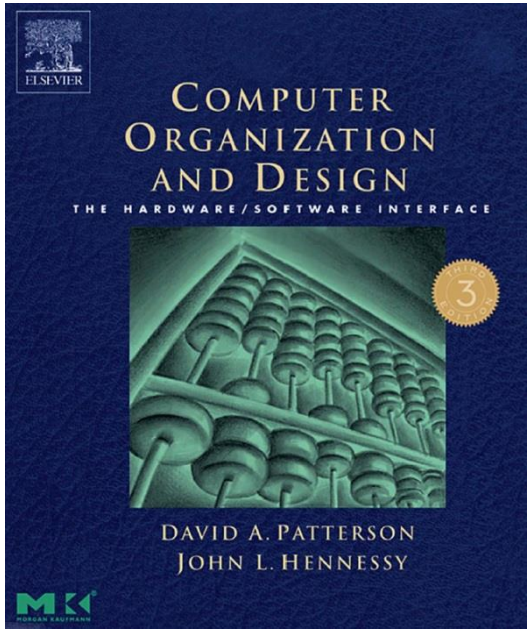
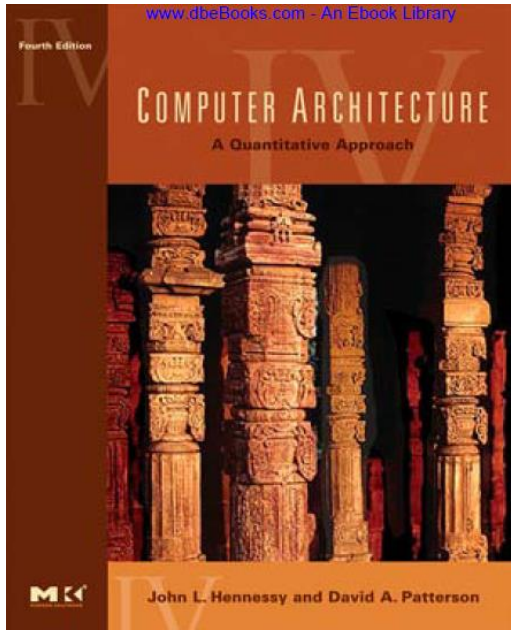
Memory Hierarchies

prepared and Instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University





Motivation

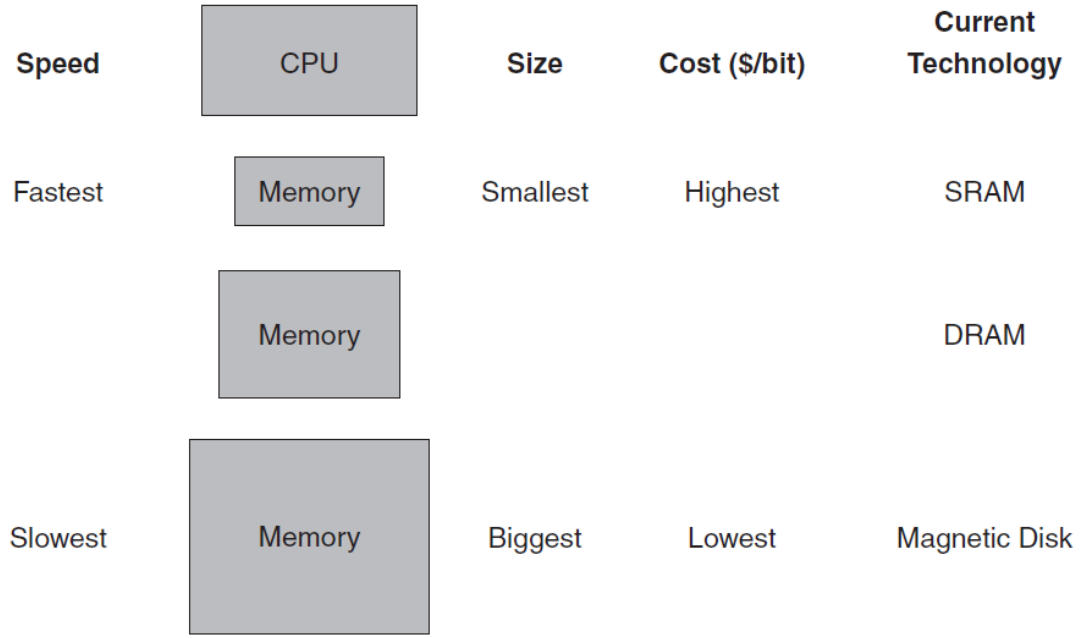
- We assumed that the **entire memory space** (2^{32} bytes) is accessed in IF stage (instruction MEM) and MEM stage (data MEM) in **1 CLK cycle**.
- In reality it takes **100s cycles**!
- Needless to mention today's 64-bit architectures, where address space is 2^{64} bytes
- Question: How to bridge the gap?
- Answer: Use **Memory Hierarchy**, where small is fast and big is slow.

Principle of Locality

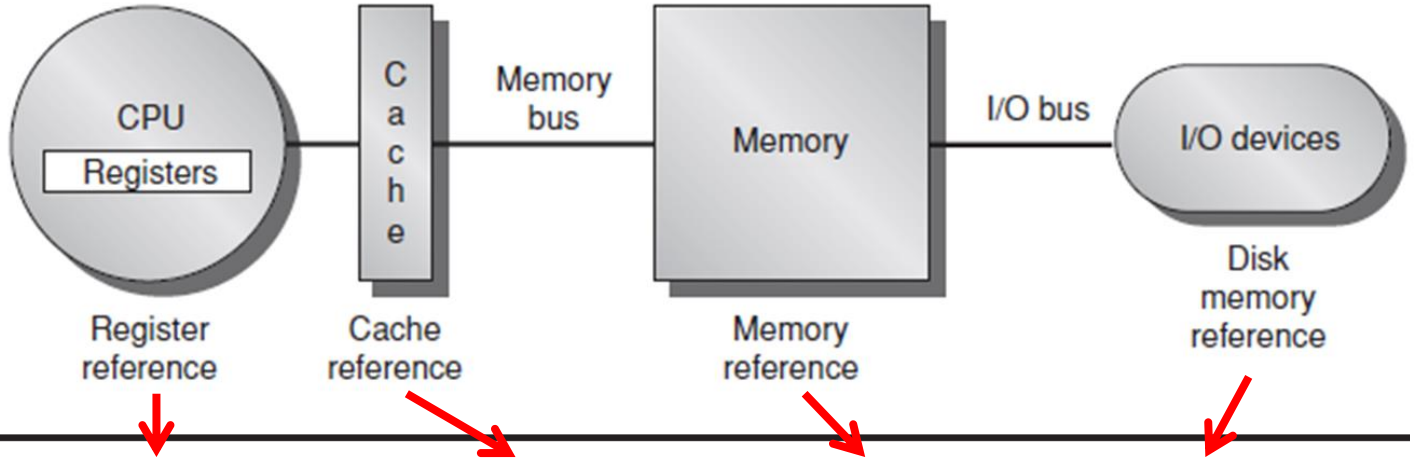
- **Temporal locality** (time): If an item is referenced, it will tend to be referenced again soon.
- **Spatial locality** (space): If an item is referenced, items whose addresses are close will tend to be referenced soon.
- locality in programs
  - loops - temporal
  - instructions are usually accessed sequentially - spatial
  - Data access of array - spatial

Memory Hierarchy

- The memory system is organized as a hierarchy
  - Level closer to the CPU is a subset of any further away.
  - All the data is stored at the lowest level.



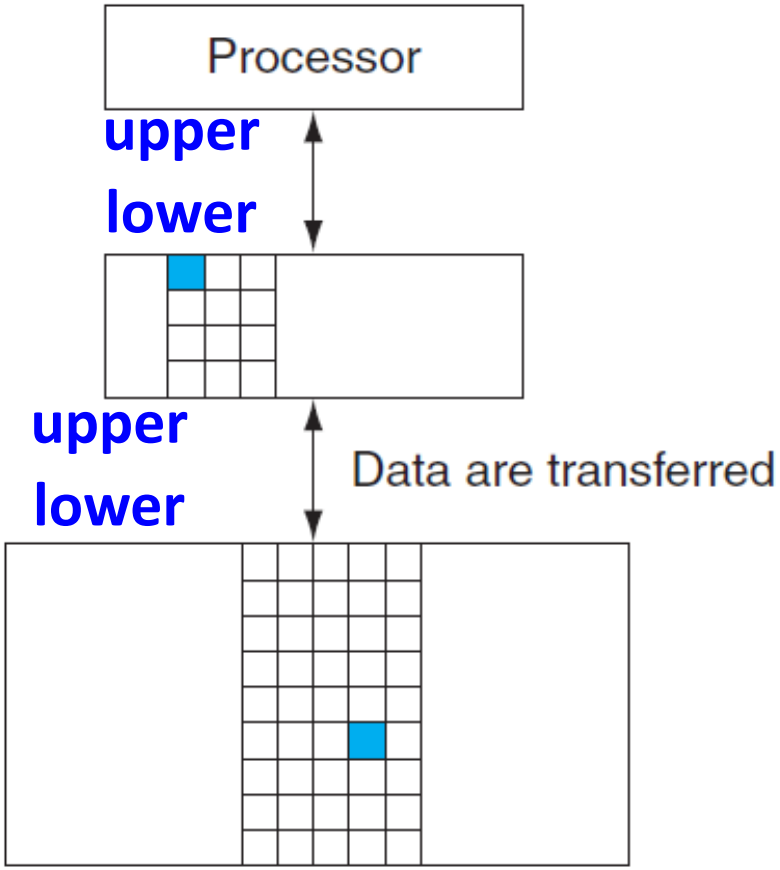
- **Hierarchical implementation makes the illusion of a memory size as the largest, but can be accessed as the fastest.**



Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape

HW: study this chart.

Hit and Miss



The unit within each level is called a **block**.

We transfer an entire block when we copy something between levels.

Hit rate (hit ratio): Fraction of memory accesses found in the upper level. **Miss rate** = 1 – hit rate.

Hit time: the time required to access a level of the memory hierarchy.

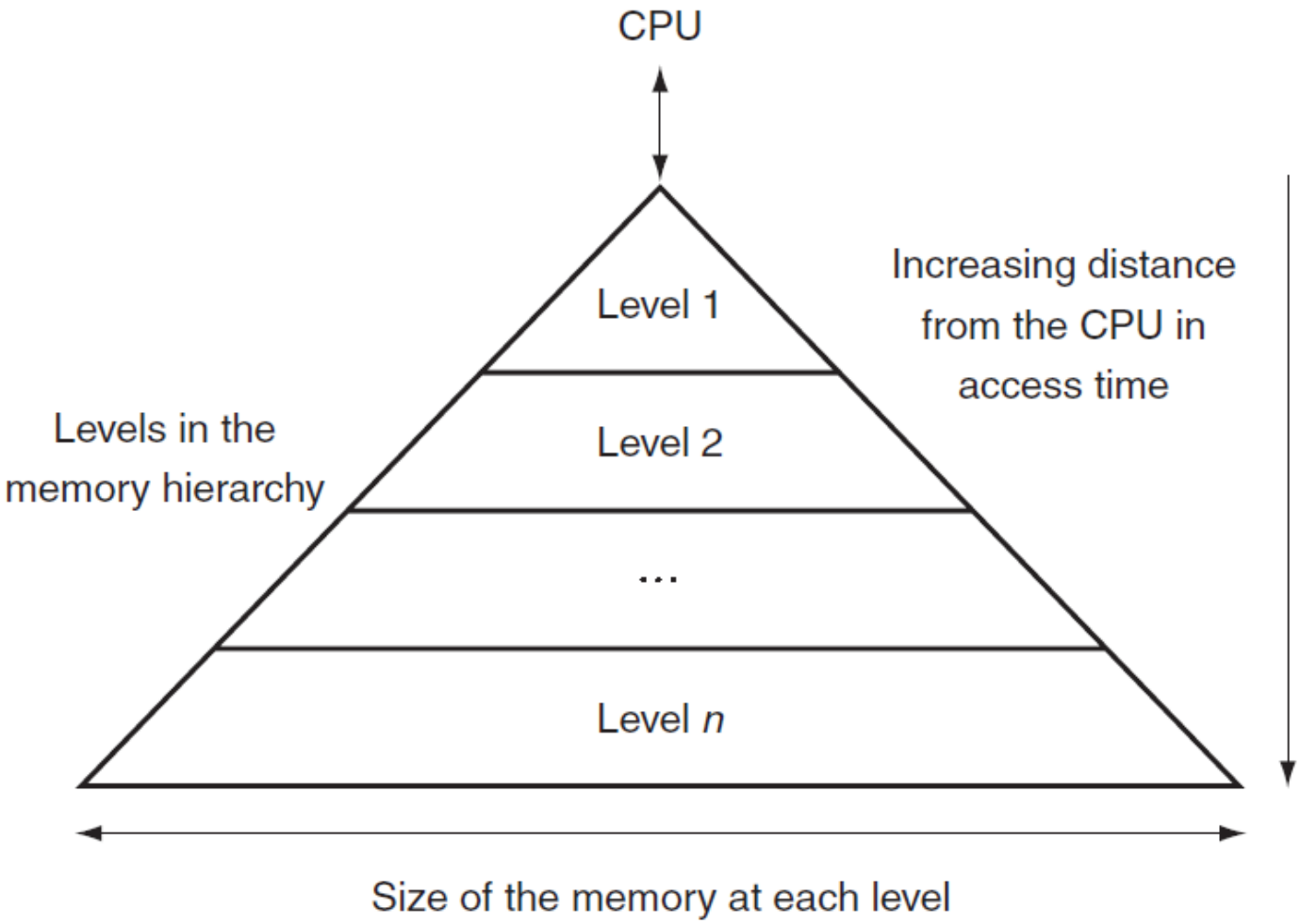
- Includes the time needed to determine whether hit or miss.

Miss penalty: the time required to fetch a block into the memory hierarchy from the lower level.

- Includes the time to access the block, transmit it from the lower level, and insert it in the upper level.

The memory system affects many other aspects of a computer:

- How the **operating system** manages memory and I/O
- How **compilers** generate code
- How **applications** use the computer

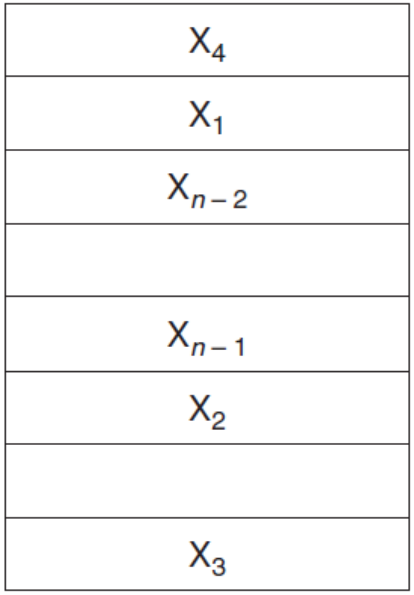


Hierarchy enables CPU **access time** determined by **level 1** and yet have **memory** as large as **level *n***.

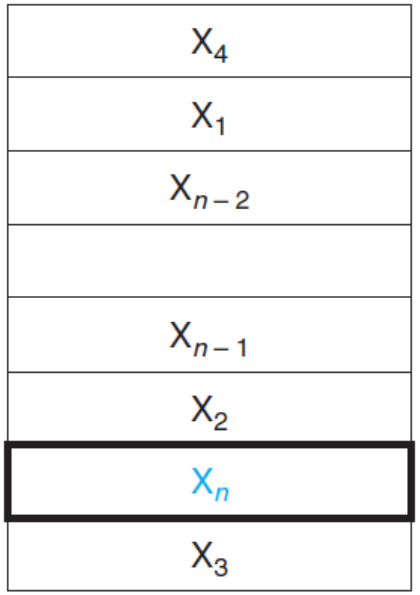
Requesting data from the cache

CPU requests word X_n that is **not** in the cache

Before reference to X_n



After reference to X_n



Two questions :

- How to know whether a data item is in the cache?
- If it does, how to find it?

Direct-Mapped Cache

Each memory location is mapped to **one** cache location

Mapping between addresses and cache locations:

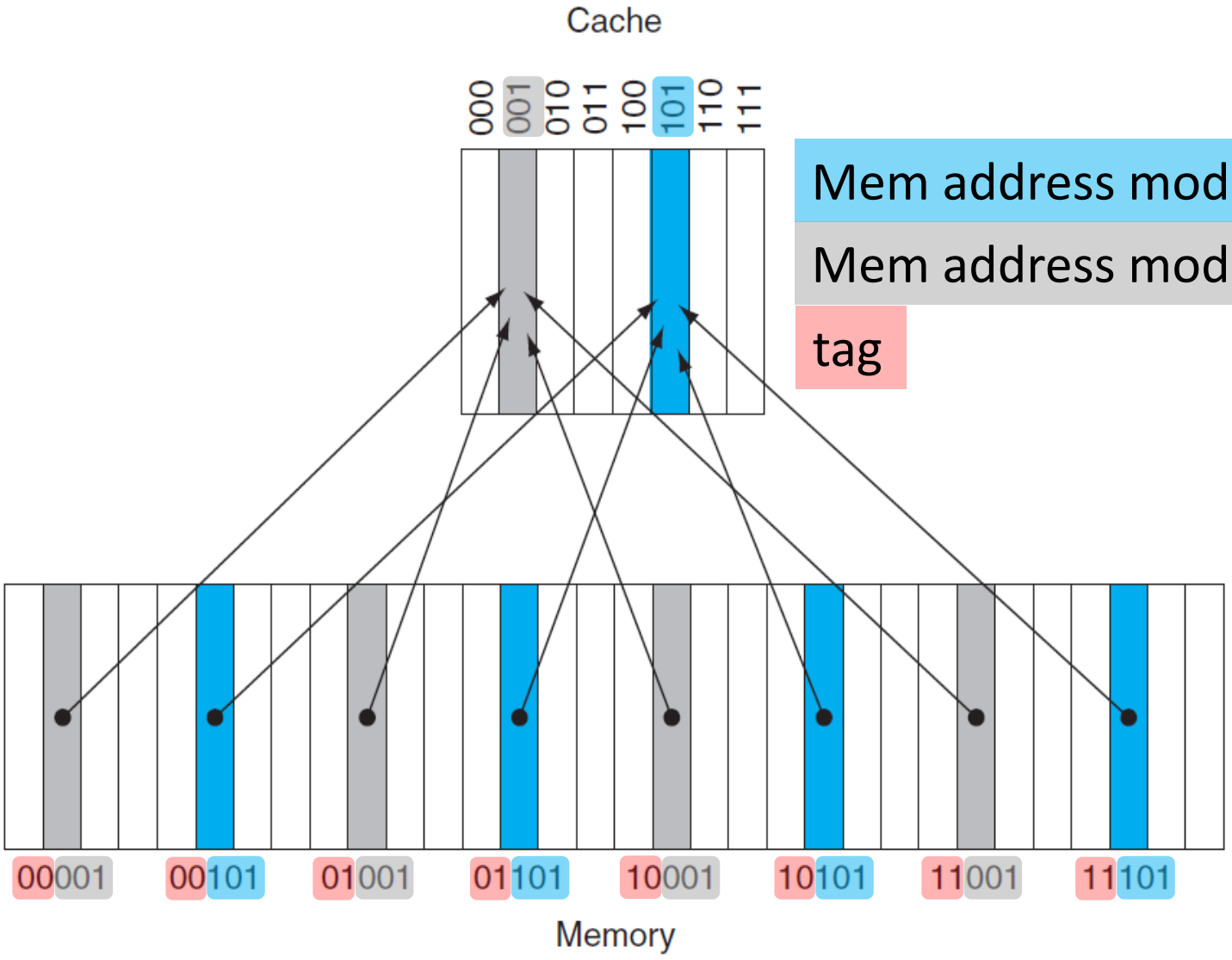
(Block address in Mem) % (# of blocks in cache)

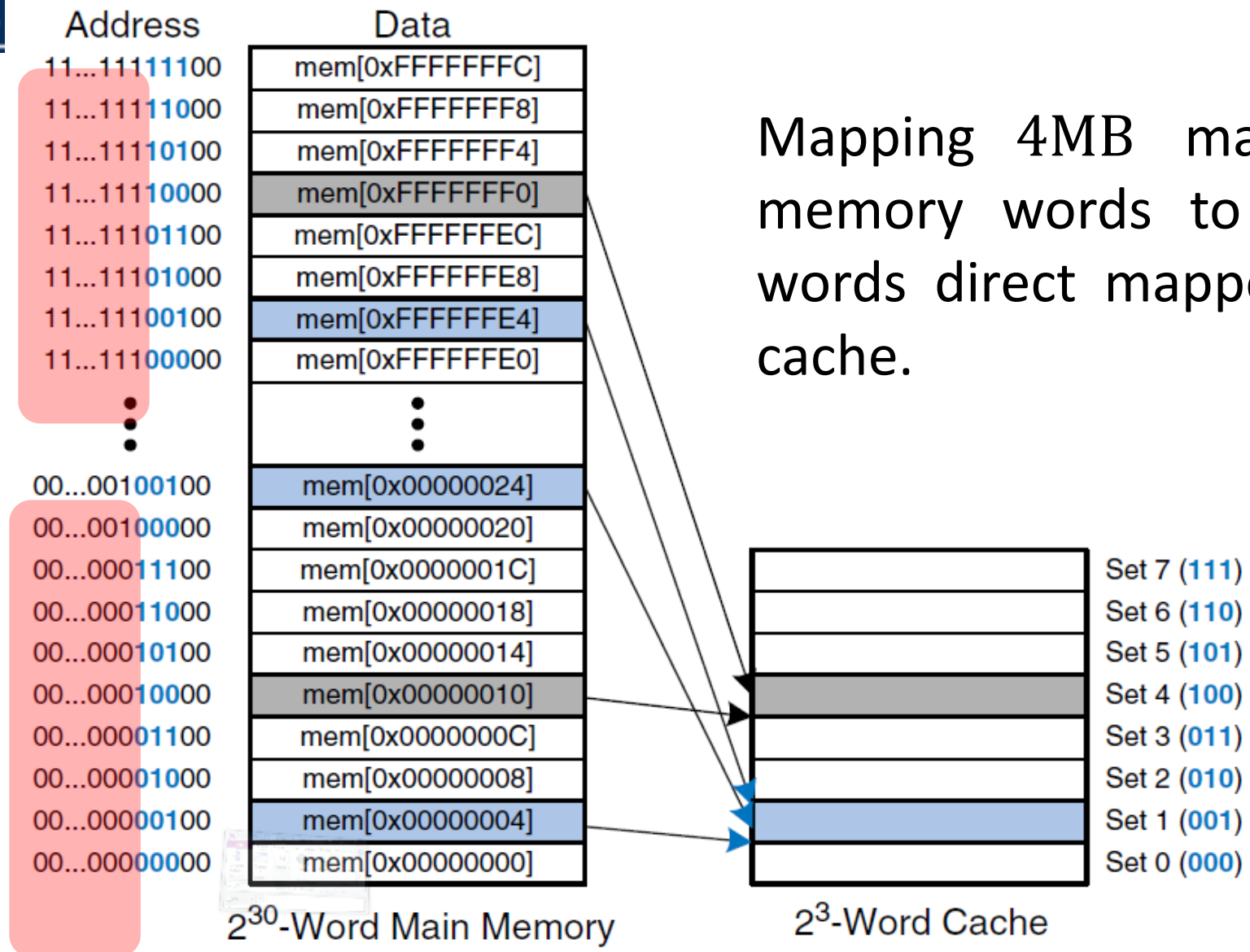
Modulo is **$\log_2$(cache size in blocks)** LSBs of address.

Cache is accessed directly with the LSBs of requested memory address.

Problem: this is a **many-to-one** mapping!

Solution: Use the MSBs of address as a **tag** attached to the cache entry.


Cache
000
001
010
011
100
101
110
111
Mem address mod 8 = 101
Mem address mod 8 = 001
tag
00001
00101
01001
01101
10001
10101
11001
11101
Memory



Mapping 4MB main memory words to 8 words direct mapped cache.

HW: ensure you understand the mapping and tags.

Some of the cache entries may still be empty (never accessed, garbage data and tag).

Need to know that the tag should be ignored for such entries.

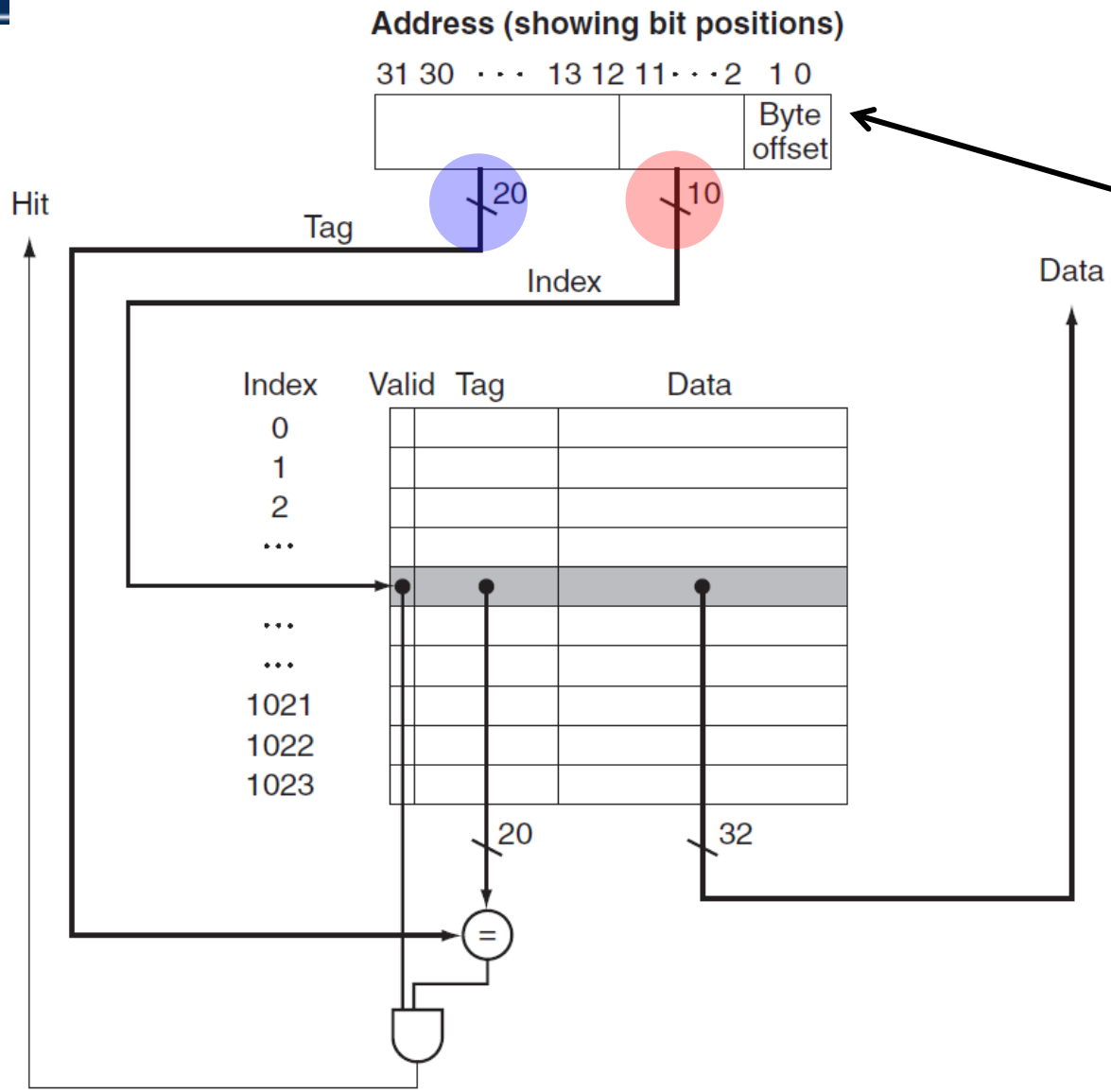
Add a **valid bit** to indicate whether an entry contains a valid address (data) of the main memory.

Cache Access Sequence

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (7.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (7.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (7.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (7.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (7.6f)	$(10010_{two} \bmod 8) = 010_{two}$

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

HW: complete.



Referenced address is divided into

cache **index** select the block

tag field compared with the value of the tag field of the cache

Cache Size

Example: Find the total size of 16 KB direct-mapped cache with 4-word blocks, assuming a 32-bit address.

Cache size = 16KB = 4K words $= 2^{12}$ words.
Block size = 4 words = 2^2 words.
Total of 2^{10} blocks.

Block's data size = 4 x 32 = 128 bits
Block's tag size = 32 - 10 - 2 - 2 bits + valid bit.
Total cache size:
2^{10} x (128 + (32 - 10 - 2 - 2) + 1) = 147 Kbits = 18.4 KB

It is 1.15 times than data storage alone.
Convention is to count only the size of the data.

HW: go over the example.

Example: Given 64-blocks cache of 16-byte block size. Find block location that byte 1200 of Mem maps to.

Cache block location

=(Mem block address)%(#blocks in cache)

Mem block address

$= \lfloor$Mem byte address/bytes per block$\rfloor$

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

Block contains all the bytes

From: $\lfloor$Mem byte address/bytes per block$\rfloor \times$ bytes per block $= 75 \times 16 = 1200$

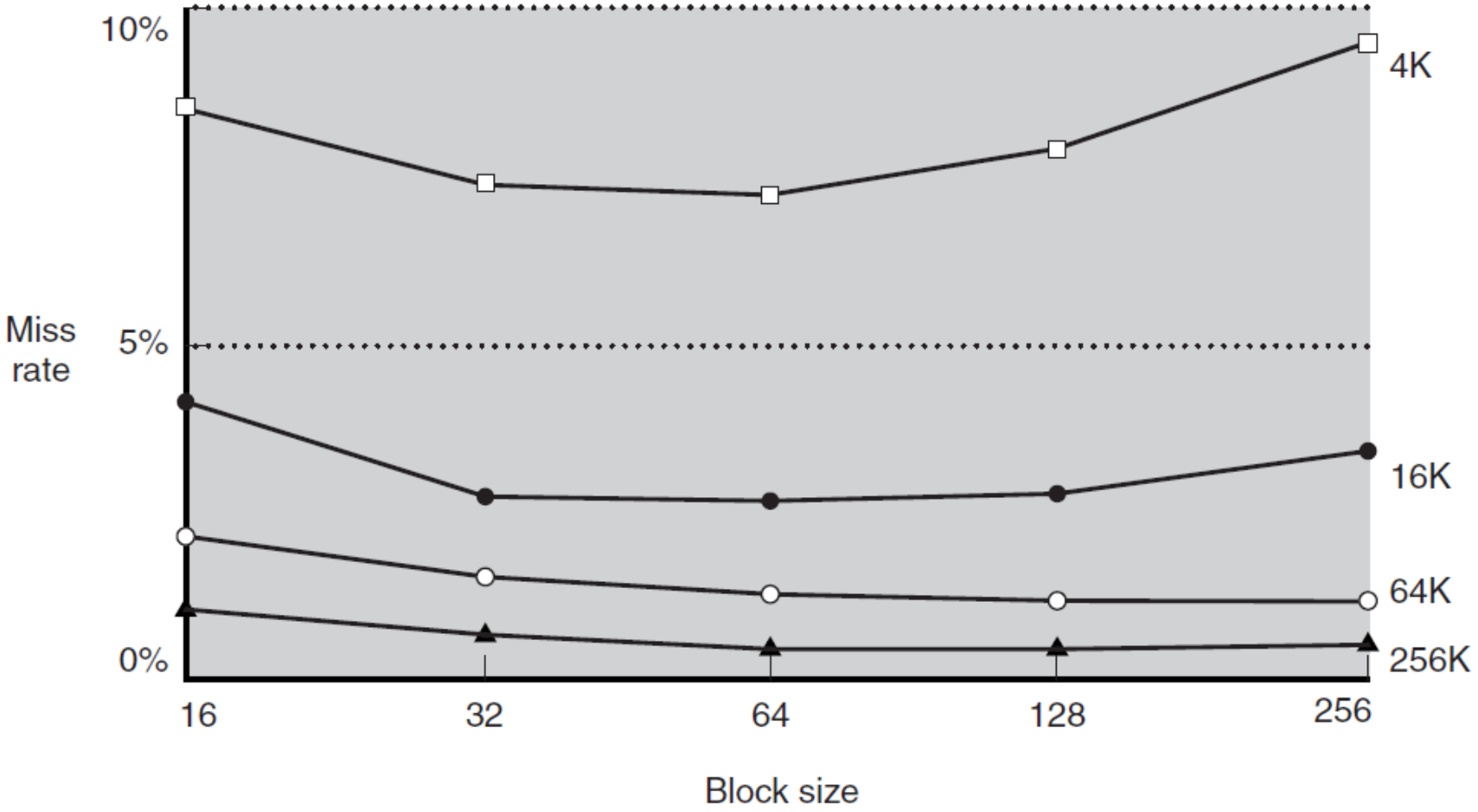
To: $(\lfloor$Mem byte address/bytes per block$\rfloor + 1) \times$ bytes per block $- 1 = 76 \times 16 - 1 = 1215$

Maps to block # $(75\ \%\ 64) = 11$ (bytes $1200 \div 1215$)

Block Size Implications

- Larger blocks exploit spatial locality to lower miss rates.
- Block increase will eventually increase miss rate
- Spatial locality among the words in a block decreases with a very large block.
  - The number of blocks held in the cache will become small.
  - There will be a big competition for those blocks.
  - A block will be thrown out of the cache before most of its words are accessed.

Miss rate versus block size



Miss cost (penalty) increases with block size.

- Time required to fetch the block and load into cache.

Fetch time has two parts:

- Latency to access first word, and
- Transfer time for the rest block.

Transfer time (miss penalty) increases with block size.

Miss penalty increase overwhelms miss rate decrease for large blocks, thus decreasing cache performance.

Handling Cache Misses

Misses require extra work of CPU's control unit and a separate **cache controller**.

Miss stalls the pipeline while waiting for memory.

Steps taken on an **cache miss (I-Cache & D-Cache)**:

1. Send to the memory the address of miss **(I$ or D$)**.
2. Instruct main memory to perform a read and wait for the memory to complete access.
3. Write the cache entry: data + tag (address's MSBs) + turn valid bit on.
4. Restart by re-fetching the instruction causing the miss, this time finding (instruction or data) in the cache.

Handling Writes

Write hit writes into cache (data), making main memory **inconsistent**.

Can always write into both main memory and cache, called **write-through**.

Write miss first fetches block from memory into cache. Then overwrite the word in cache's block and also write to main memory.

Write-through is simple but perform badly. Writing to main memory takes 100s clock cycles.

If 10% stores and **CPI** without misses was 1.0, new CPI is 1.0 + 100 x 10% = 11, **10x slowdown**!

Speeding Up

1. **Write buffer** (queue) holding data to write in memory. The CPU can continue. When memory write completes, queue's entry is freed.

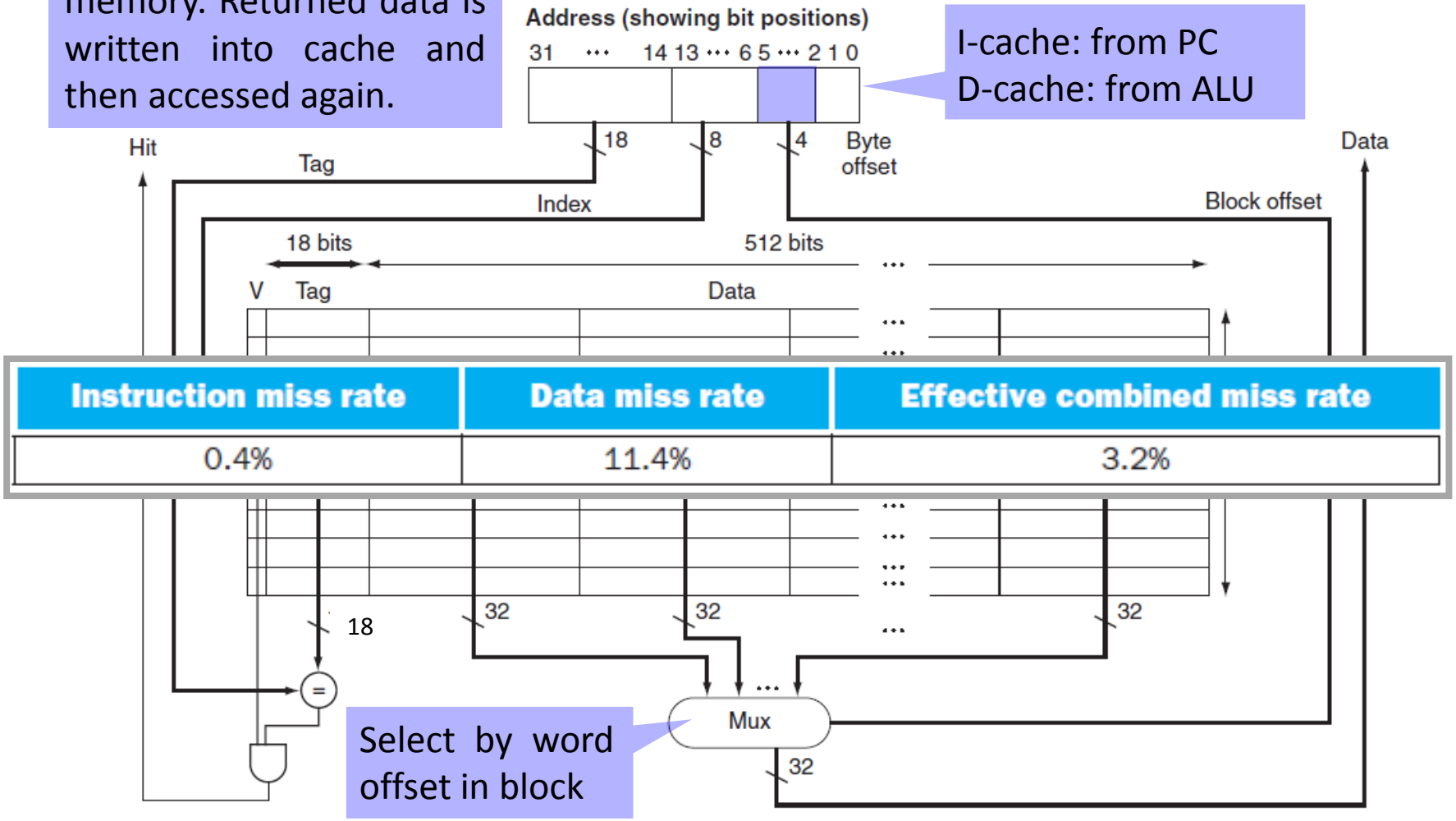
Full queue at CPU write **stalls** the CPU until queue has empty entry.

2. **Write-back** writes the new only in cache. Modified block is written to main memory when it is replaced (due to miss).

Write-back improves performance for extensive writes. Implementation is more complex than **write-through**.

Cache Example (Data and Instruction)

Miss sends address to memory. Returned data is written into cache and then accessed again.



Main Memory Design Considerations

Main memory (DRAM) is designed for density rather than access time.

Miss penalty is reduced by increasing bandwidth from memory to cache.

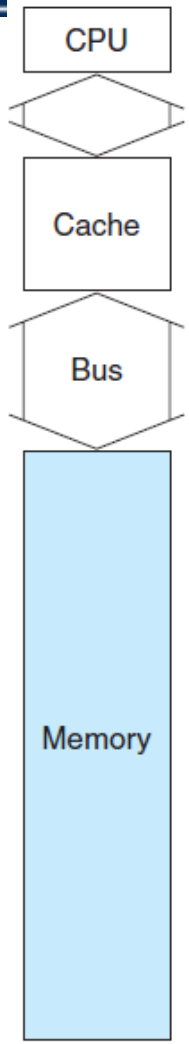
Bus clock 10x slower than CPU clock. Assume

- 1 memory bus cycle to send address
- 15 memory bus cycles for each DRAM access initiated
- 1 memory bus cycle to send data word

For 4-word block and 1-word DRAM bank width

$\text{miss penalty} = 1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles.

Byte transfer per **bus** cycle $= (4 \times 4)/65 = 0.25.$



a. One-word-wide memory organization

$\text{Miss penalty} = 1 + 1 \times 15 + 1 = 17$ cycles. Byte transfer $= (4 \times 4)/17 = 0.94/\text{cycle}$. **Wide bus (area) and MUX (latency) are expensive.**

$\text{Miss penalty} = 1 + 1 \times 15 + 4 \times 1 = 20$ cycles. Byte transfer $= 0.8/\text{cycle}$.

Cache Performance Improvements

Two techniques to reduce miss rate:

- **Associativity:** Reducing probability that different memory blocks will contend for same cache location.
- **Multilevel caching:** Adding levels to hierarchy (L1, L2, L3…).

CPU Time Definitions

CPU time = (CPU execution cycles + Memory-stall cycles) x Cycle time

Memory-stall cycles = Read-stall cycles + Write-stall cycles

Read-stall cycles = Reads/Program x Read miss rate x Read miss penalty

Write-stall cycles = Writes/Program x Write miss rate x Write miss penalty + Write buffer stall cycles (write-through)

Write buffer term is complex, usually ignored.

Write-back has additional stalls arising from writing cache block back to memory upon replacement.

Write-through has same read and write miss penalties (write buffer ignored).

Miss penalty:

Memory-stall clock cycles (simplified) =

Memory accesses/Program x Miss rate x Miss penalty =

Instructions/Program x Misses/Instruction x Miss penalty

Example: impact of an ideal cache

A program is running I instructions. 2% instruction cache miss, 4% data cache miss, 2 CPI without any memory stalls, and 100 cycles penalty for all misses.

How faster is a processor with a never missed cache?

Instruction miss cycles = $I \times 2\% \times 100 = 2.0 \times I$

With 36% loads and stores,
Data miss cycles = $I \times 36\% \times 4\% \times 100 = 1.44 \times I$

CPI with memory stalls = $2.0 + (2.0 + 1.44) = 5.44$

$$\text{Speedup} = \text{CPI}_{\text{stall}} / \text{CPI}_{\text{perfect}} = 5.44/2.0 = 2.77$$

Example: Accelerating processor but not memory. Memory stalls time fraction is increased.

CPI reduced from 2.0 to 1.0 (deeper pipeline).
System with cache misses $\text{CPI} = 1.0 + 3.44 = 4.44$.
System with perfect cache $4.44/1.0 = 4.44$ faster.

Execution time spent on memory stalls increases from $3.44/5.44 = 63\%$ to $3.44/4.44 = 77\%$.

CPU 2X faster but memory bus not.
$\text{CPI}_{\text{stall}} = 2 + 2\% \times 200 + 36\% \times 4\% \times 20 = 8.88$

$\text{Perf}_{\text{slow}}/\text{Perf}_{\text{fast}} = 5.44/(8.88 \times 1/2) = 1.22$,
rather than 2X expectation.

Cache Associativity

Direct map places a block in a **unique** location.

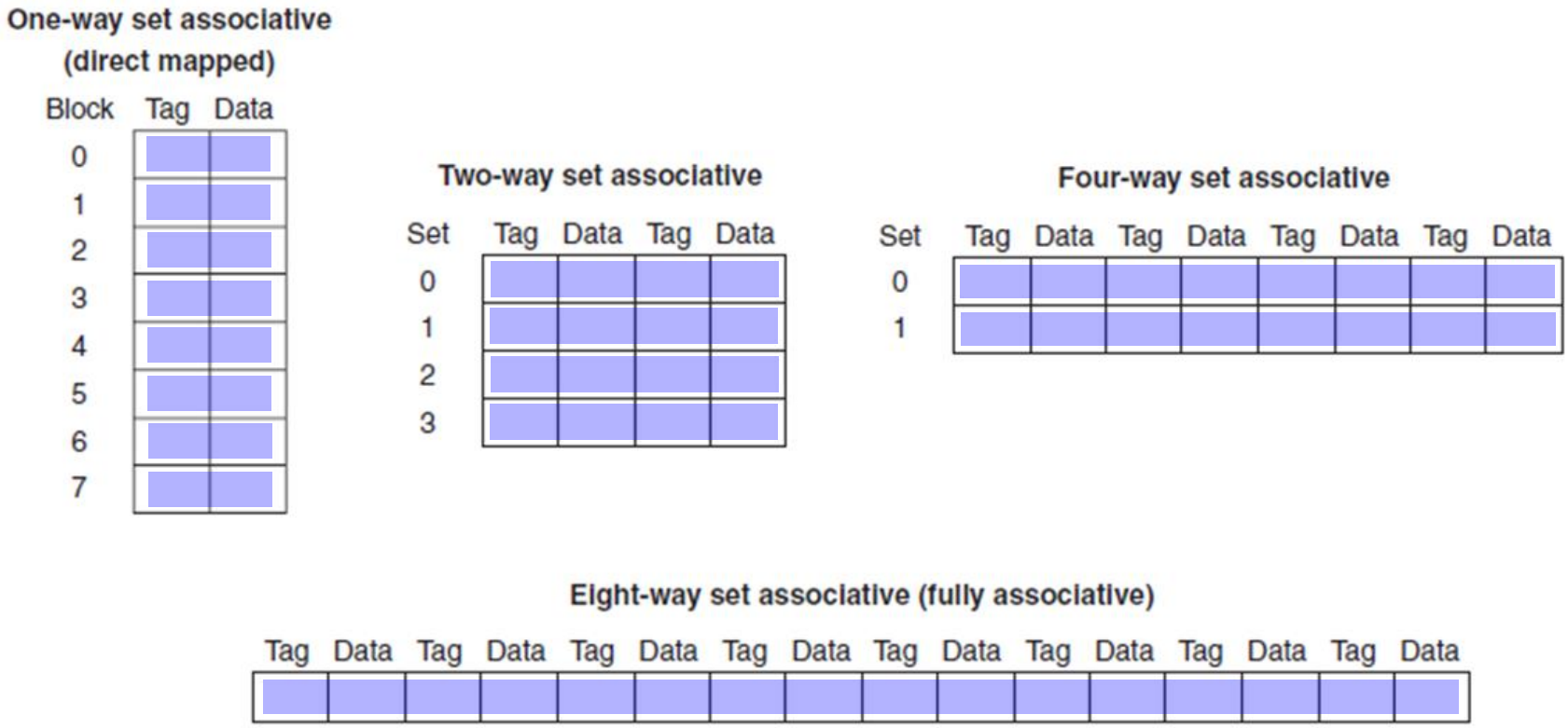
Fully associative places a block in **any** location.

- All cache's entries are searched. Expensive, done in parallel with comparator for each entry.
- Practical only for small caches.

Mid solution is ***n*-way set-associative** map.

- ***n*** locations where a block can be placed.
- Cache comprised of ***n***-blocks **sets**.
- A memory block maps to a unique **set** by index field.
- A block is placed in **any** element of that set.


Fully associative:
block 12 can go
anywhere
Direct mapped:
block 12 can go
only into block 4
(12 mod 8)
Set associative:
block 12 can go
anywhere in set 0
(12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Set 0
Set 1
Set 2
Set 3
Cache
Block frame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory



Cache size (blocks) = number of sets x associativity.

For fixed cache size, increasing associativity decreases number of sets.

HW: solve the example by yourself.

Example: Misses and associativity in caches.

Three caches comprising 4 1-word blocks.

Fully associative, two-way set associative, and direct mapped.

For the sequence of block addresses: 0, 8, 0, 6, 8, what is the number of misses for each cache?

direct mapped

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

5 misses

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

two-way set associative

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

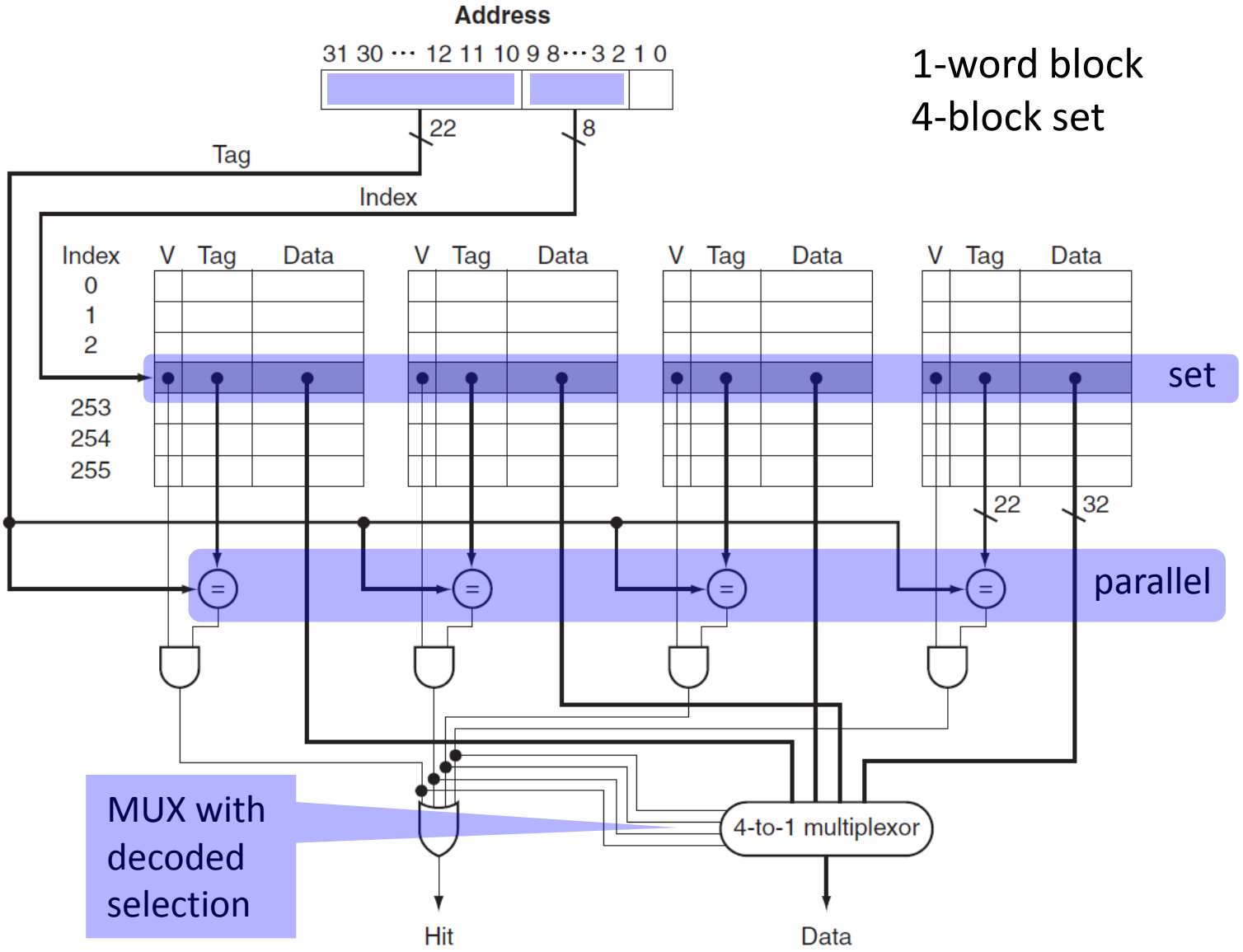
4 misses

fully associative

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

3 misses

Four-way set-associative cache



1-word block
4-block set

Which Block to Replace?

Direct-mapped maps to unique block address.

Set-associative chooses among the set's blocks.

Least Recently Used (LRU) is most commonly used.

Replace the block replaced that has been unused for longest time among the set's blocks.

Simple implementation for 2-way set-associative.

Implementation gets complex with associativity increase.

Random

- Spreads allocation uniformly.
- Blocks are randomly chosen.
- HW generates pseudorandom numbers to get reproducible behavior (useful for HW debug).
- Surprisingly, this is the best policy!

First in, first out (FIFO)

- Simple implementation.
- Use the **oldest** block entered to set.
- Fair approximates to LRU.

Multilevel Caches

Reduces miss penalty considerably.

Many CPUs support a 2^{nd}-level (L2) cache, which can be on the same die or in separate SRAMs (old days).

L2 is accessed whenever L1 misses.

If L2 hits, L1 miss penalty is the access time to L2, far shorter than access time to main memory.

If L2 misses, main memory is accessed and higher miss penalty incurs, but with far lower probability.


Example: performance of multilevel caches

Given 5GHz CPU with base CPI 1.0 if all references hit L1.

Main memory access time is 100ns, including all the miss handling.

L1 miss rate per instruction is 2%.

How faster the CPU is if we add L2 having 5ns access time for either hit or miss, which reduces miss rate to main memory to 0.5% ?

Miss penalty to main memory (memory-stall):

5GHz x 100ns = 500 cycles.

The effective CPI with L1:

Base CPI + Memory-stall cycles per instruction =

1 + 500 x 2% = 11

The effective CPI with L2:

1 + 25 x (2% - 0.5%) + (500 + 25) x 0.5% = 4

The processor with L2 is faster by:

11 / 4 = 2.8

HW: solve the example. A good certificate you are on board.

Concluding example: Given CPU of 500MHz frequency with L1 and L2 incorporated on CPU die.

Data L1 is 8KB size with 8B block size and 15% miss rate. It is direct-map, write-through with perfect buffer (no overflow).

Instruction L1 is 4KB size with 8B block size and 2% miss rate. It is direct-map.

L2 is 2GB size with 32B block size, backing both L1 caches. It is 2-way associative with miss rate 10%.

On average 50% of L2 blocks are “dirty”, namely, containing data not in main memory.

How many bits are used for index ?

L1 Data: 8Kbyte/8Byte = 1024 blocks => 10 bits

L1 Instruction: 4Kbyte/8Byte = 512 blocks => 9 bits

L2: 2MByte/32Bytes = 64K blocks = 32K sets => 15 bits

What % of memory data accesses (lw, sw) reaches main memory?

(L1 miss rate) x (L2 miss rate) = 0.15 X 0.1 = 1.5%

40% of instructions are memory access, 60% of which load and 40% store. L1 hits never cause stalls.

L2 access time is 20ns. 128-bit bus connects L2 to main memory, which access time is 0.2µs. The entire bus is used for transfer.

What is the maximal number of cycles required for main memory access? What is the sequence of events then?

Maximum cycles occur when L1 missed first, then L2 missed, then write-back takes place.

L2 access cycles: 20ns / 2ns = 10 cycles

Main memory access cycles: 0.2µs / 2µs = 100 cycles

L2 block has 32 Bytes and memory bus is 128 bits (16 Bytes), 2 bus transactions per block are required.

The first 16 bytes take 100 cycles, the next 16 bytes takes one cycle.


Getting a new block from the memory may evict a dirty block from L2, which requires write-back.

In such case the evicted block must be written into the memory, requiring a total of L2-memory transactions, yielding 2 x (100 + 1) = 202 cycles.

Summing all:
L1 miss + L2 miss + write-back = 1 + 10 + 202 = 213 cycles

What is the Average Memory Access Time (AMAT) (in cycles), including instructions and data accesses?

The weight of instruction accesses to memory is 1/(1 + 0.4), while the weight of data accesses is 0.4/(1 + 0.4). Hence $AMAT_{total} = 1/1.4\ AMAT_{inst} + 0.4/1.4\ AMAT_{data}$

For any 2-level cache system there is

AMAT = (L1 hit time) + (L1 miss rate) x (L2 hit time) + (L1 miss rate) x (L2 miss rate) x (main memory transfer time).

AMAT must account for the average % of L2 dirty blocks, namely, 50% of the blocks must be updated in main memory upon L2 miss, yielding a factor of 1.5 multiplying (100 +1) main memory access time

$AMAT_{inst}$ =

1 + 0.02 x 10 + 0.02 x 0.1 x 1.5 x (100 + 1) = 1.503

$AMAT_{data}$ =

1 + 0.15 x 10 + 0.15 x 0.1 x 1.5 x (100 + 1) = 4.7725

$AMAT_{total}$ = 1/1.4 x 1.503 + 0.4/1.4 x 4.7725 = 2.44