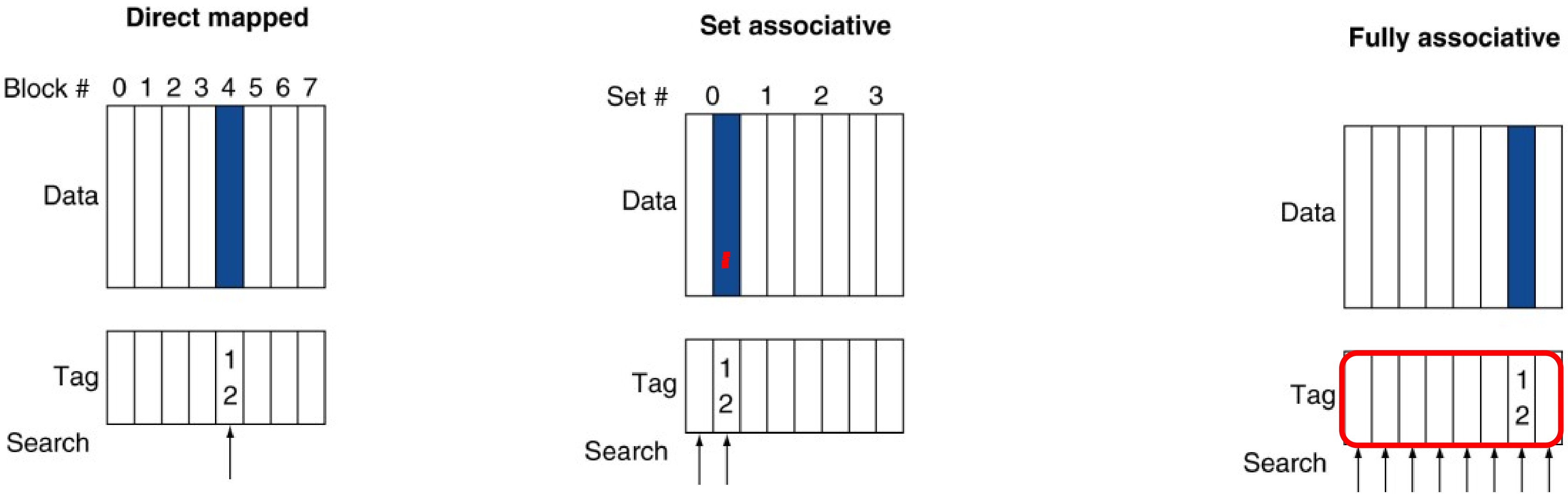
- A memory block can be located to a set of *n* cache blocks

Which Set? (Memory Block number) modulo (#Sets in cache)

Which Block? Any block in the set

How to search? Search all possible entries in a given set at once ($)

Comparing Cache Organization

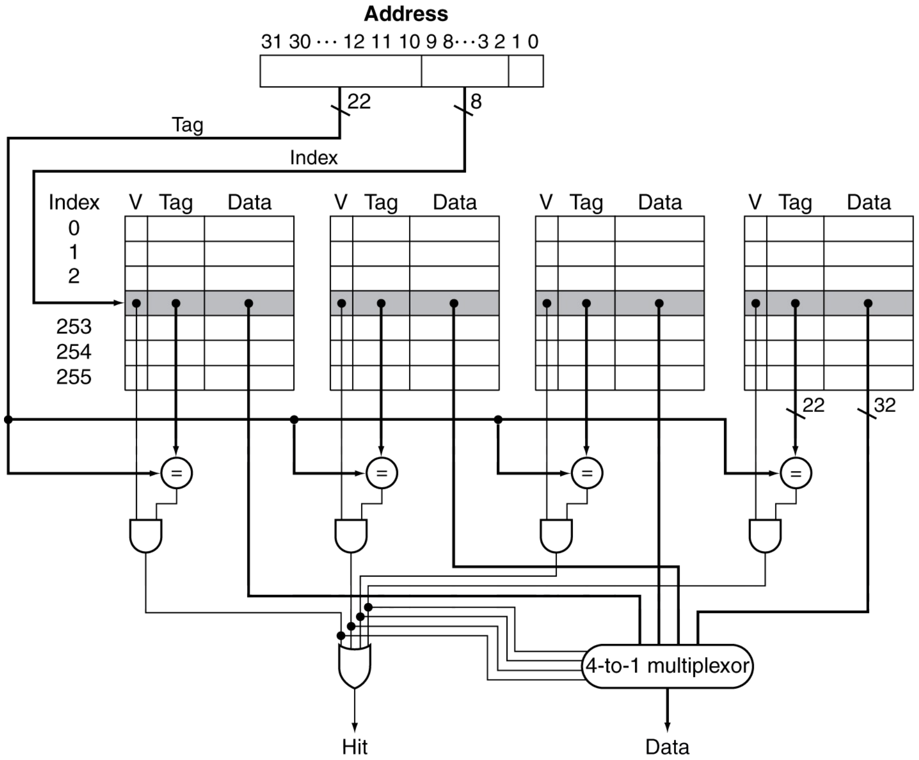


Block position =
(Memory Block #)
modulo
(# of cache blocks)

Set position =
(Memory Block #)
modulo
(# of cache sets)

Set Associative Cache Organization

Each block may be in a different locations → Additional HW to support simultaneous search is used to ensure that the performance is not affected.



n-set Associativity Example

- Consider a system with a **4-block** cache using **2-way** set associative memory organization
    - **Block access sequence**: 0, 8, 0, 6, 8

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		**Set 1**	
0	0	miss	**Mem[0]**			

Hit ratio: 0% → 1/5 = 20%

Fully Associativity

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

Hit ratio: 0% → 2/5 = 40%

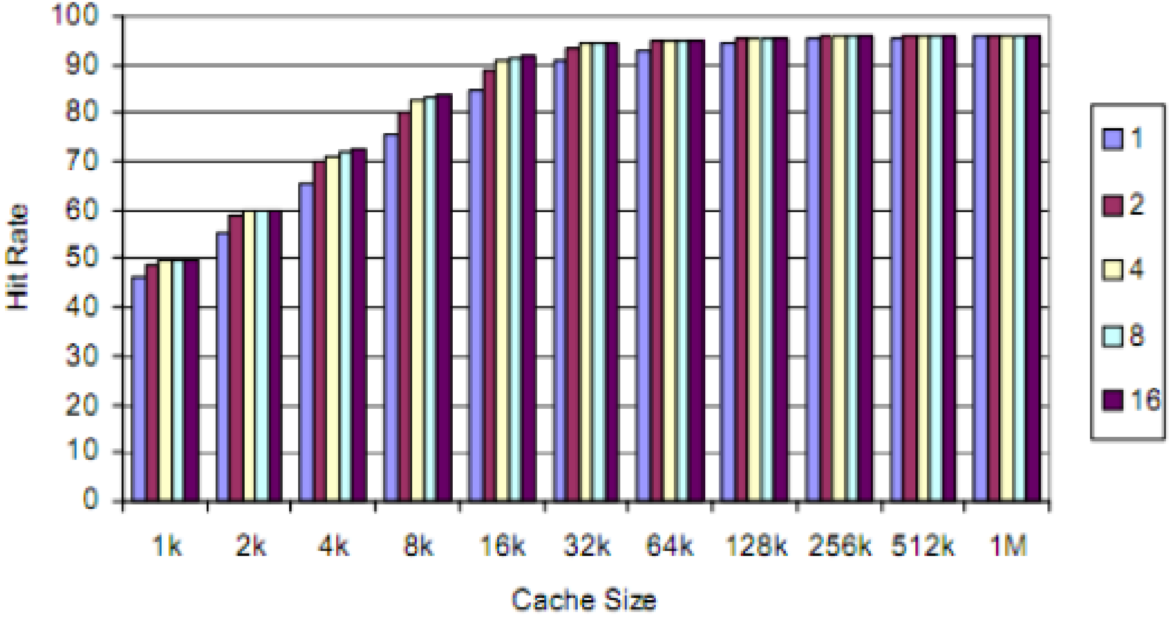
How Much Associativity?

Increased associativity decreases miss rate **but with diminishing returns**

Performance-cost tradeoff

Better performance requires additional hardware for speeding hit time

Hit Rate v. Associativity & Cache Size



(L1 cache, Running GCC)

Replacement Policy

- ***Direct mapped: no choice***
- ***Set associative***
    - Prefer non-valid entry, if there is one
    - Otherwise, choose among entries in the set

Least-recently used (LRU)

Choose the one unused for the longest time

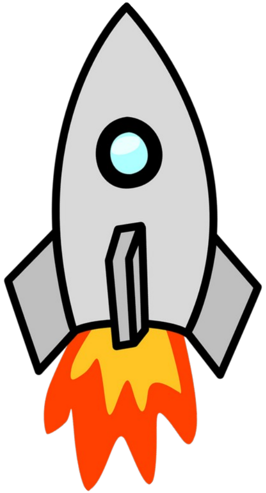
- Simple for 2-way, manageable for 4-way, too hard beyond that

Random

Gives approximately the same performance as LRU for high associativity

Improving Cache Performance

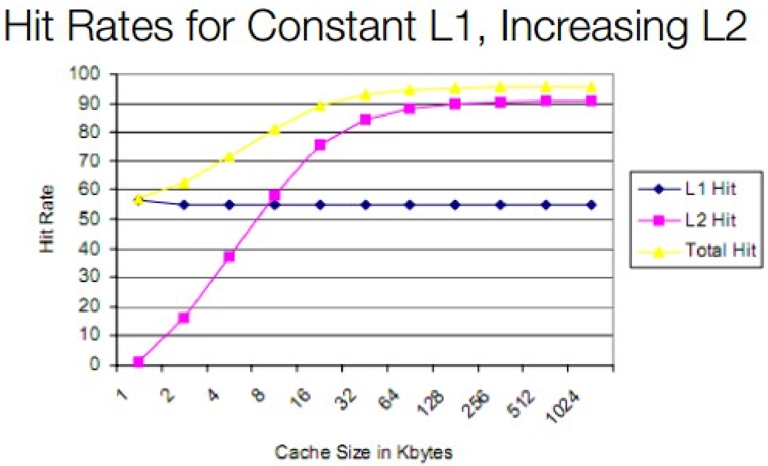
Reduce miss rate (**Associative cache**)
Reduce miss (time) penalty (??)
Speed hit access time!



Reducing the miss penalty

Multi-level Cache

- Level-1 cache (Primary cache): service the CPU, Small (~KB), but fast
- Level-2 cache (~MB): services misses from primary cache, still faster than main memory



Multi-level cache organization

The L3 cache tends to be around 8 MB.

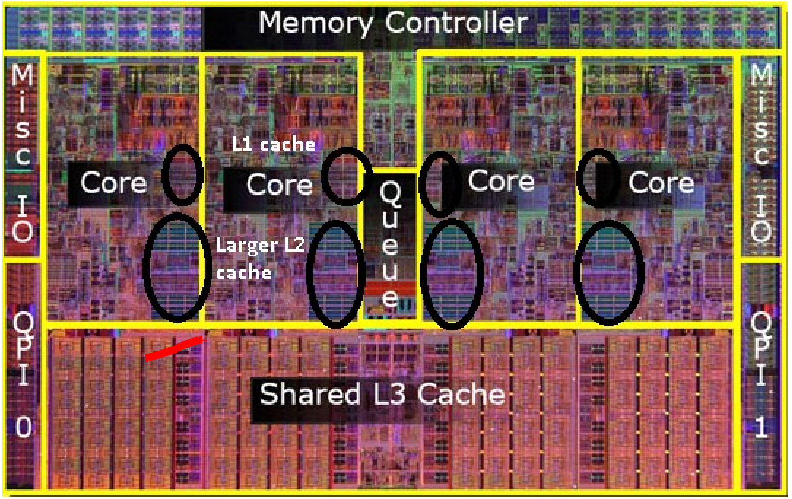
Performance difference between L1, L2 and L3 caches

L1 cache access latency: x cycles

L2 cache access latency: ~3x cycles

L3 cache access latency: ~9x cycles

Main memory access latency: ~30x cycles



Multilevel Cache Example

- CPU base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns

With just a primary cache

- Miss penalty = 100ns/0.25ns = 400 cycles
- Effective CPI = Base CPI + Memory-stall cycles per inst.

  = 1 + 0.02 × 400 = 9

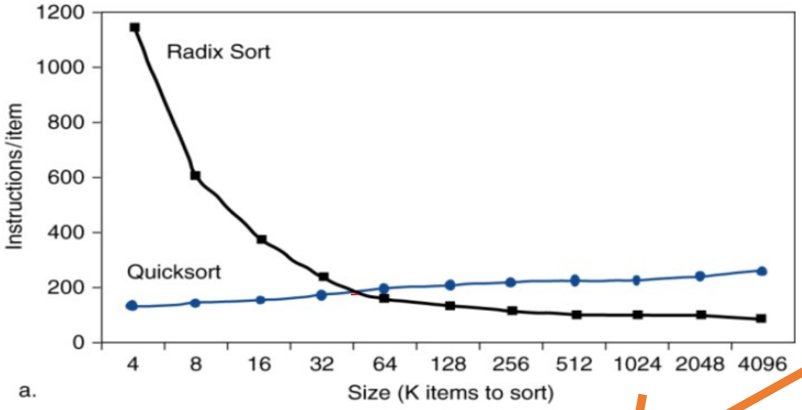
Example (cont.)

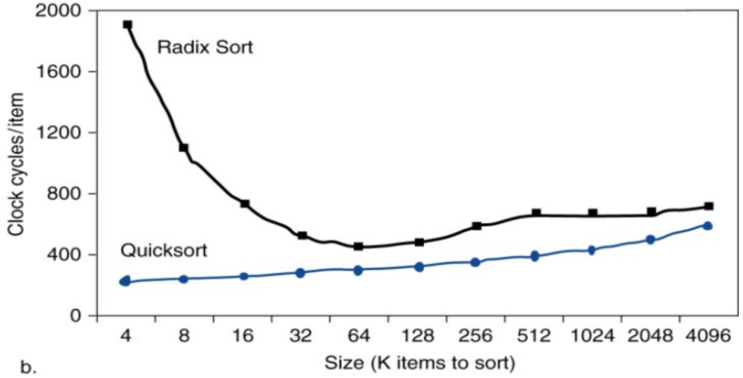
- ***Now add L-2 cache***
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- ***Primary miss with L-2 hit***
  - Penalty = 5ns/0.25ns = 20 cycles
  - Total CPI = 1 + primary stall + secondary stall
  - CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4
- ***Performance ratio = 9/3.4 = 2.6***
  - Faster by a factor of 2.6 with a secondary cache

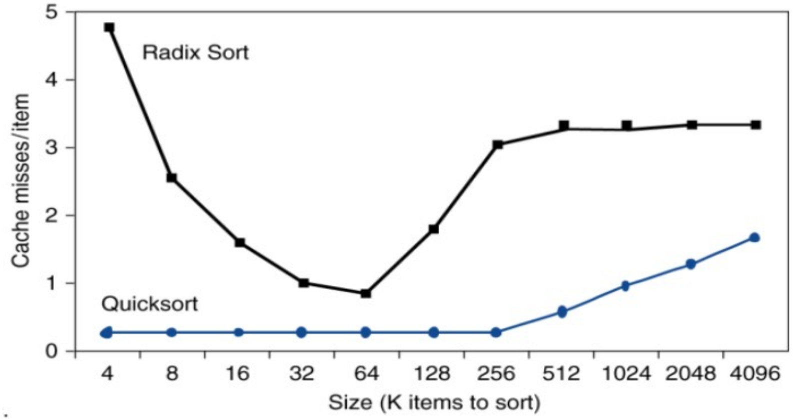
Multilevel Cache Considerations

- ***Primary cache***
  - Focus on minimal hit time
- ***L-2 cache***
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- ***Results***
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

Interactions with Software







- ***Misses depend on memory access patterns***
  - Algorithm behaviour
  - Compiler optimization for memory access

How SW developer code for cache performance ?

https://tinyurl.com/Cpu-Caches-Why-Care

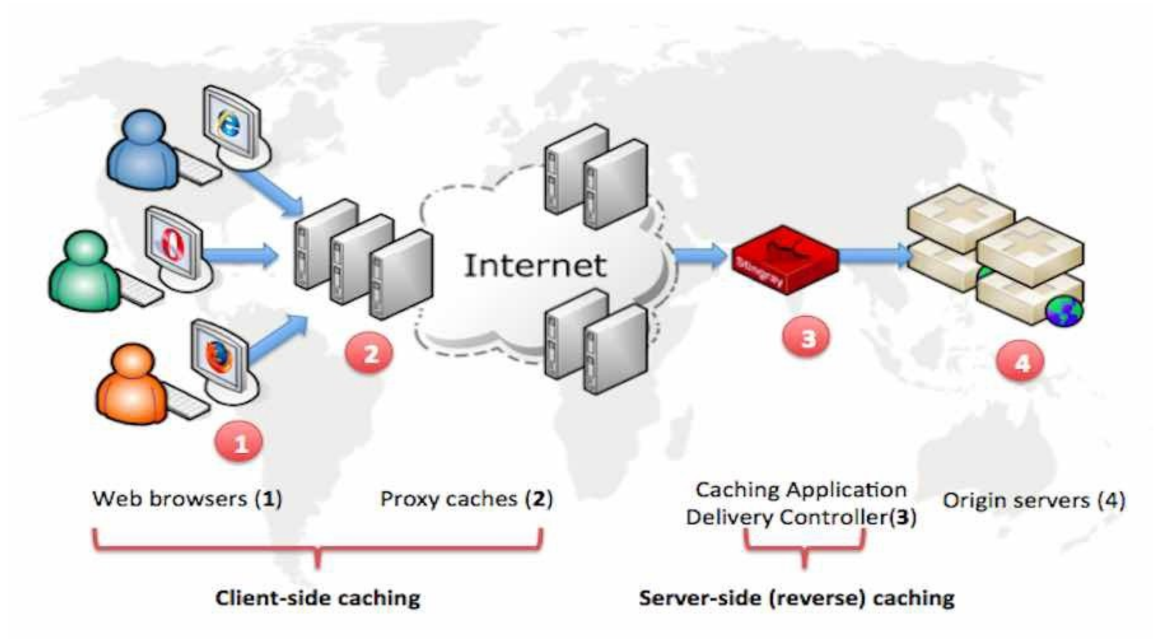
https://www.youtube.com/watch?time_continue=62&v=WDIkqP4JbkE&feature=emb_logo

Revisiting statistical library design

A library has implementation for: min, max, average, 25%tile, ….
Common task: find all stats for long arrays

How would you code that? Why?

Caching is effective in other systems!



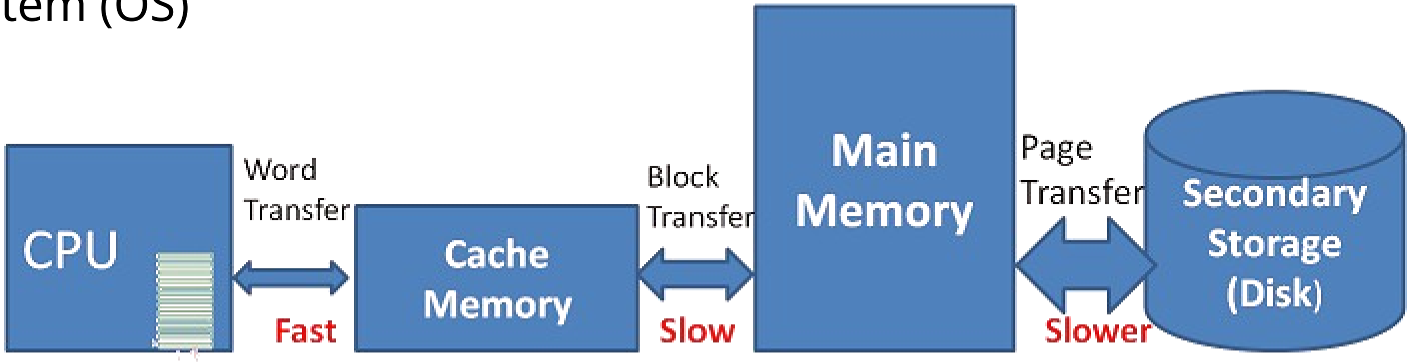
Virtual Memory

The large memory illusion

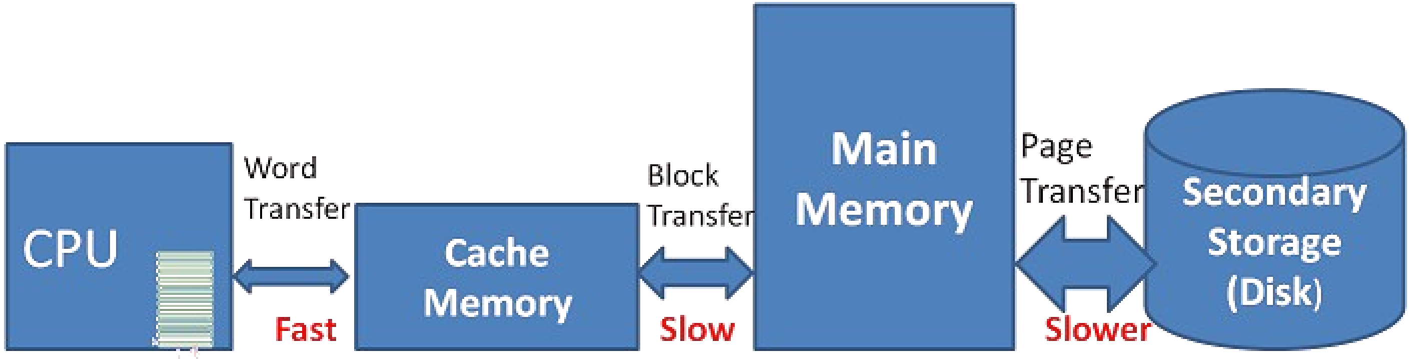
Sections 5.7

Virtual Memory

- ***The physical address may be limited by the installed memory or connected HW address lines***
  - Virtual memory enables extending the memory size beyond main memory
  - Virtual memory extends the physical memory to the HDD
  - Virtual memory is managed jointly by CPU hardware and the operating system (OS)



Virtual Memory



- *Programs share main memory*
  - Each gets a private virtual address space holding its code and data
  - Physical memory is used as a “cache” for the ***virtual memory file***
  - Virtual memory “block” is called a **page**
  - Virtual memory “miss” is called a ***page fault***
  - ***Note that cache-main memory remains operating at a higher level in the hierarchy***

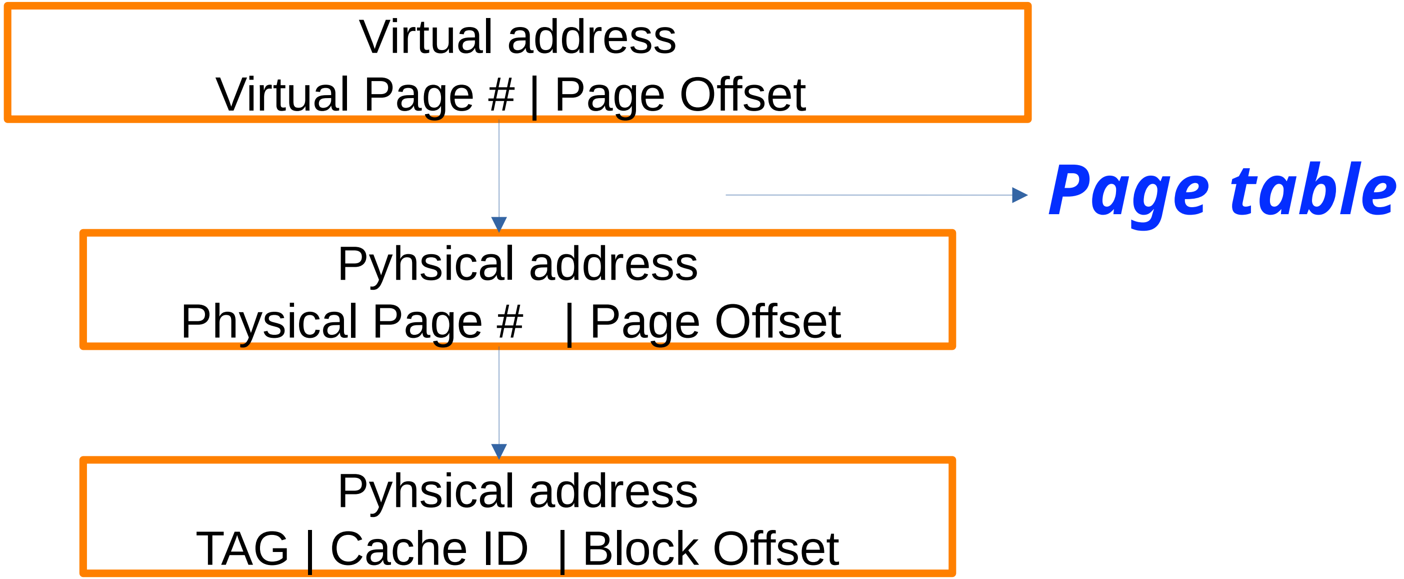
Virtual memory design

- ***Minimize page fault rate, Why?***
  - Every page fault → the page must be fetched from disk → Takes **millions** of clock cycles

- ***How to achieve this design goal?***
  - **Fully associative placement** in the main memory (RAM)
  - **Smart page replacement:** OS prefers least-recently used (LRU) replacement
    - Reference bit (aka use bit) in Page Table set to 1 on access to page
    - Periodically cleared to 0 by OS
    - A page with reference bit = 0 has not been used recently
  - **Practical write policy** (Disk writes take millions of cycles)
    - Write through is impractical
    - Use **write-back** when replacing a page
    - Dirty bit in Page Table set when page is written

Virtual Address Translation

- *CPU and OS* ***translate*** *virtual addresses to physical addresses before checking for cache availability*



Virtual address
Virtual Page # | Page Offset

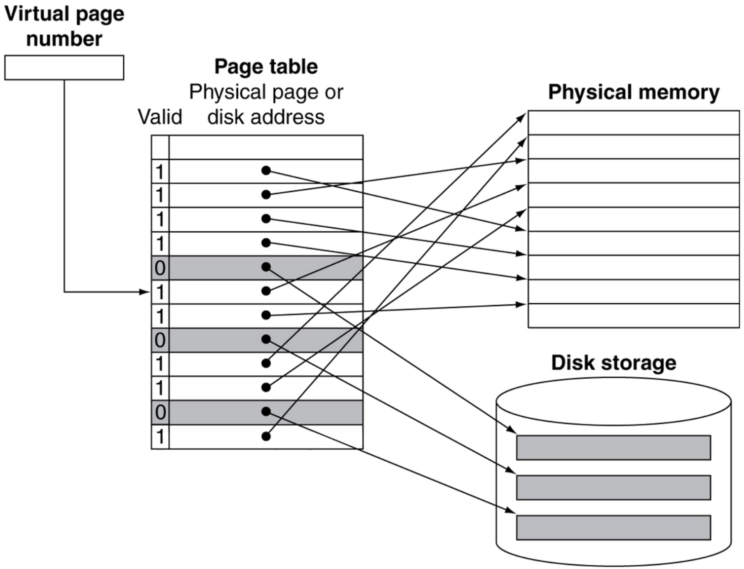
Page table

Pyhsical address
Physical Page # | Page Offset

Pyhsical address
TAG | Cache ID | Block Offset

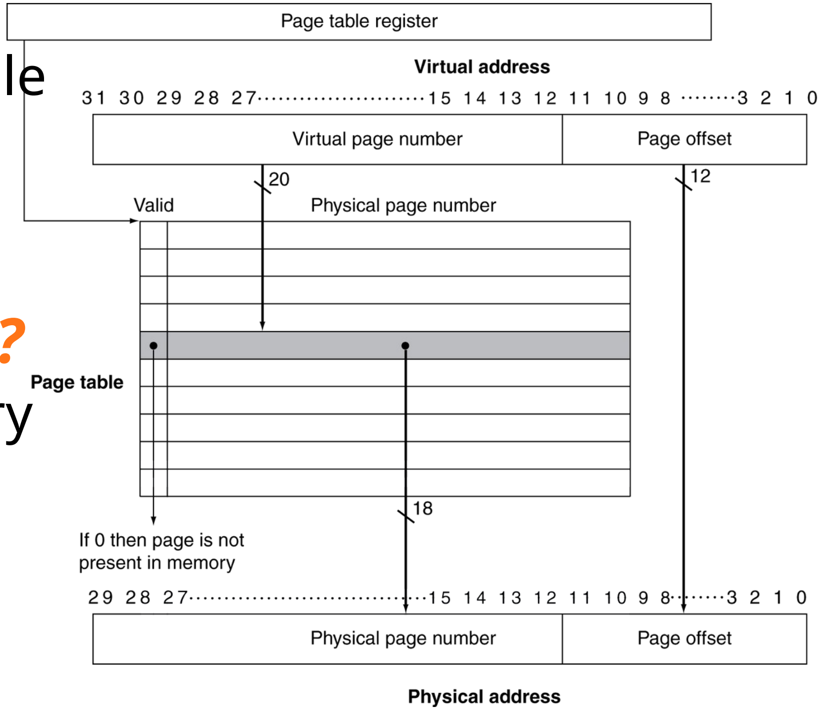
Page table

- Array of ***page table entries (PTE)***, indexed by virtual page number
- If page is present in memory → Page table entry stores the physical page number (Plus other status bits (referenced, dirty, …)
- If page is not present → Page table entry can refer to location in swap space on disk



Address translation

- *Each program has its own page table*
  - hardware includes a register that points to the start of the page table ***[page table register]***



- ***How many times is memory referenced to access a variable?***
  - One to access the Page Table entry
  - Then the actual memory access

Speeding Address Translation

- ***Use*** ***Translation Look-aside Buffer (TLB)***
  - a fast cache of ***Page Table Entries*** (**PTE**) within the CPU
  - Typical: 16–512 PTEs
    - **Hit**: 1 cycle
    - **Miss**: 10–100 cycles
  - Access to page tables has good locality
    - 0.01%–1% miss rate

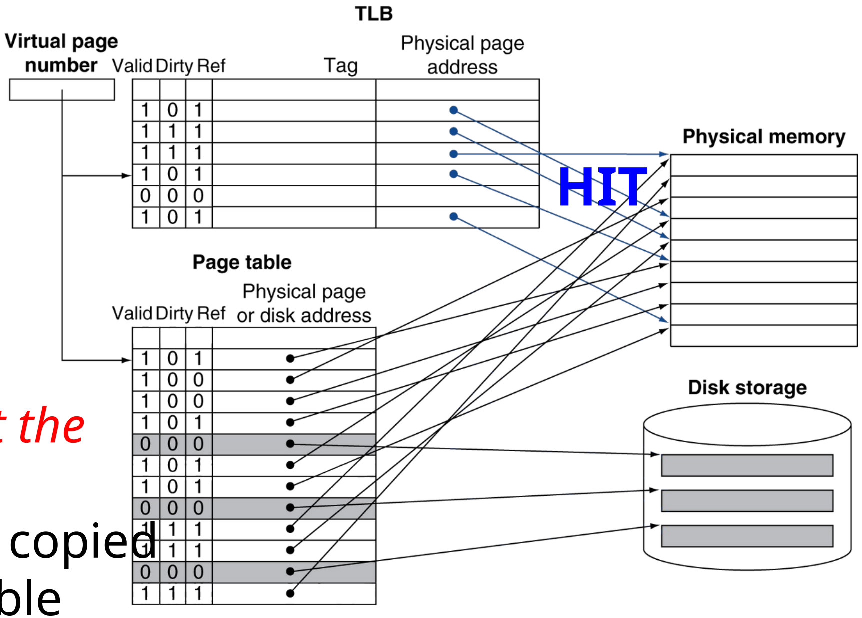
TLB operation

TLB misses will be much more frequent than ***true page faults***

TLB MISS

PTE is not in TLB but the page is in the RAM

1. Replaced PTE is copied back to page table
2. PTE from memory to TLB
3. Then restarts instruction



TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		
1	1	1		
1	1	1		
1	0	1		
0	0	0		
1	0	1		

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	
1	0	0	
1	0	0	
1	0	1	
0	0	0	
1	0	1	
1	0	1	
0	0	0	
1	1	1	
1	1	1	
0	0	0	
1	1	1	

True page fault

the PTE is not in TLB and the page is on the HDD (not in RAM)

the processor invokes the operating system using an exception

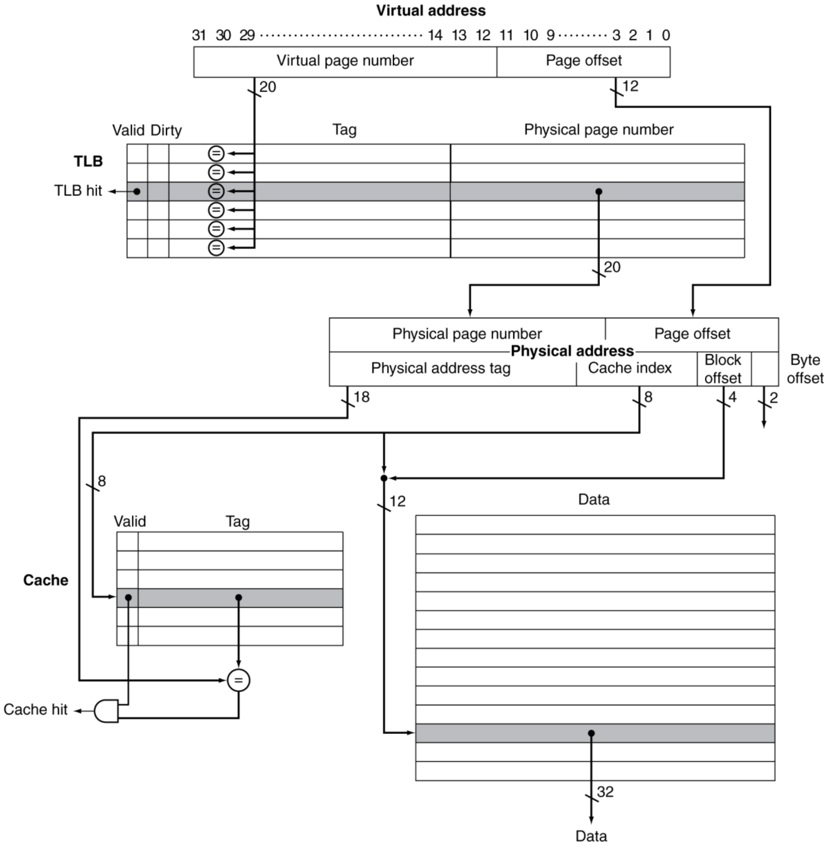
Page Fault handling

- *Handled by the OS*
- *Use faulting virtual address to find PTE*
- *Locate page on disk*
- *Choose page to replace*
  - If dirty, write to disk first
- *Read page into memory and update page table*
- *Make process runnable again*
- *Restart from faulting instruction*

TLB and Cache Interaction

Translate virtual address to a physical address

identify if the content is available in the cache or not

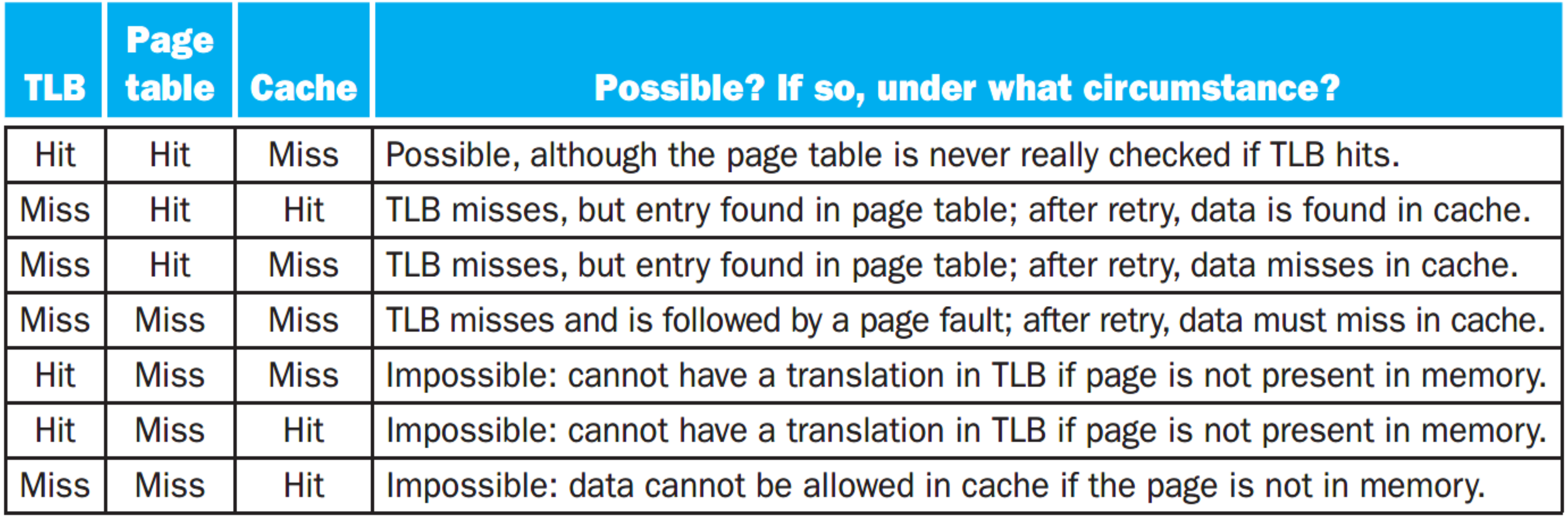


Discussion

Is the PTE available in the TLB?

Is the needed page already loaded in the Memory?

Is the target memory address loaded in the cache?



	TLB	Page table	Cache	Possible? If so, under what circumstance?
G1	Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
G2	Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
G3	Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
G4	Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
G3	Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
G2	Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
G1	Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

Sections 5.8 for the big picture