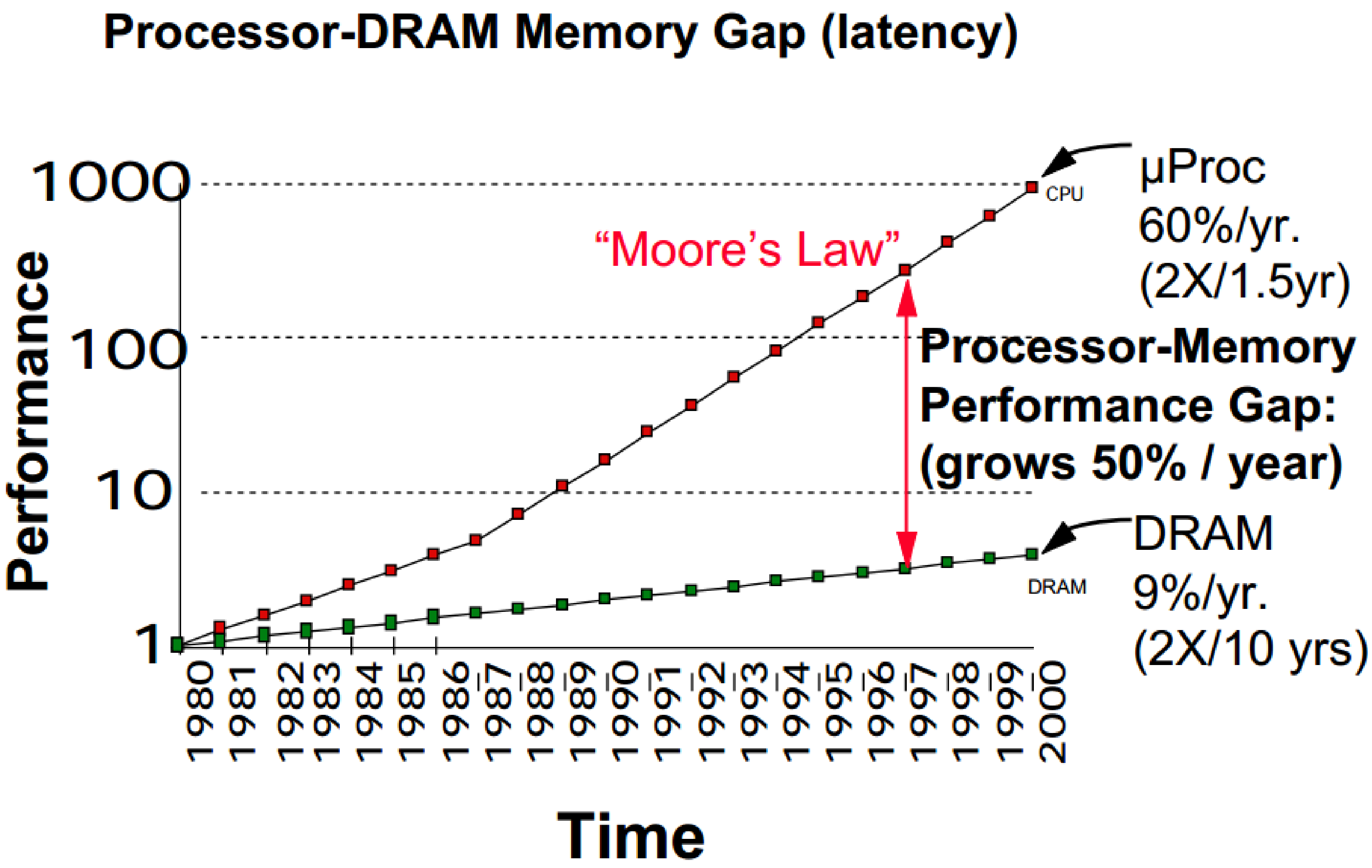
CS425
Computer Systems Architecture

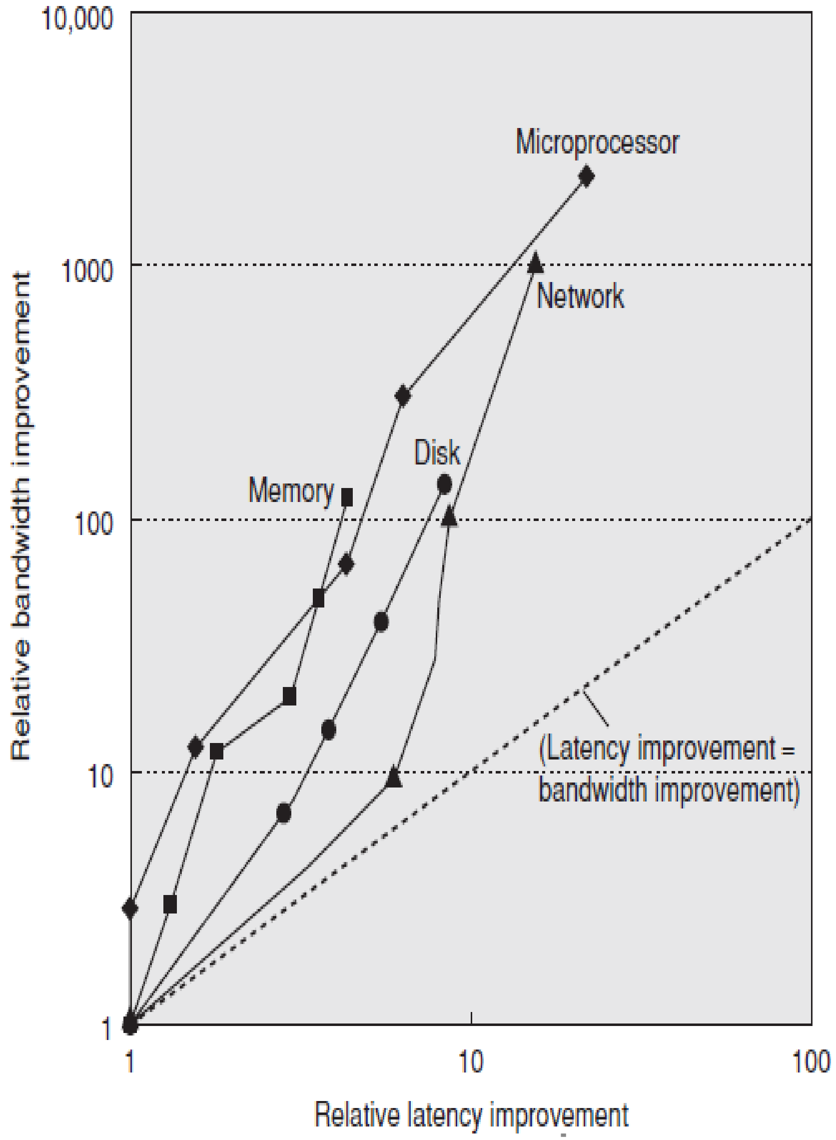
Fall 2022

Caches: The Basics

Who Cares about Memory Hierarchy?



Latency lags bandwidth

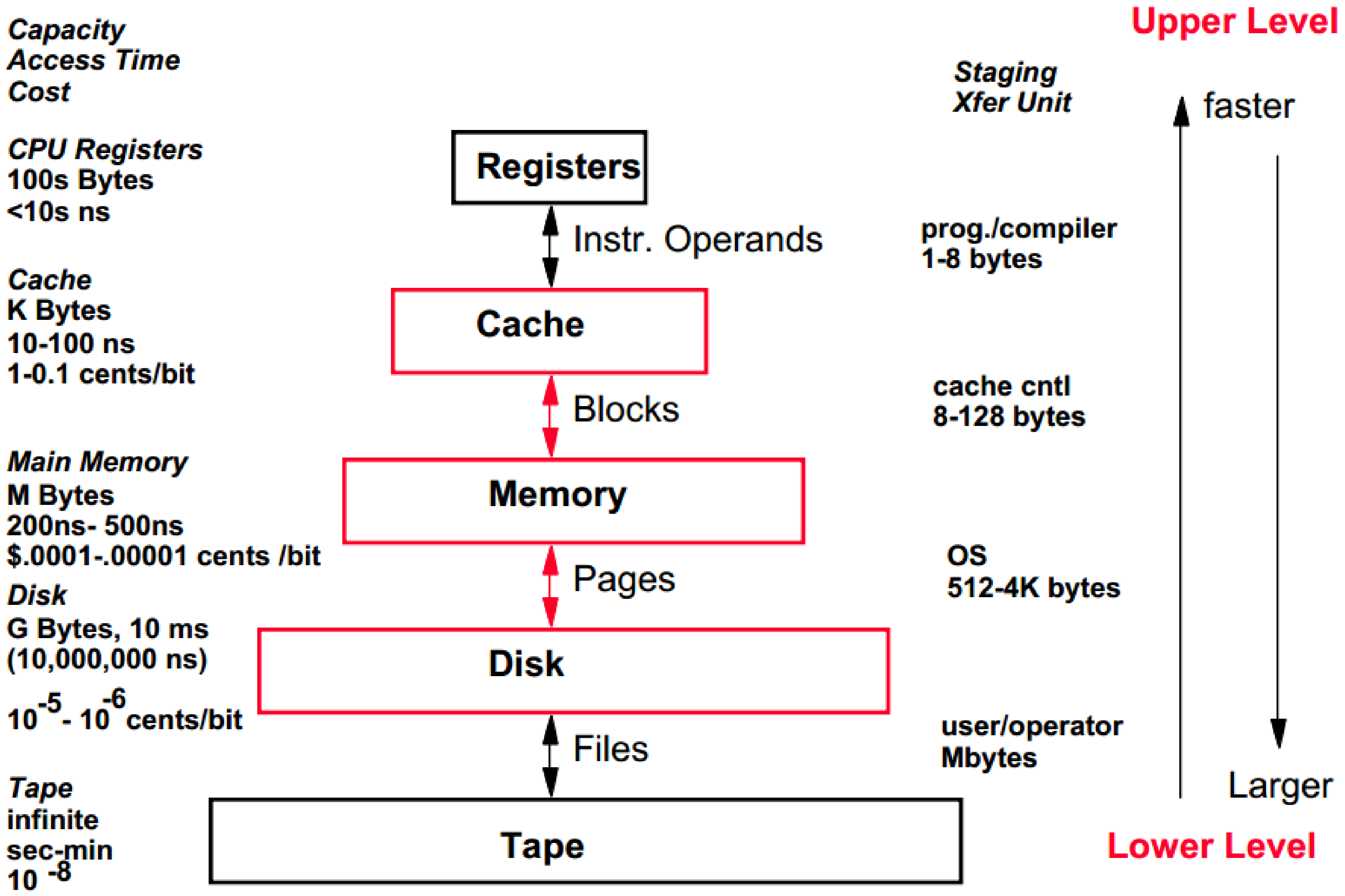


Reasons for Bountiful Bandwidth but Lagging Latency

"There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God."

—Anonymous

Levels of Memory Hierarchy

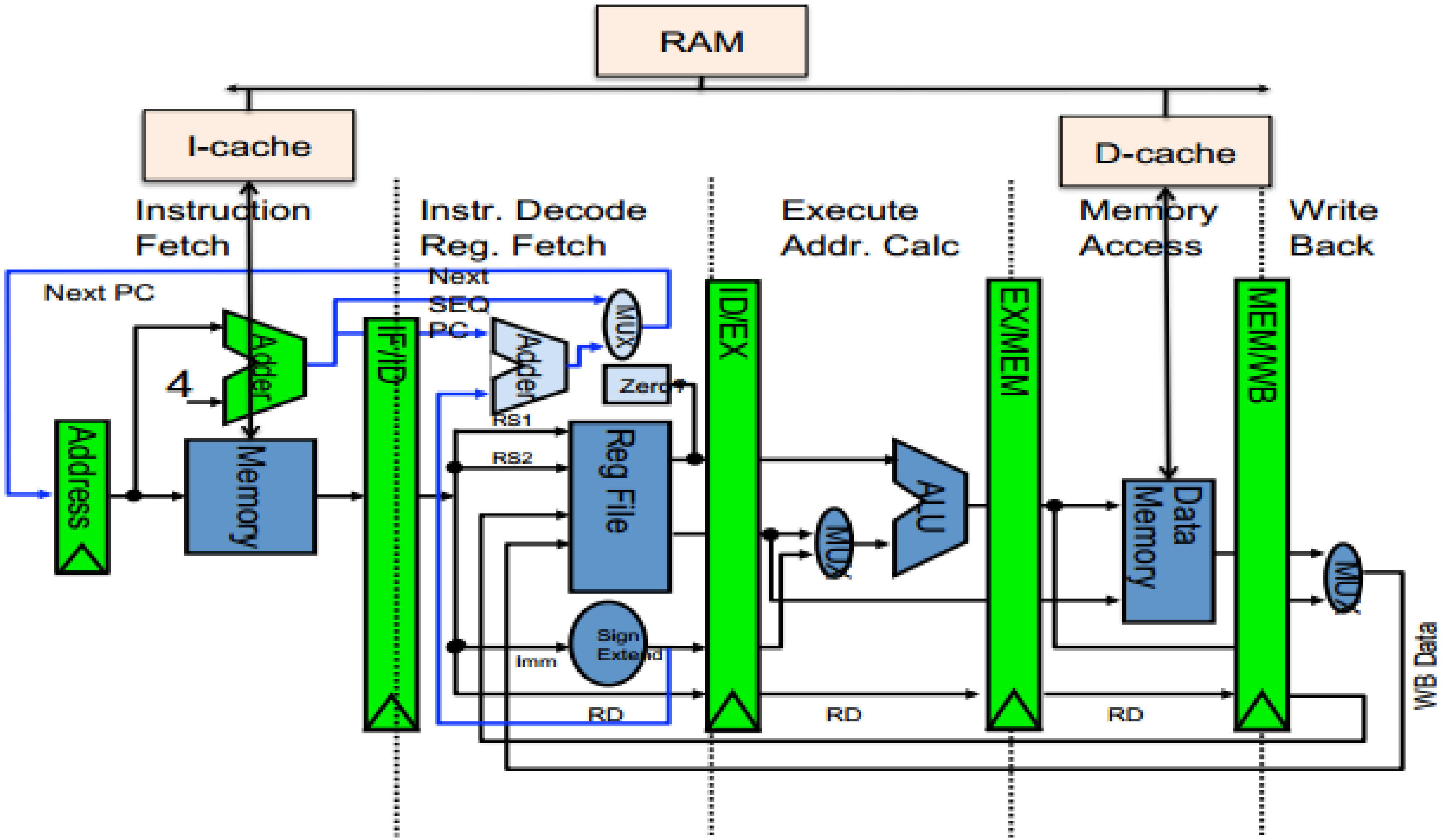


Definition of Cache

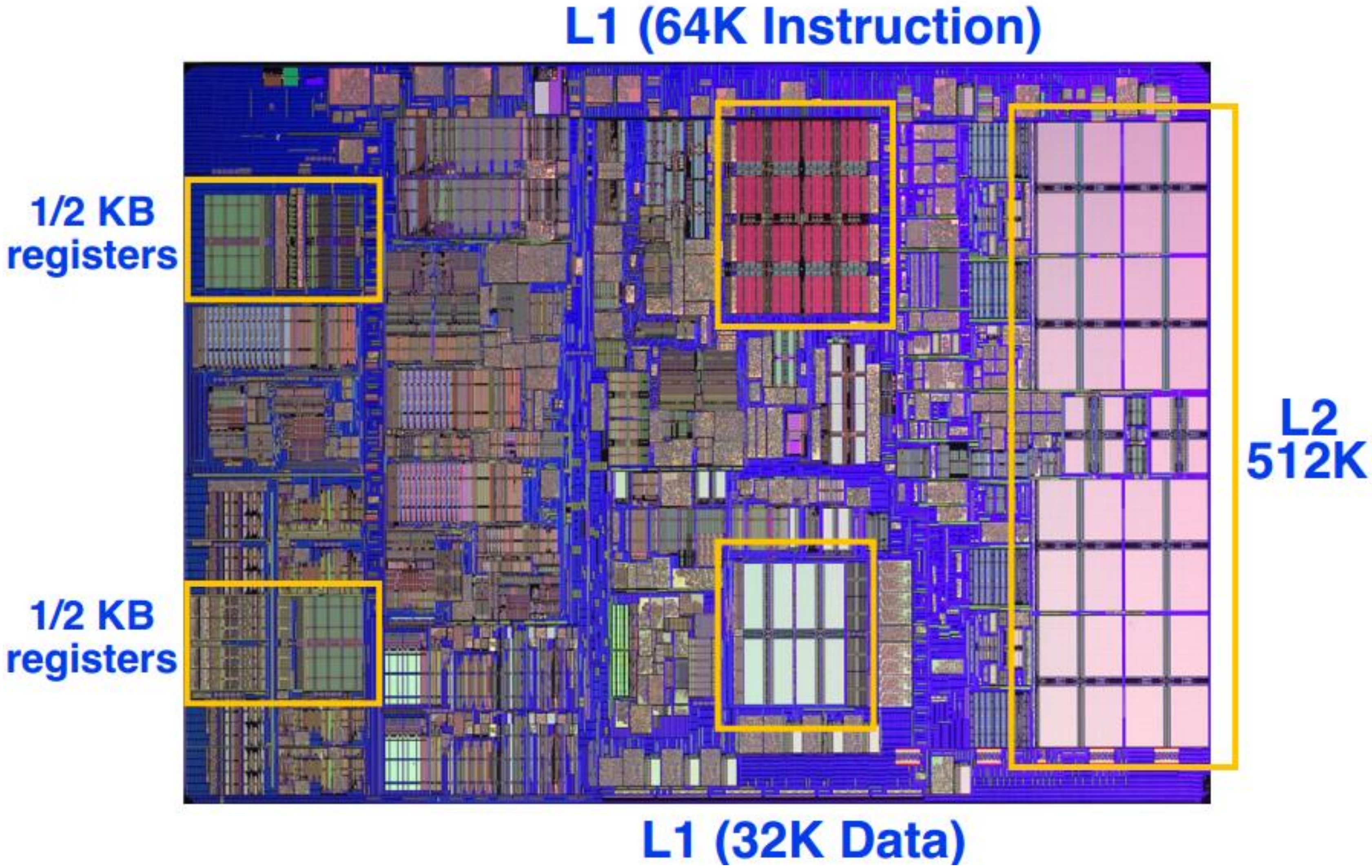
Definition

- First level of memory hierarchy after registers
- Any form of storage that bufferes temporarily data
  - OS buffer cache, name cache, Web cache, ...
- Designed based on the principle of locality
  - Temporal locality: Accessed item will be accessed again in the near future
  - Spatial locality: Consecutive memory accesses follow a sequential pattern, references separated by unit stride

Caches on RISC



PowerPC 970 (G5): All caches on-chip



Locality

Spatial locality

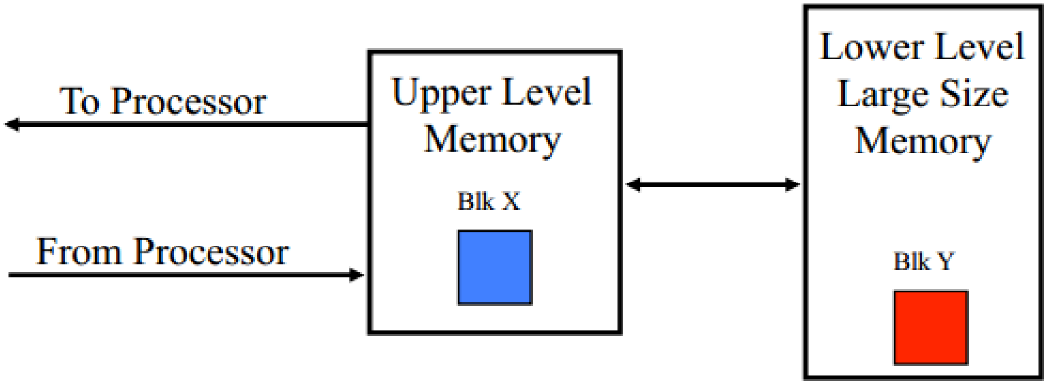
- Appears due to iterative execution and linear data access patterns
- Exploited by using larger block sizes – data to be used prefetched with block
- Exploited by data and code transformations by the compiler
- Exploited by unit-stride prefetching mechanisms and policies

Temporal locality

- Appears due to iterative execution and data reuse
- Exploited by caches, through which data is reused
- Working set: data that needs to be kept cached in a window of time to maximize locality
- Reuse distance: number of blocks of memory accessed between two consecutive accesses to same block

Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level
  - Hit Rate: the fraction of memory accesses found in the upper level
  - Hit Time: Time to access the upper level which consists of
    - Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +
    - Time to deliver the block to the upper level
- Hit Time << Miss Penalty (=500 instructions on 21264!)



Cache Measures

- Hit rate: fraction found in that level
  - So high that usually talk about Miss rate = 1 - Hit rate
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to AMAT in memory
- AMAT = Hit time + Miss rate x Miss penalty (ns or clocks)

- Miss penalty: time to supply a missed block from lower level, including any CPU-visible delays to save replaced write-back data to make room in upper level cache. {“All active caches are full”}
  - access time: time to lower level = *f* (latency to lower level)
  - transfer time: time to transfer block = *f* (BW between upper & lower levels)
  - replacement time: time to make upper-level room for new block, if all active caches are full

Average Memory Access Time (AMAT)

AMAT components

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

$$\text{CPU time} = IC \times \left(CPI_{execution} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU time} = IC \times \left(CPI_{execution} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

Assuming that cache hits do not stall the machine!

An example

- Assumption on computer A
  - CPI = 1.0 when all memory accesses hit
  - Data accesses are only loads and stores (explain 50% of insts.)
  - Miss penalty: 25 cc
  - Miss rate: 2%
- Compute the speedup of computer B, for which all cache accesses are hit

$$exectime_B = (CPUcc + MemStallcc) \times \textit{Clock cycle time}$$
$$= (IC \times CPI + 0) \times cct = IC \times 1.0 \times \textit{Clock cycle time}$$

$$MemStallcc_A = IC \times \frac{MemAccess}{Instruction} \times MissRate \times MissPenalty$$
$$= IC \times (1 + 0.5) \times 0.02 \times 25 = IC \times 0.75$$

$$exectime_A = (CPUcc + MemStallcc) \times \textit{Clock cycle time}$$
$$= (IC \times CPI + IC \times 0.75) \times \textit{Clock cycle time}$$
$$= IC \times 1.75 \times \textit{Clock cycle time}$$

4 Questions for Memory Hierarchy

For a given level of the memory hierarchy

- Q1: Where can a block be placed in the upper level? (Block placement)
- Q2: How is a block found if it is in the upper level? (Block identification)
- Q3: Which block should be replaced on a miss? (Block replacement)
- Q4: What happens on a write? (Write strategy)

Q1: Where to Place Blocks?

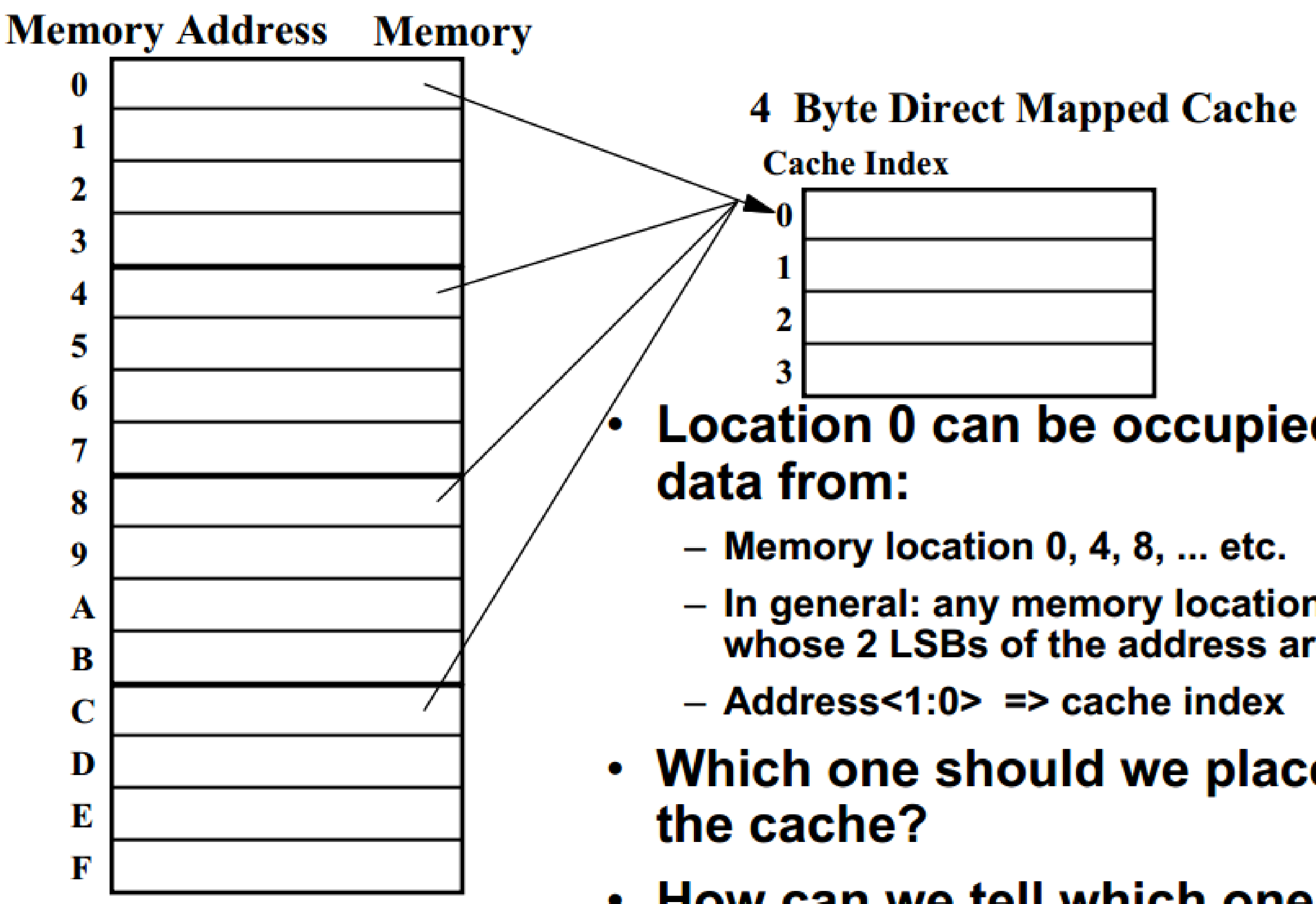
- Jargon: Each address of a memory location is partitioned into:
  - block address
    - tag
    - index
  - block offset

Block address		Block offset
Tag	Index	

Fig. C.3

Simplest Cache: Direct Mapped

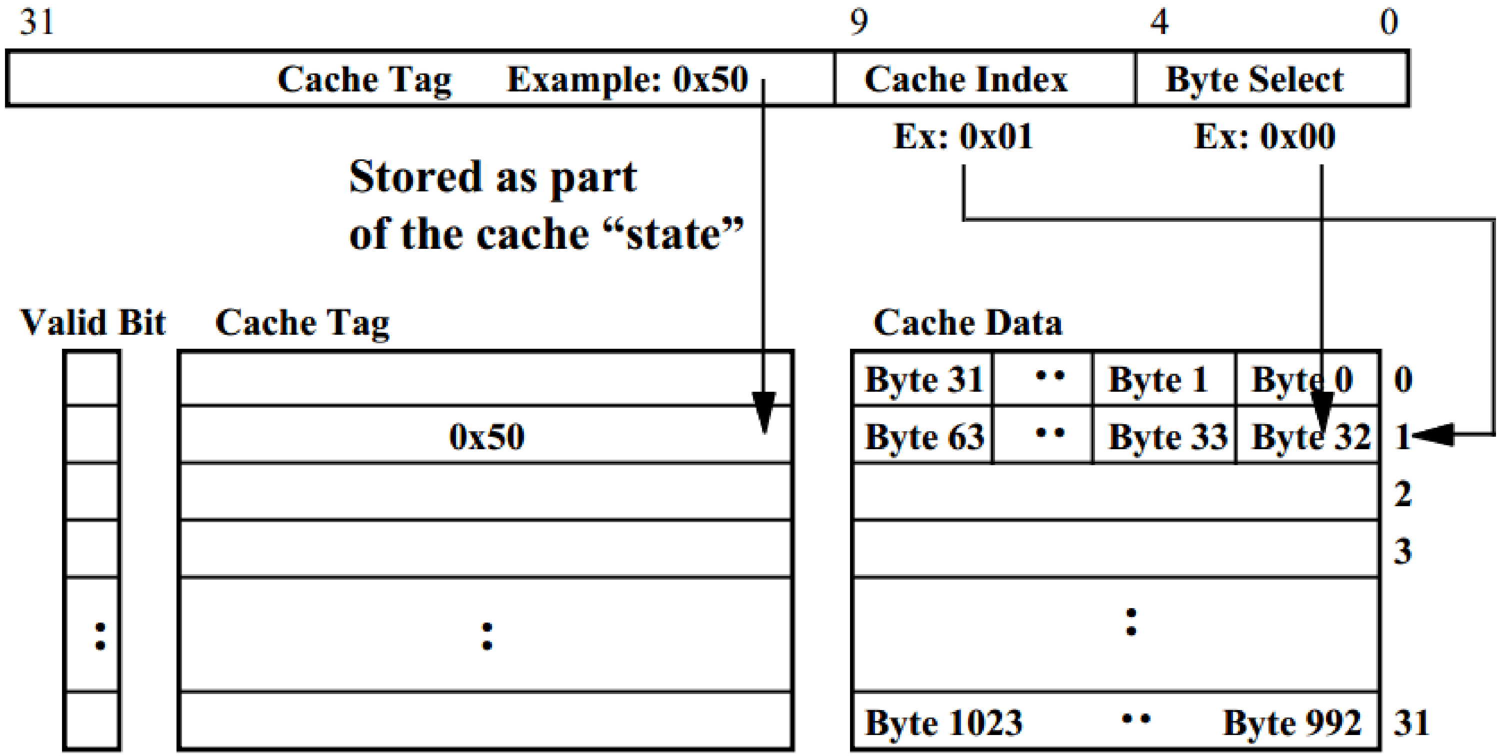
Use Index in Address to find Cache Location



- **Location 0 can be occupied by data from:**
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- **Which one should we place in the cache?**
- **How can we tell which one is in the cache?**

1 KB Direct Mapped Cache, 32B blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



Direct Mapped Cache

Advantages

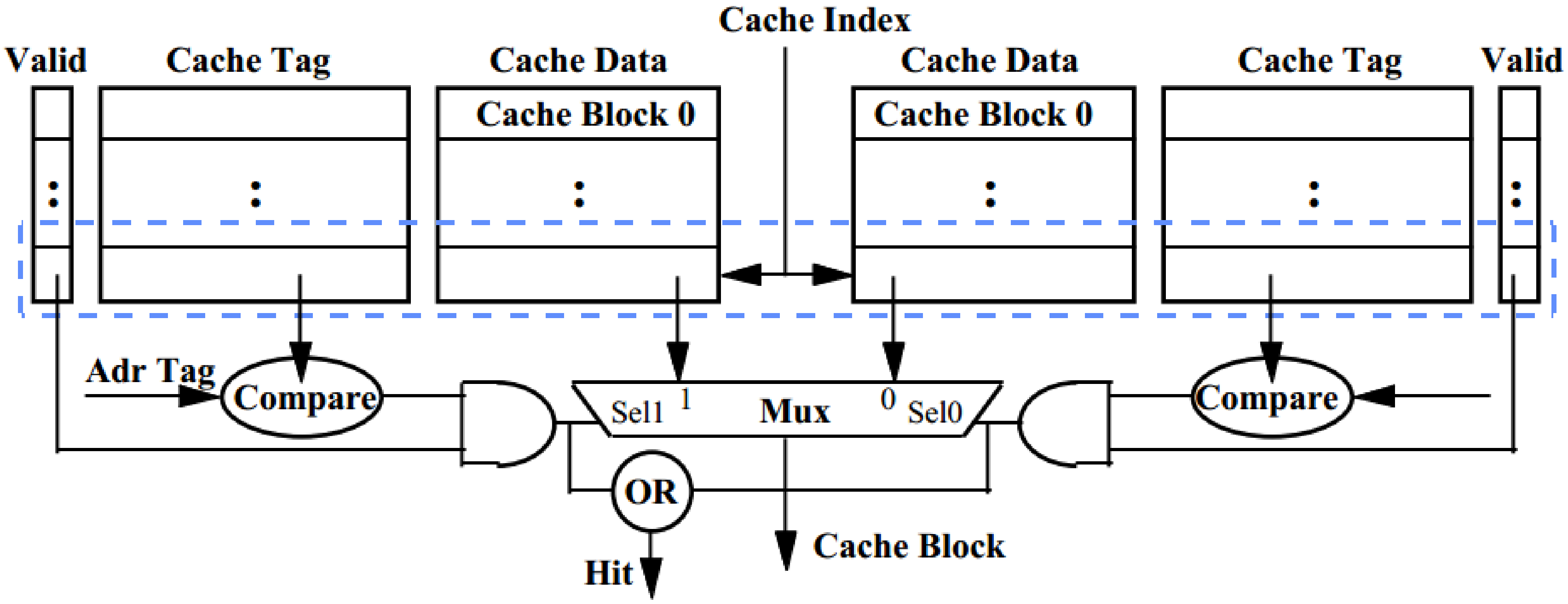
- Simple, low complexity, low power consumption
- Fast hit time
- Data available before cache determines hit or miss
  - Hit/miss check done in parallel with data retrieval

Disadvantages

- Conflicts between blocks mapped to same block in cache

Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



Two-way Set Associative Cache

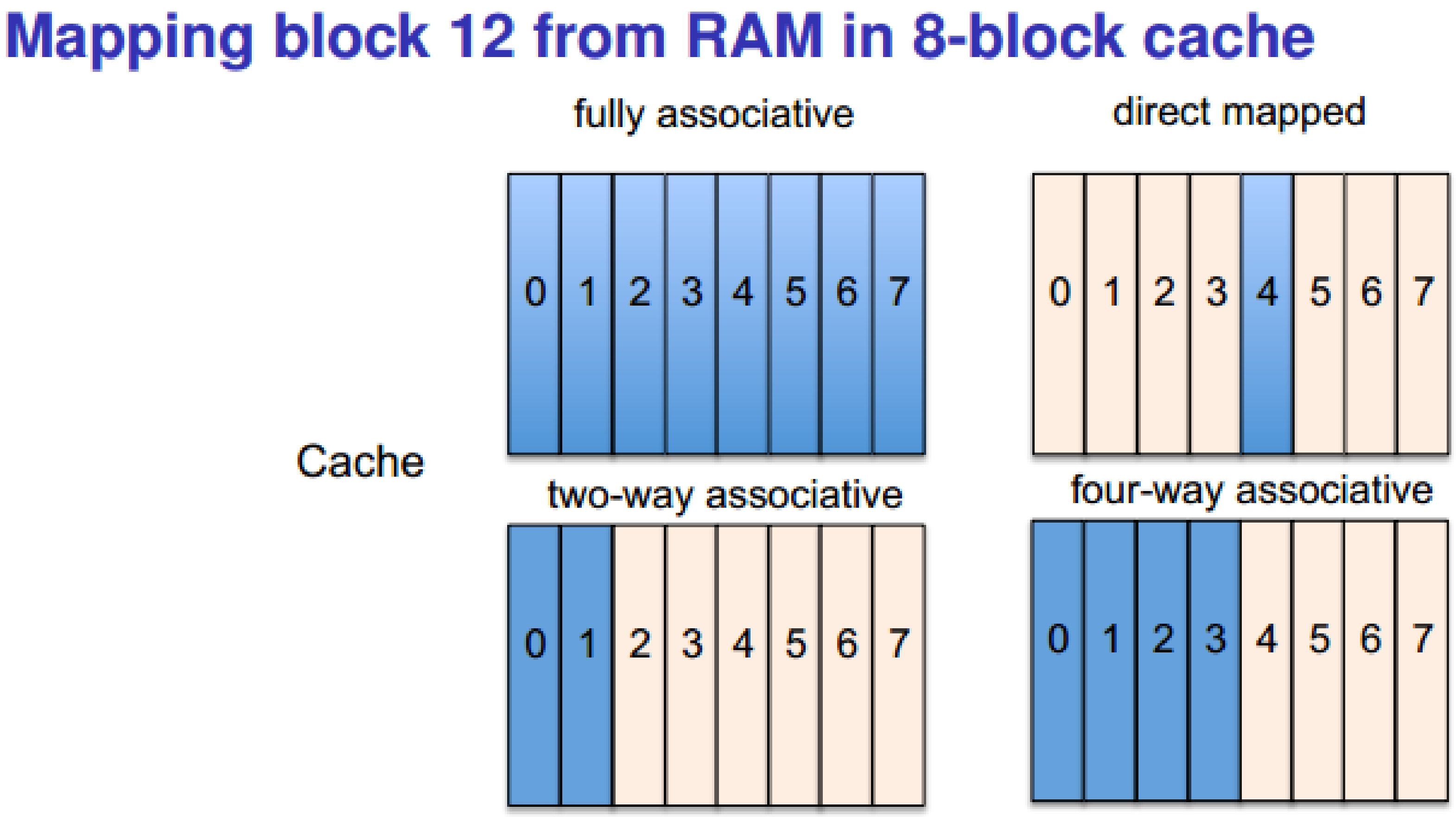
Advantages

- Choice of mapping memory block to different cache blocks in a set
  - LRU or other policies for good selection of victim blocks
- Reduction of conflicts

Disadvantages

- Increased complexity – comparators, multiplexor, parallel tag comparison
- Increased power consumption
- Increased hit time, due to comparators and multiplexor
- Data available after cache determines hit or miss

Cache Mapping Example

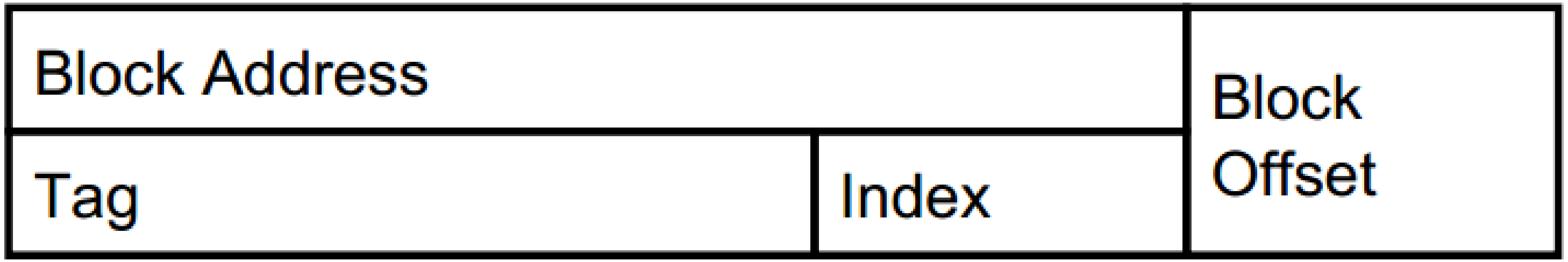


Number of sets = #Blocks / Associativity

Set/Index = (Block Address) MOD (Number of sets in cache)

Q2: How is a block found in the cache

Cache tag array



- Index points to line in data array – one block or set
- Offset points to byte in block
- Tag compared against tag field in address
- Valid bit ORed with output of tag comparator

Q3: Which block is replaced on a miss

- Easy if direct-mapped (only 1 block "1 way" per set index)
- Three common choices for set-associative cache:
  - Replace an eligible *random* block
  - Replace the least recently used (LRU) block
    - can be hard to keep track of, so often only approximated
  - Replace the oldest eligible block (First In, First Out, or FIFO)
- SPEC2000 benchmark (misses per 1000 instructions)

Set associativity

	Two-way			Four-way			Eight-Way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

(From Sussman)

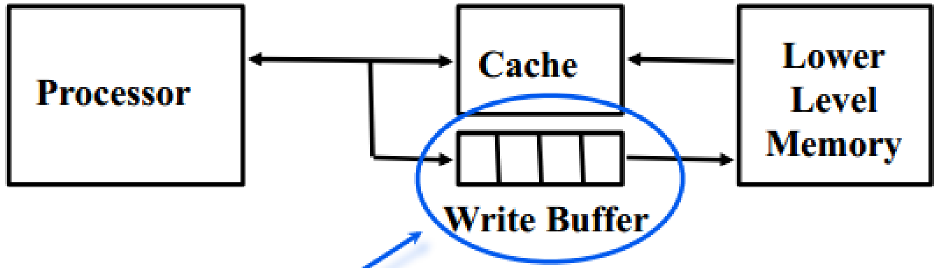
Q4: What happens on a write?

	Write-Through	Write-Back
Policy	Data word written to cache block is also written to next lower-level memory Example, instr. sw to L1$ also goes to L2$	Write new data word only to 1 cache block Update lower level just before a written block leaves cache, so not lose true value
Debugging	Easier	Harder
Can read misses force writes?	No	Yes (used to slow some reads; now write-buffer)
Do repeated writes touch lower level?	Yes, memory busier	No

Two options on a write miss:

- *Fetch line from lower-level and perform write hit ("write allocate")*
- *Perform write only to the lower-level cache ("no-write allocate")*

Write Buffers for Write-Through Caches



Holds (addresses&) data awaiting write-through to lower level memory

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Q. Can Write Buffer work with Write-Back Cache?

A. Yes. Send a block in the write-buffer on each write-back.

Write Buffer Optimization: Write Combine Buffer

- Write buffer mechanics, with merging
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there's an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer – if so, combine the new data with that entry
  - If buffer full and no address match, cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time

Recap: Average Memory Access Time (AMAT)

AMAT components

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

$$\text{CPU time} = IC \times \left(CPI_{execution} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU time} = IC \times \left(CPI_{execution} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

Assuming that cache hits do not stall the machine!

Example

UltraSPARC III

- in-order processor
- $CPI_{execution} = 1.0$
- miss penalty = 100 cycles
- miss rate = 2%
- 1.5 memory references per instruction
- 30 cache misses per 1000 instructions

$$\text{CPU time} = IC \times \left(1.0 + 0.02 \times \frac{1.5}{1} \times 100\right) \times \text{Clock cycle time} = IC \times 4 \times \text{cycle time}$$

$$\text{CPU time} = IC \times \left(1.0 + \frac{30}{1000} \times 100\right) \times \text{Clock cycle time} = IC \times 4 \times \text{cycle time}$$

Example

UltraSPARC III

- Cache miss latency increases execution time by 4*x*
- Higher clock rates imply more clock cycles wasted due to miss penalty
  - Higher relative impact of cache on performance
- HW/SW cache-conscious optimizations attempt reduce AMAT
- Performance depends on both clock cycle and AMAT – trade-off

Example

Direct-mapped vs. set-associative cache

- 1 GHz processor
- $CPI_{execution} = 2.0$
- 64 KB caches with 64-byte blocks
- 1.5 memory references per instruction
- Direct mapped cache miss rate = 1.4%
- Set associative cache stretches clock cycle by 1.25, miss rate = 1.0%
- 75 ns miss penalty **(i.e. 75 cc or 60 cc)**
- 1 cycle hit time

$$AMAT_{direct-mapped} = 1.0 + (.014 \times 75) = 2.05ns$$
$$AMAT_{2-way} = 1.0 \times 1.25 + (.01 \times 75) = 2.00ns$$

Example

Direct-mapped vs. set-associative cache

$$\text{CPU time} = IC \times \left(CPI_{execution} + \frac{Misses}{Instruction} \times \text{miss penalty} \right) \times \text{clock cycle time}$$

$$\text{CPU time}_{direct-mapped} = IC \times (2.0 \times 1.0 + 0.014 \times 1.5 \times 75) = 3.58 \times IC$$

$$\text{CPU time}_{two-way} = IC \times (2.0 \times 1.25 + 0.01 \times 1.5 \times 75) = 3.63 \times IC$$

- Associative cache achieves lower AMAT than direct-mapped cache
- Direct-mapped cache achieves higher performance than associative cache

Why? In this example common case (hits) are faster for Direct-mapped cache.

Overlapping memory latency in OOO processors

Miss penalty in OOO

- Processor can execute instructions while cache miss is pending
- Processors can execute instructions also while cache hit is pending
- Hard to attribute stall cycles to instructions
  - Stall cycle is any cycle where at least one instruction does not commit
  - First

$$\frac{\text{Memory stall cycles}}{\text{instruction}} = \frac{\text{Misses}}{\text{instruction}} \times (\text{Total miss latency} - \text{overlapped miss latency})$$