Chapter 5 – Part 1

Large and Fast Memory: Exploiting Memory Hierarchy

Ideal Memory

- Ideal memory for computer programmers:
  - Large (almost no limit)
  - Fast
  - Inexpensive
- But in reality we can not have all three at the same time.
  - Choose 2 out of three, maybe.
- So we must somehow create an illusion of an **affordable** and **large** memory that can be accessed as **fast** as possible.

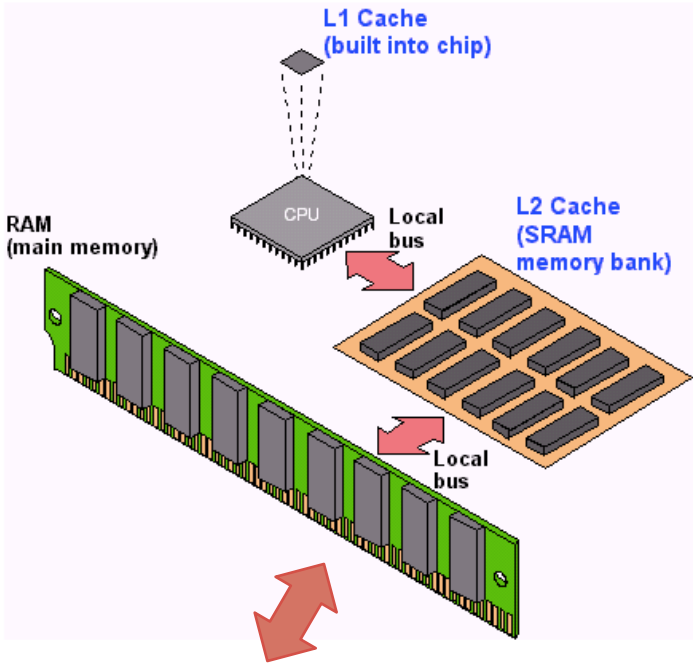
Principle of Locality

- 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**

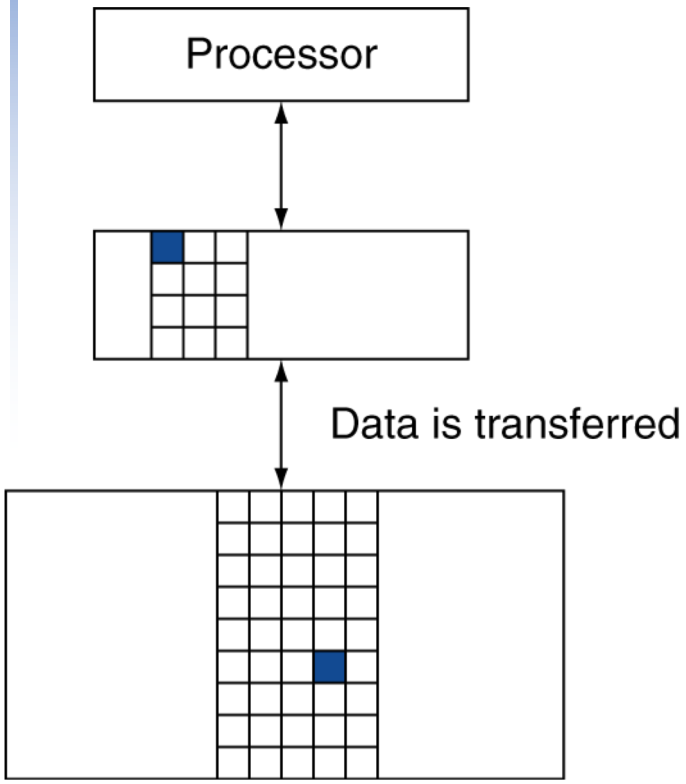
	Spatial	*Temporal*
Data	arrays	loop counters
Code	no branch/jump	loop

Taking Advantage of Locality

- Memory hierarchy
  - Store everything on disk
  - Copy recently accessed (and nearby) items from disk to small DRAM memory
    - Main memory
  - Copy more recently accessed (and nearby) items from DRAM to even smaller SRAM memory (off-chip L2 Cache)
  - Copy even more recently accessed items to L1 Cache
    - memory attached to CPU

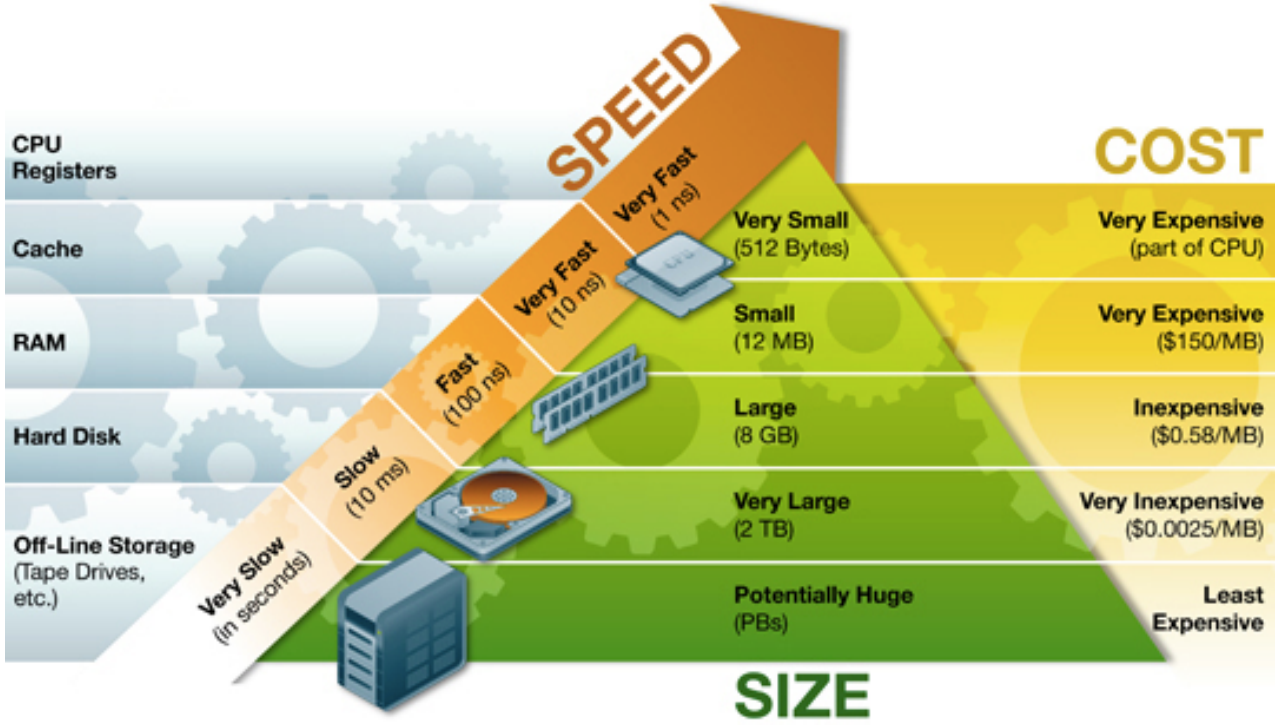


Memory Hierarchy Levels



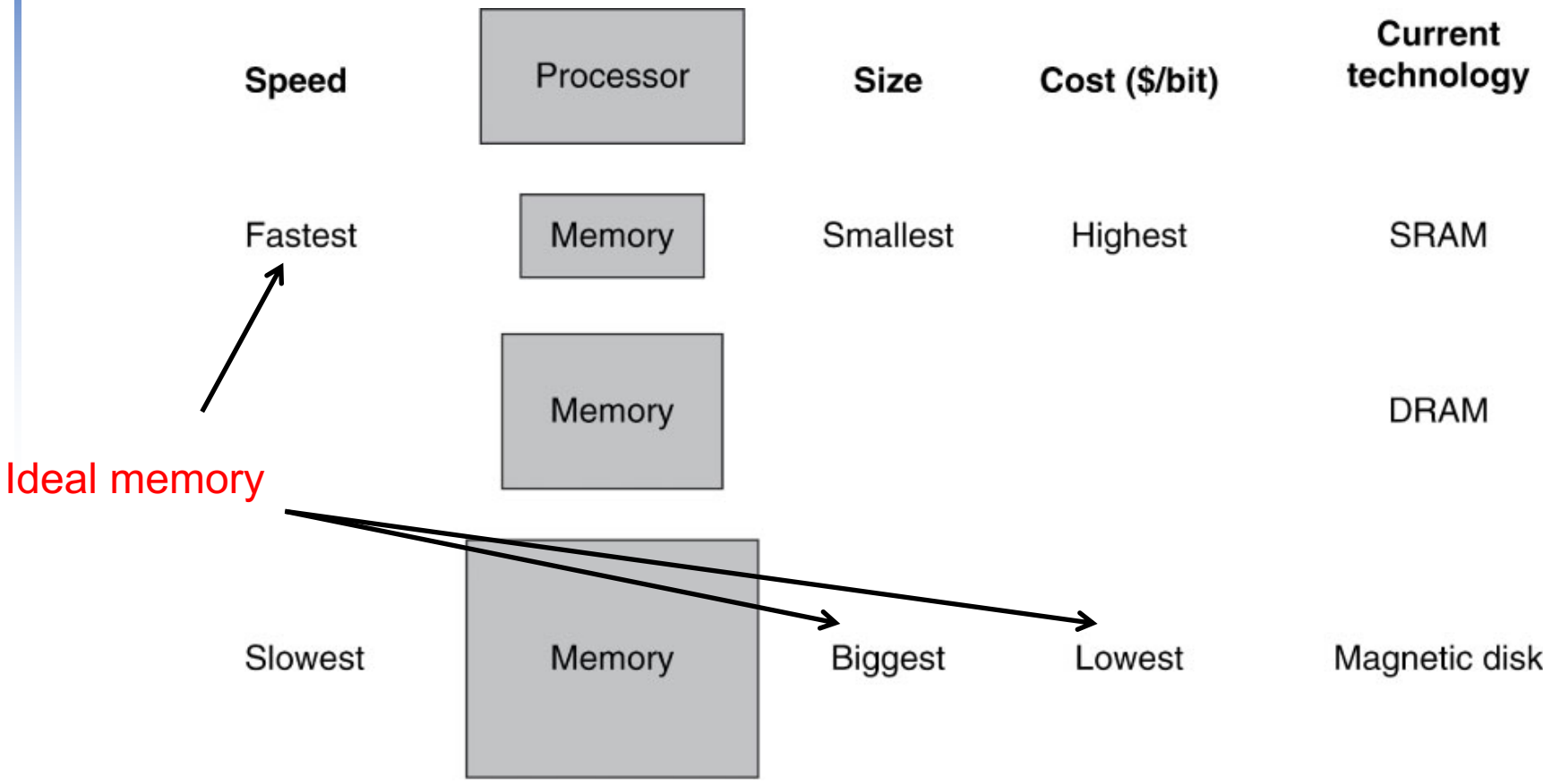
- Block (aka line): unit of copying
  - May be multiple words, or more
- If accessed data is present in upper level
  - Hit: access satisfied by upper level
    - Hit ratio: hits/accesses
- If accessed (requested) data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
    - Miss ratio: misses/accesses
      - Miss ratio = 1 – hit ratio

Memory Hierarchy Levels

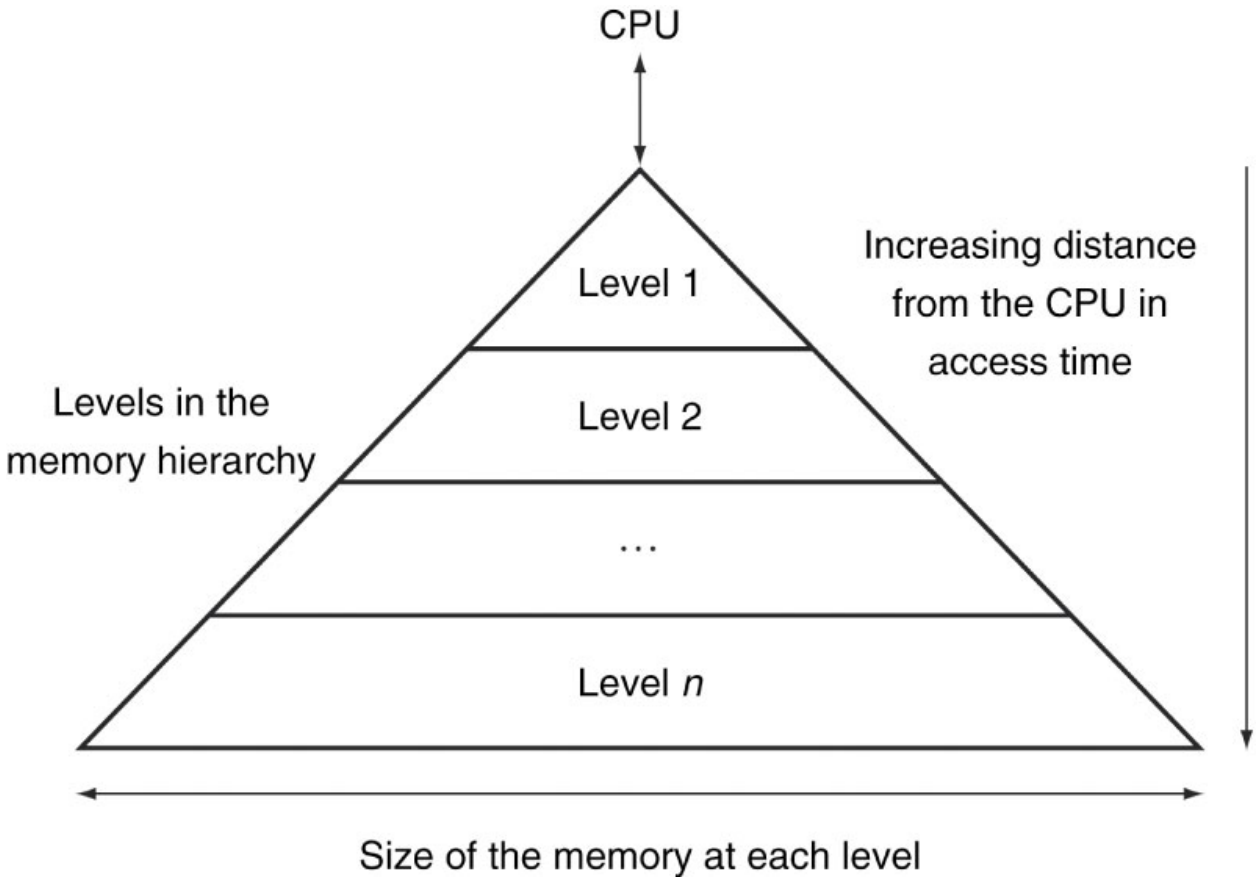


- **Ideal memory**
  - Access time of SRAM
  - Capacity and cost/GB of disk

Memory Hierarchy Levels

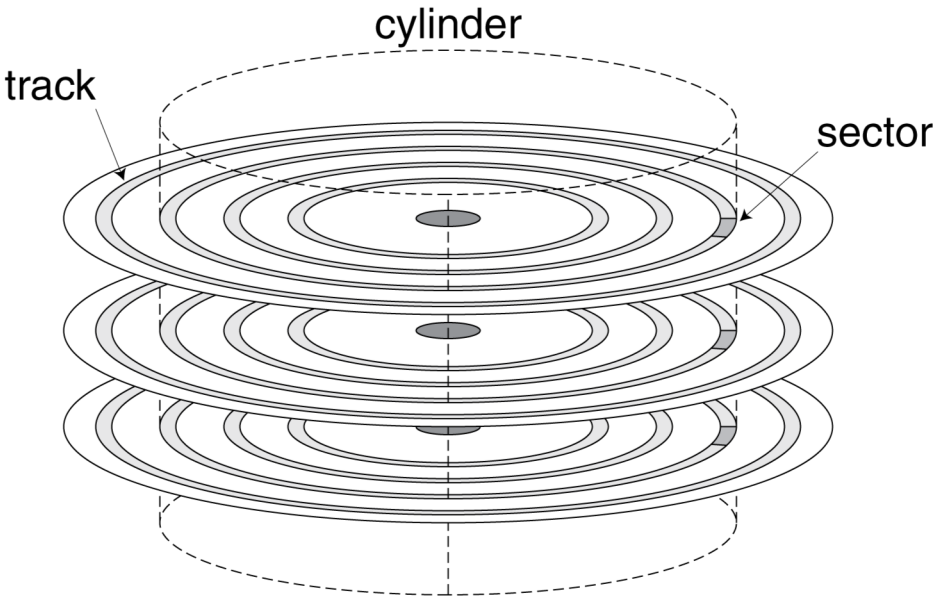


Memory Hierarchy Levels



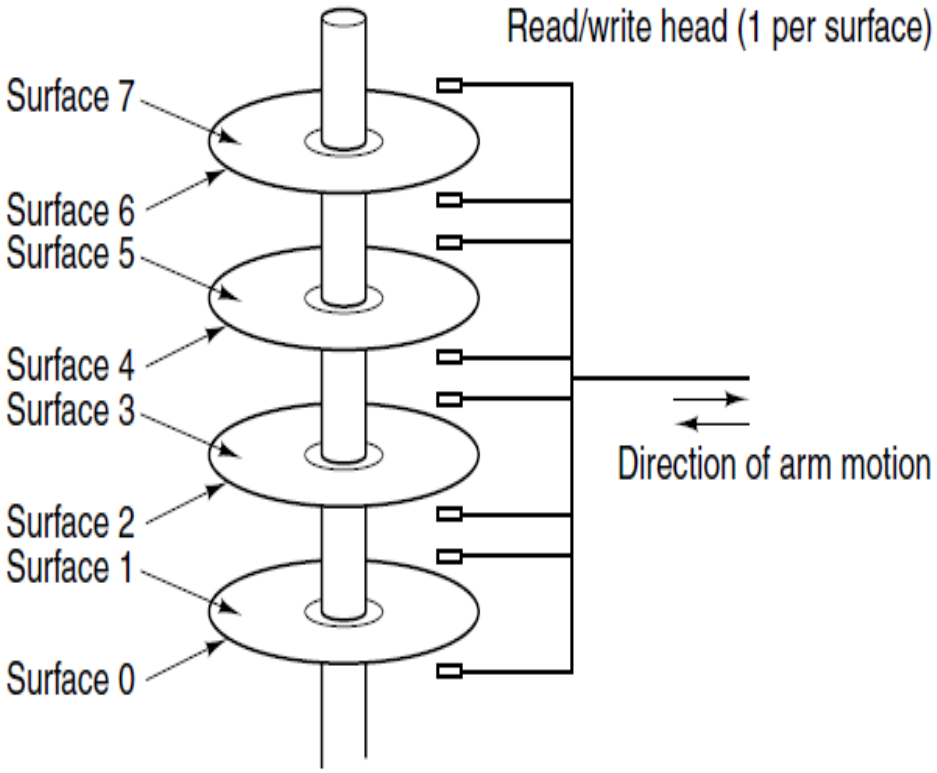
Disk Storage

- Nonvolatile, rotating magnetic storage



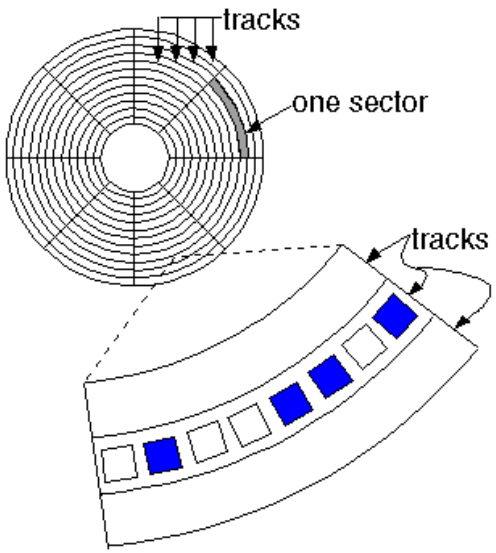
Disk Storage

- Data stored on surfaces
  - Up to two surfaces per **platter**
  - One or more platters per **disk**
- Data in concentric tracks
  - Tracks broken into **sectors** - 256B–4KB per sector
  - **Cylinder**: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move all together



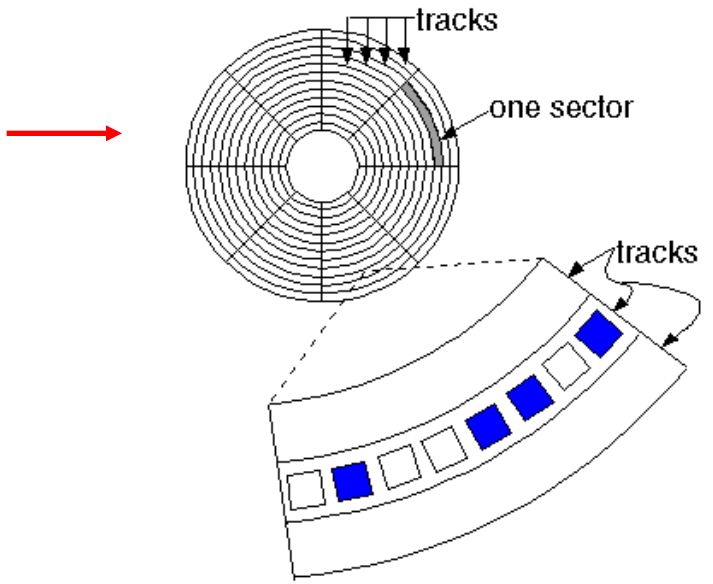
Disk Sectors

- Each sector contains:
  1. Sector ID (number and location)
  2. Data
  3. Error correcting code (ECC)
  4. Synchronization fields and gaps



Disk Sector Access

- Access to a sector involves
  - Queuing delay if other accesses are pending
  - Seek: move the heads to proper track
  - Rotational latency (wait for the desired sector to rotate under the read/write head)
  - Data transfer (time to transfer a block of bits)
  - Controller overhead



Disk Sector Access Example

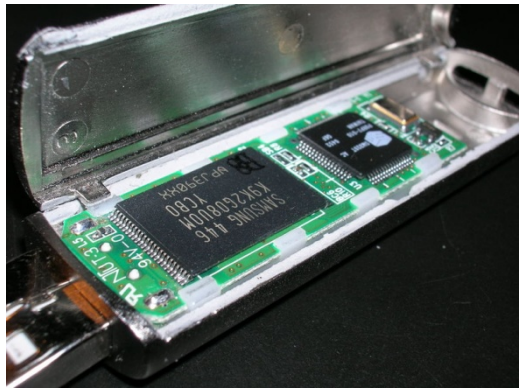
- Given
  - 512B sector, 15,000 rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time for one sector
  - 4ms (seek time)
  - + [½ * 1/(15,000/60)] = 2ms (avg. rotational latency)
  - + [512B / 100MB/s] = 0.005ms (transfer time)
  - + 0.2ms (controller delay)
  - **= 6.2ms** (average read time)
- If we can reduce average seek time to 1ms
  - Average read time = 3.2ms

Disk Performance Issues

- Manufacturers quote average seek time
- Smart disk controller allocate physical sectors on disk
- Disk drives include caches

Flash Storage

- Non-volatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between hard disk and main memory DRAM)



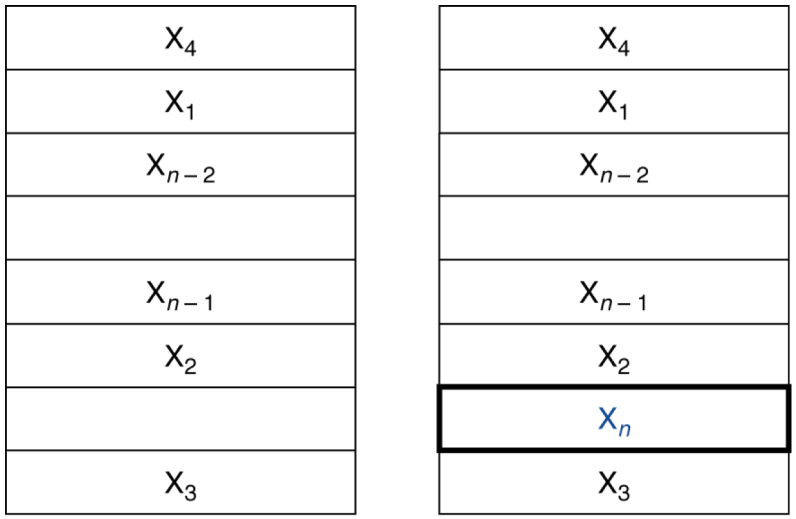
Flash Types

- NOR flash: bit cell like a NOR gate
- NAND flash: bit cell like a NAND gate
- Flash bits wears out after 1000's of accesses

https://www.youtube.com/watch?v=ELl3abwYQ90

Cache Memory

- Cache memory
  - The level of the memory hierarchy closest to the CPU
- Given block accesses $X_1, \ldots, X_{n-1}, X_n$



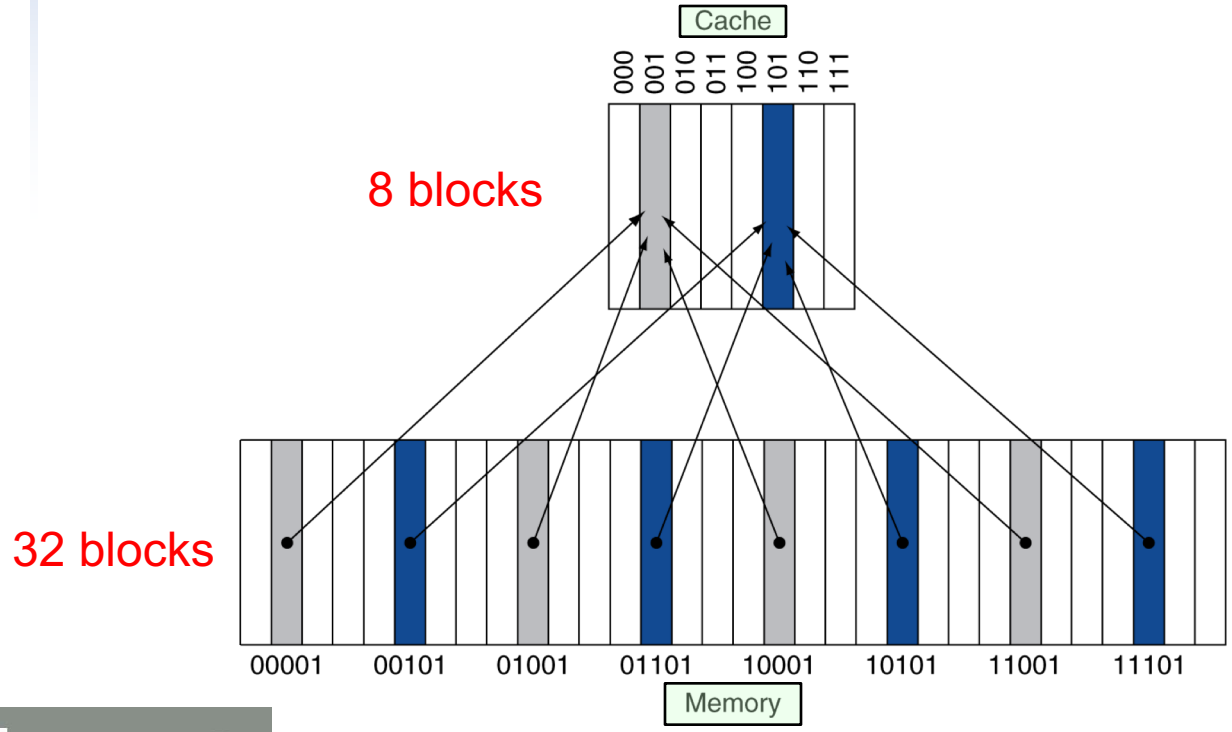
a. Before the reference to X_n

b. After the reference to X_n

- How do we know if the data is present in cache?
- Where do we look?
- How do we search the cache for that item?

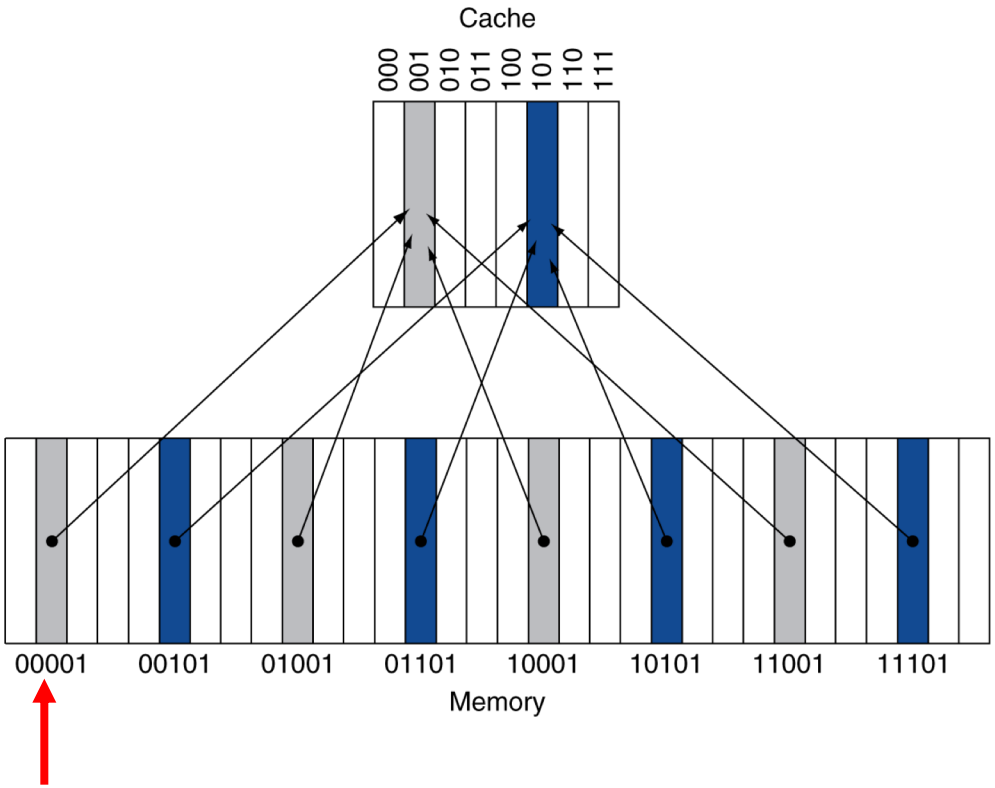
Direct Mapped Cache

- A 32-block memory and a 8-block cache system
- Cache location determined by address in mem
- Direct mapped: only one choice
  - (Block address in mem) mod (# of blocks in cache) will lead you to the block address in cache



- # of blocks in cache is a power of 2
- Use 3 low-order address bits as cache index

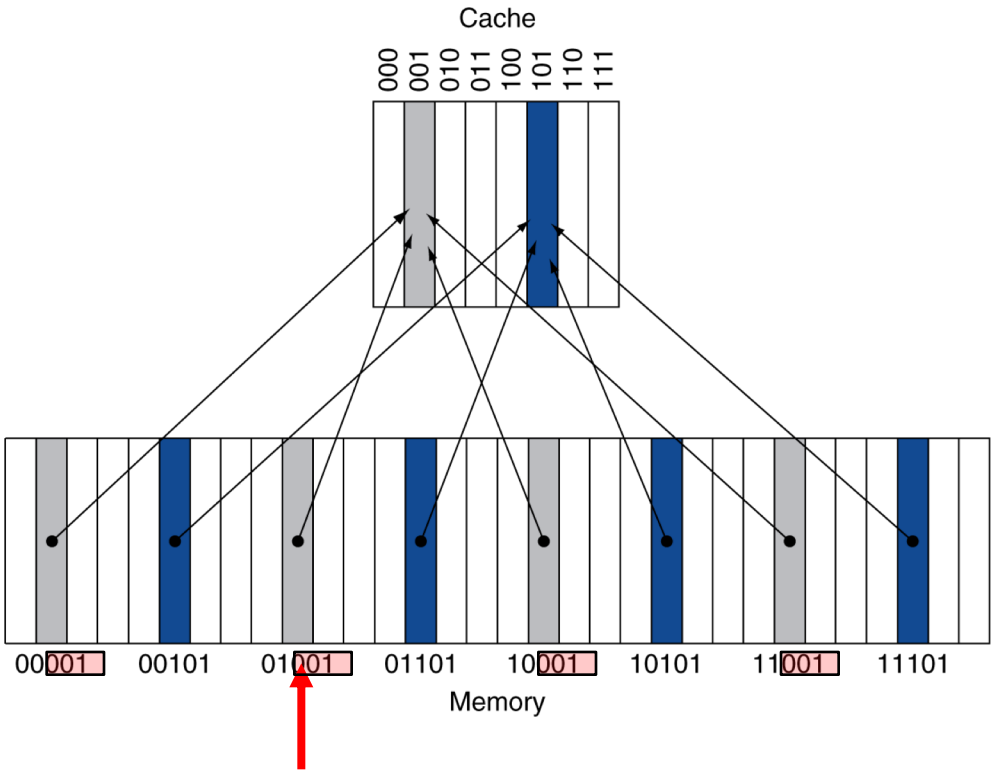
Direct Mapped Cache



Mem	Cache
1 (Mod 8)	1
5 (Mod 8)	5

Direct mapping of 32 memory words into 8 cache words

Direct Mapped Cache

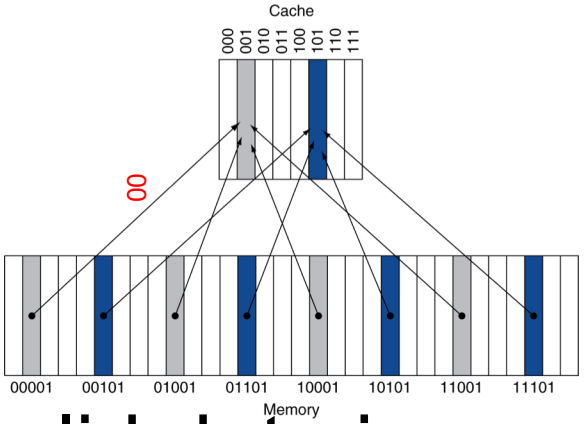


Direct mapping of 32 memory words into 8 cache words

Mem		Cache
1	(Mod 8)	1
5	(Mod 8)	5
9	(mod 8)	1
13	(mod 8)	5
17	(mod 8)	1
21	(mod 8)	5
25	(mod 8)	1
29	(mod 8)	5

Tags and Valid Bits

- How do we know which particular memory content is stored in a cache block?
  - Store also the high-order memory address bits with the data
    - Called the tag



- How can we tell if there is valid data in a cache block?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Cache Example

- 8-block cache, 1 word/block, direct mapped
- Initial state

8-block Cache System

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Mem word address	Mem binary address	Hit/miss	Cache block
22	10 110	Miss	110

8-block Cache System

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Mem word address	Mem Binary address	Hit/miss	Cache block
26	11 010	Miss	010

8-block Cache System

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110		110

8-block Cache System

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	**Y**	**10**	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000		000
3	00 011		011

8-block Cache System

Index	V	Tag	Data
000	**N**		
001	N		
010	Y	11	Mem[11010]
011	**N**		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000		000

8-block Cache System

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

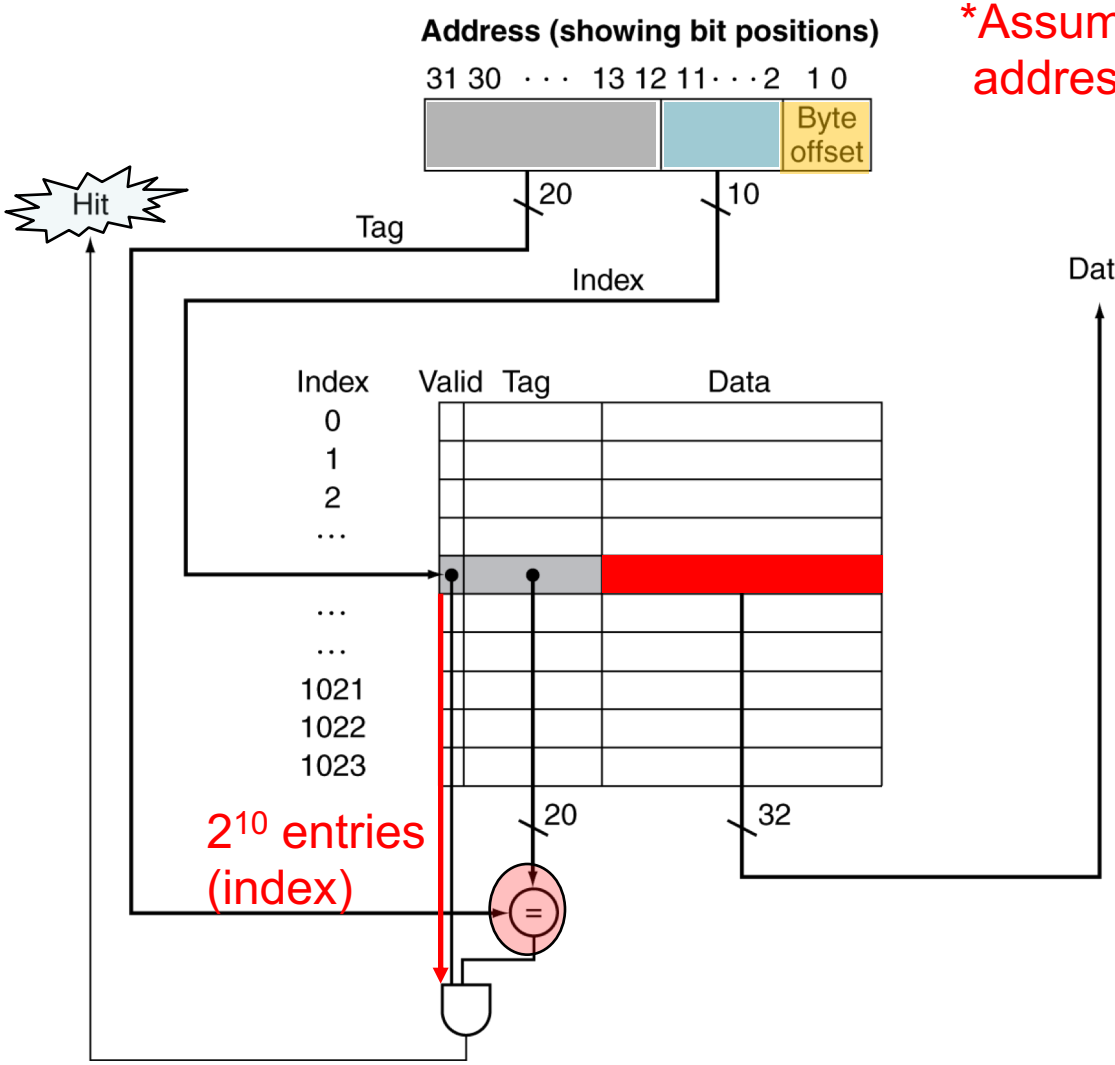
Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

8-block Cache System

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Address Subdivision

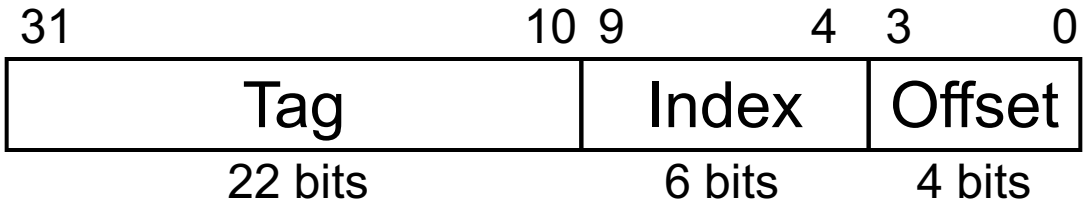


*Assuming a 32-bit addressing

* Assuming 1-word data blocks

Larger Block Size

- Cache: 64 blocks (or 64 cache entries), **16 bytes per block** (assume 32-bit addressing)
  - How many bits required for byte offset within a block?
    - $2^{4(\text{bits})}$ = 16 words
  - How many bits required to uniquely identify the block index?
    - 64 (cache entries) = $2^{6(\text{bits})}$
  - And # of bits left to be used for the tag?
    - 22 bits



Block Size Considerations

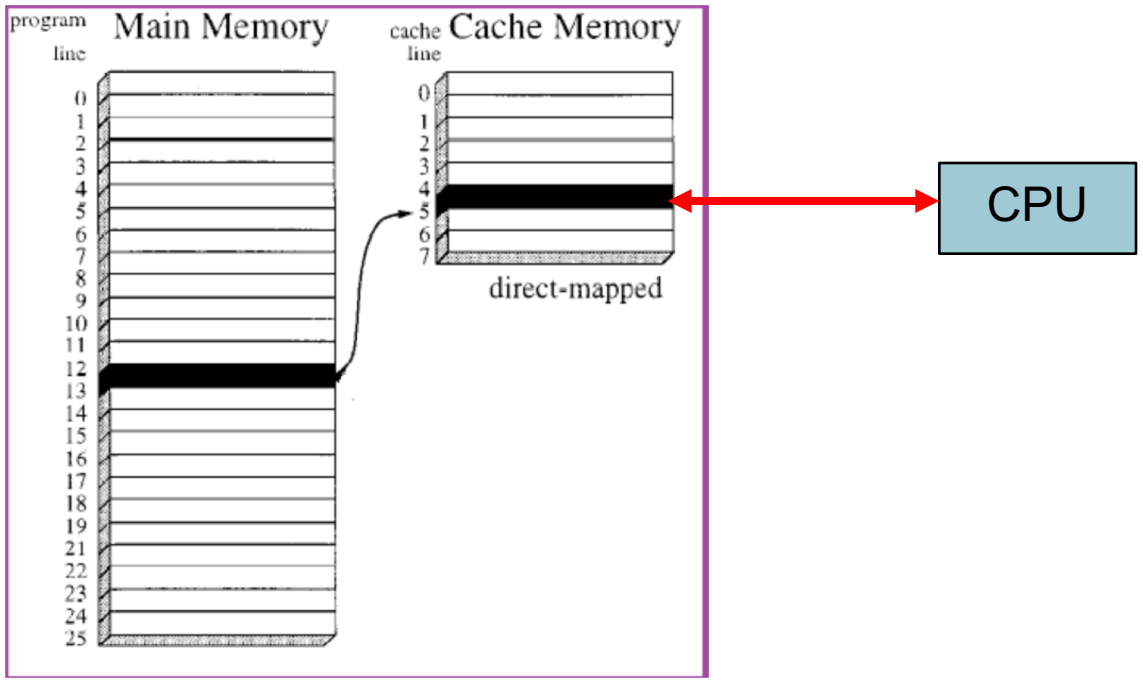
- Larger blocks should reduce miss rate. Why?
  - Due to spatial locality
- But programs with lots of branches
  - Larger blocks ⇒ lead to fewer blocks in cache
    - More competition ⇒ increased miss penalty
  - Larger blocks ⇒ pollution
- So increased miss penalty
  - Can override benefit of reduced miss rate
  - However, early restart and critical-word-first techniques can help with the increased miss penalty

Block Size Considerations

- How restart and critical-word-first can help with the increased miss penalty?
  - In both techniques: do not wait for full large block to be loaded before restarting CPU
  - **Early restart** - As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - **Critical Word First** - Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block.

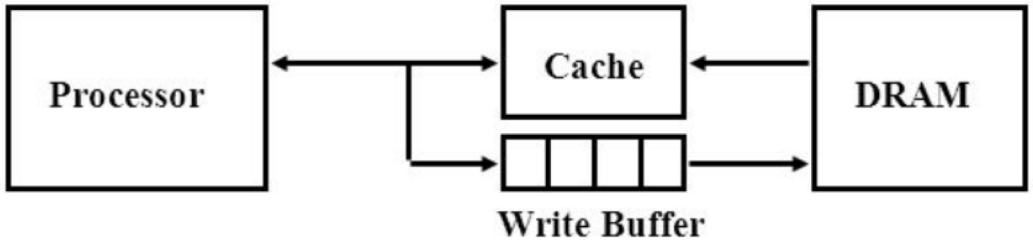
Write-Through

- On data-write hit (data to be updated exists in cache):
  - Could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: then also update memory



Write-Through

- Memory writes take much longer
  - e.g., if base CPI = 1, and 10% of instructions are stores, and write to memory taking 100 cycles
    - Effective CPI = 1 + 0.1 × 100 = 11 (a significant increase in CPI as a result of just10% memory writes)
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full



Write-Back

- Alternative: On data-write hit
  - Just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced (evicted)
  - Write it back to memory
  - Can also use a write buffer to allow replacing block to be read first

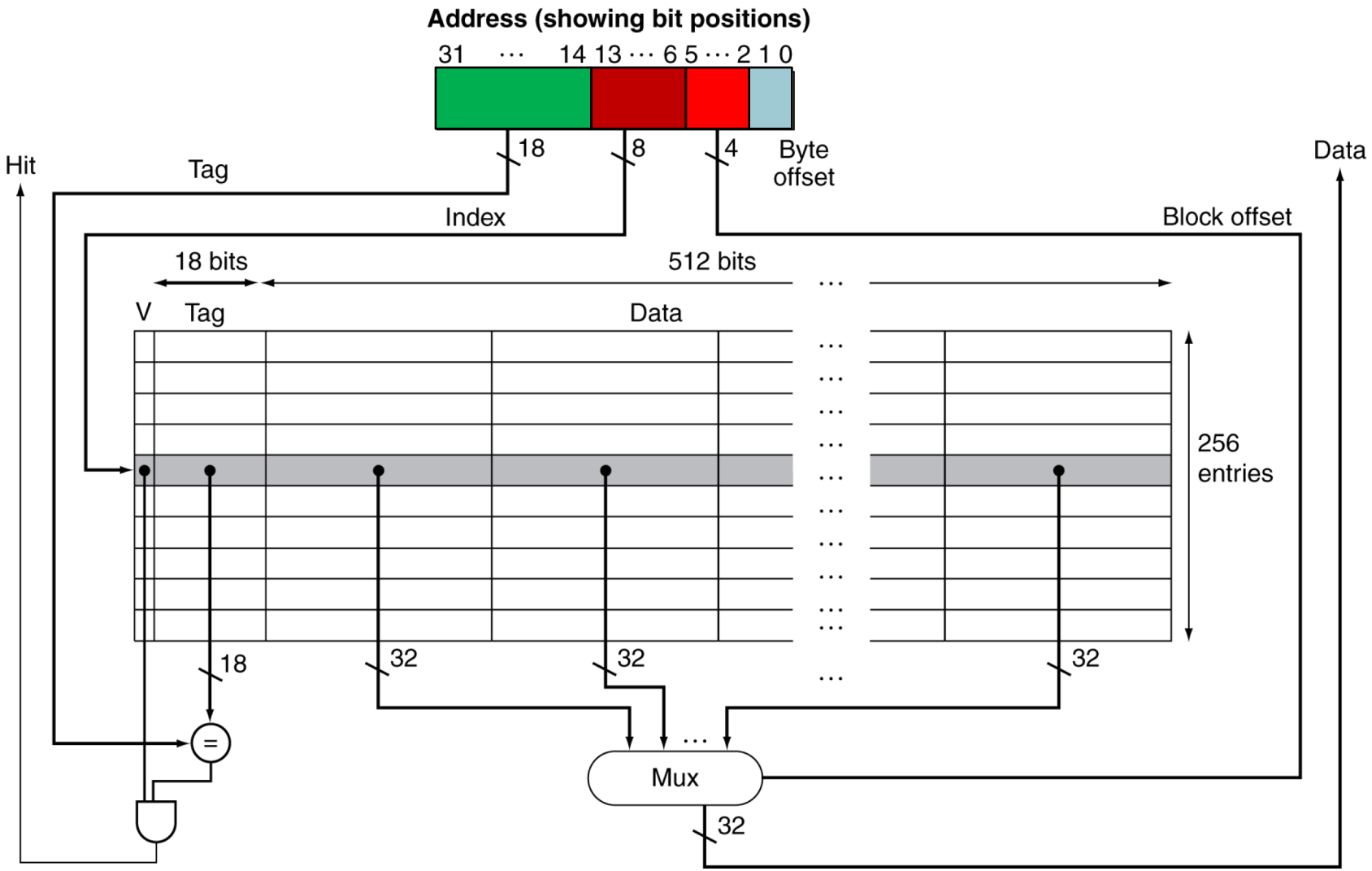
SPEC

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

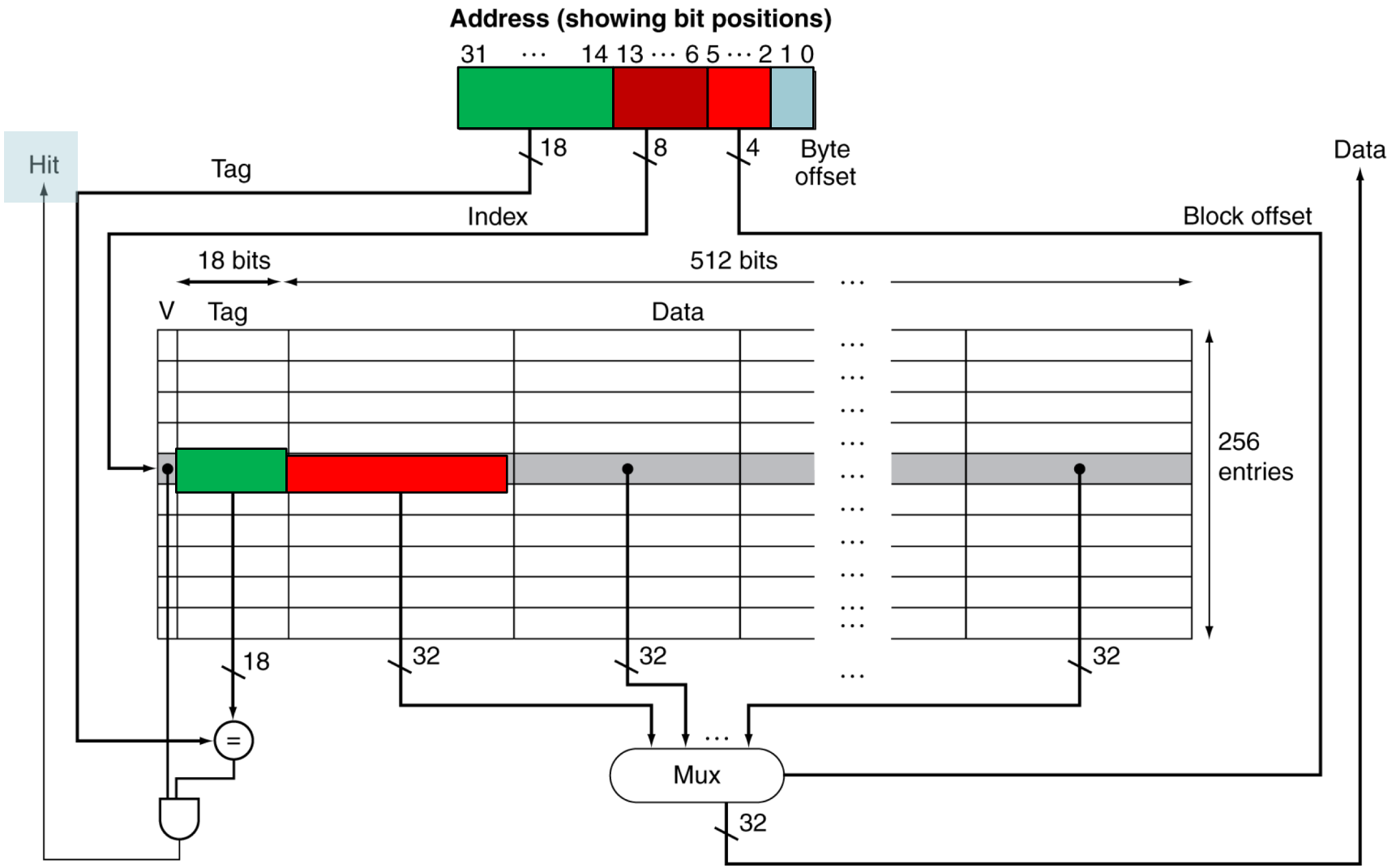
Example: Intrinsity FastMATH

- Embedded MIPS (RISC-based) processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 256 blocks × 16 words per block (64 bytes per block) = 16KB (total size of each cache):
  - D-cache: write-through or write-back
- SPEC2000 miss rates for:
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Example: Intrinsity FastMATH



Measuring Cache Performance

- Components of CPU execution time
  - Program execution cycles
    - Includes cache hit time (about 1 cycle)
  - Memory stall cycles
    - Mainly from cache misses
    - This component would be zero for a perfect/ideal cache

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

Cache Performance Example

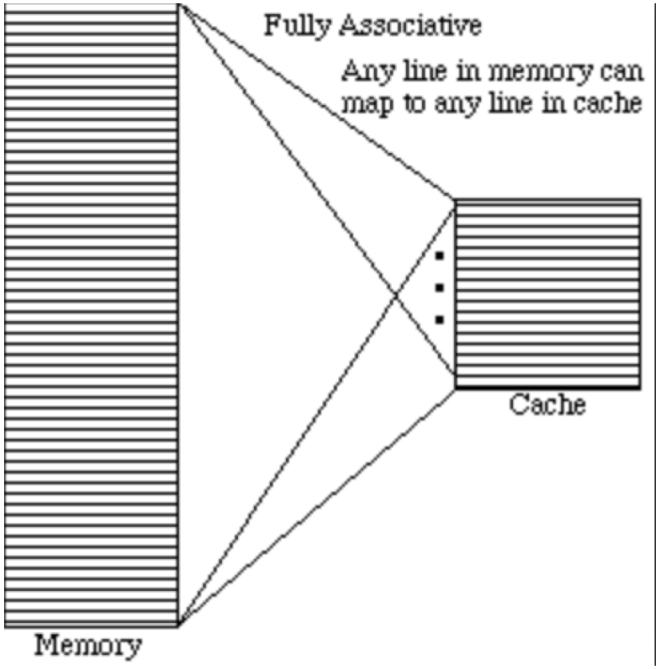
- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal/perfect cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 1(100% of instructions use I-cache) x 0.02 (2% of them miss) × 100 (miss penalty) = 2 cycles
  - D-cache: 0.36(36% of instructions use D-cache) × 0.04 (4% of them miss) × 100 = 1.44 cycles
- Actual CPI = 2 + 2 + 1.44 = 5.44 cycles
  - CPU with ideal cache is 5.44/2 = 2.72 times faster

Average Memory Access Time

- Hit time is also important for performance
- Average memory access time (AMAT)
  - AMAT = Hit time
- Example
  - CPU with 1ns clock,
    - Hit time = 1 cycle,
    - I-cache miss rate = 5%,
    - Miss penalty = 20 cycles,
  - AMAT = 1 + 0.05 × 20 = 2 cycles
    - 2 cycles per instruction fetch (twice the cycles due to miss rate and penalty)

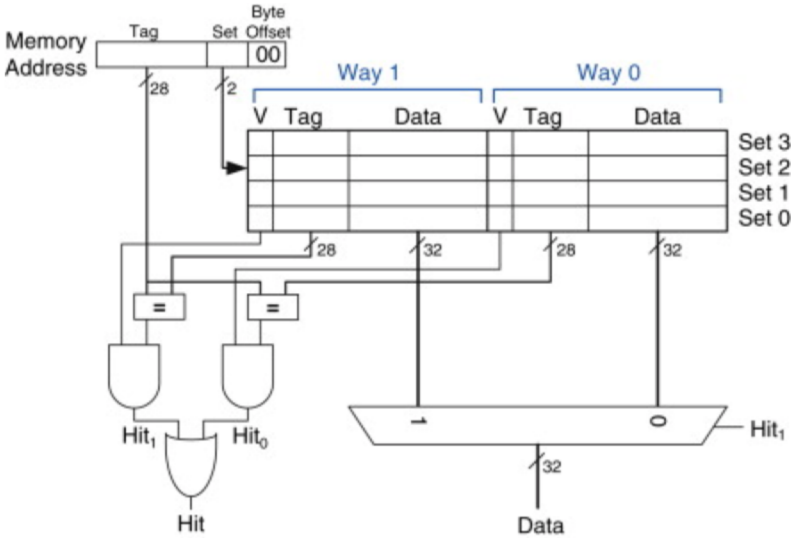
Associative Caches

- Fully associative
  - Allow a given block to go in any cache entry
  - Requires all entries to be searched at once



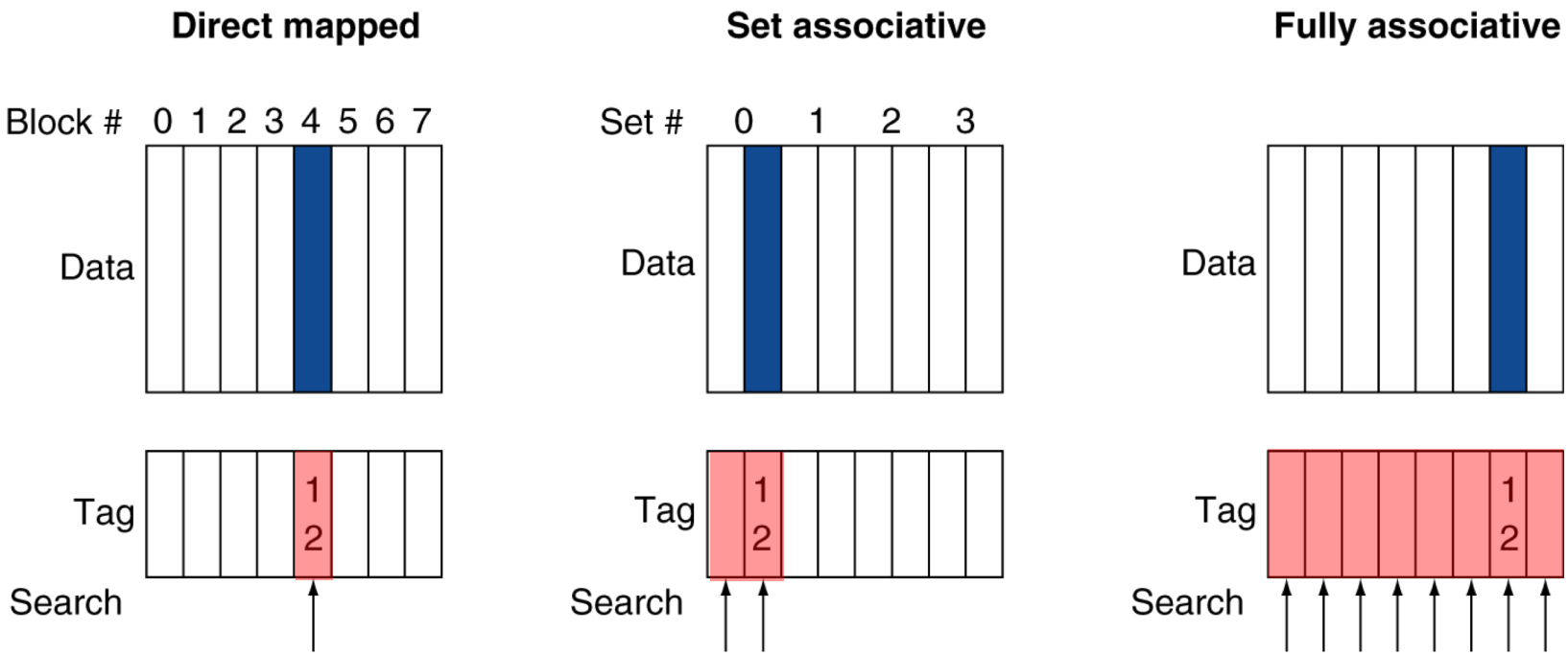
Associative Caches

- *n*-way set associative
  - Each entry (set) contains *n* possibilities to store data
  - Block number in memory determines which set
    - (Block number) mod (# of Sets in cache)
  - Just search all the possibilities within a set (n)



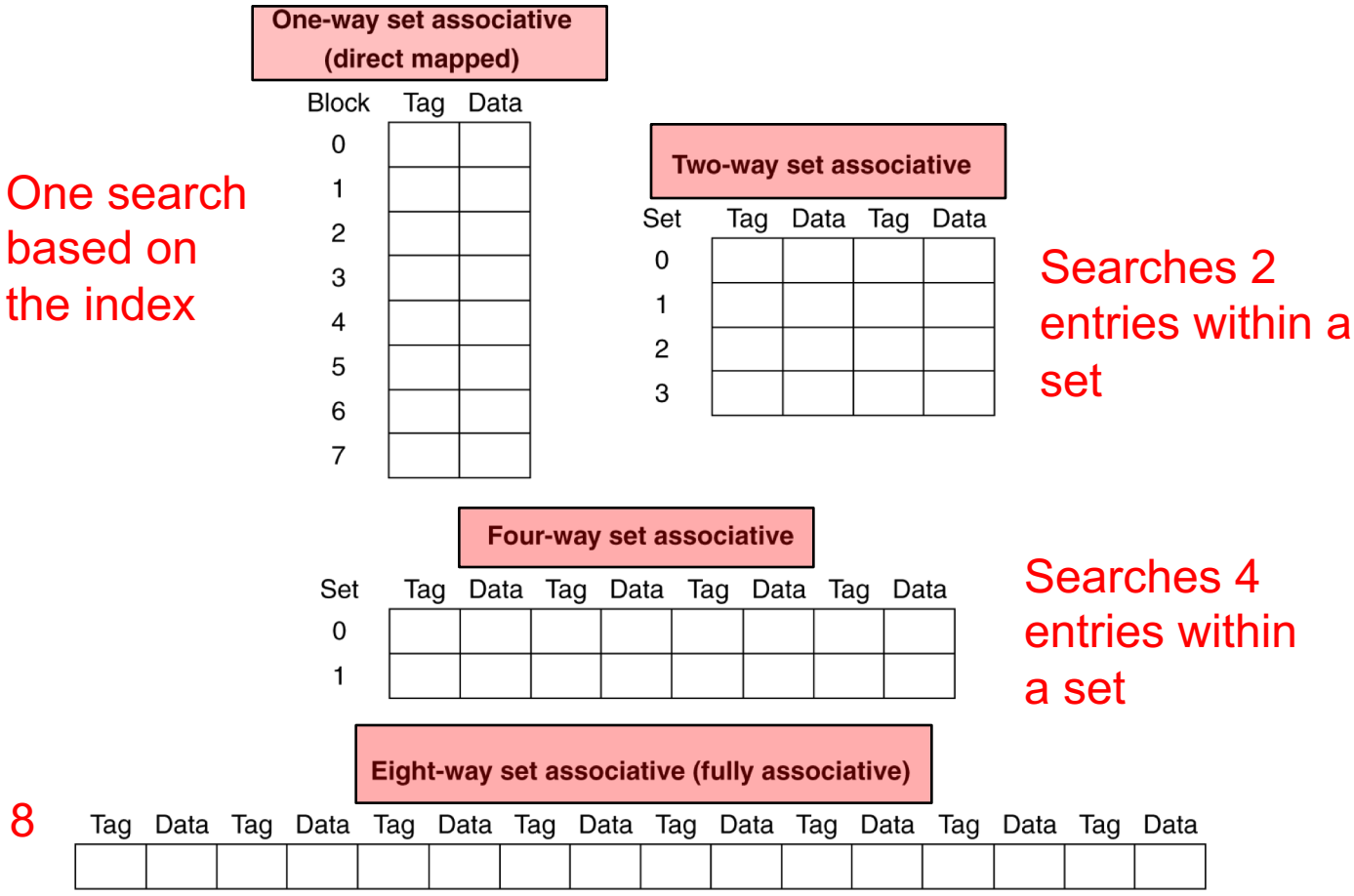
2-way associative cache

Associative Cache Example



Spectrum of Associativity

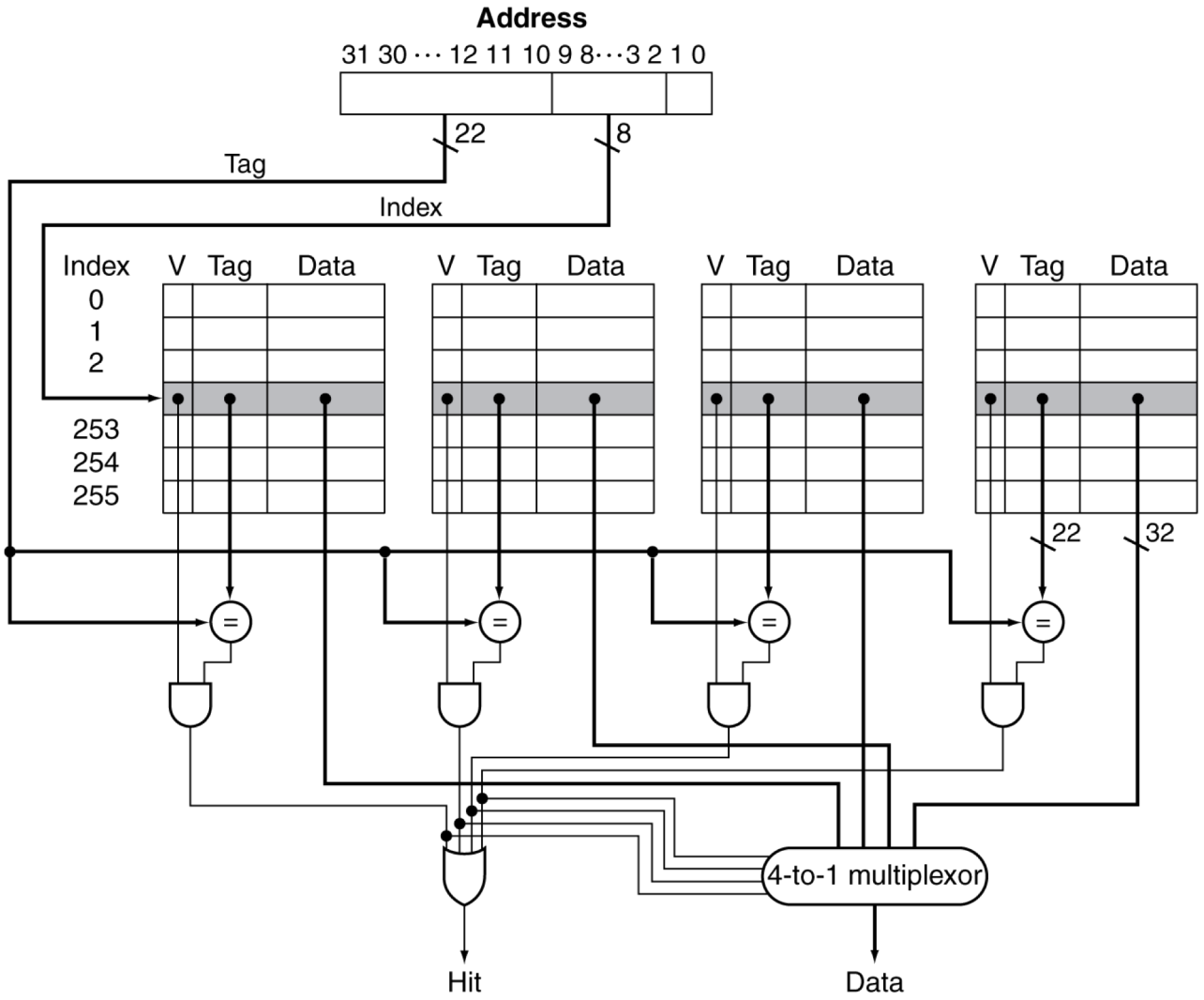
- For a cache size of 8 blocks



How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3% (direct map)
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Set Associative Cache Organization

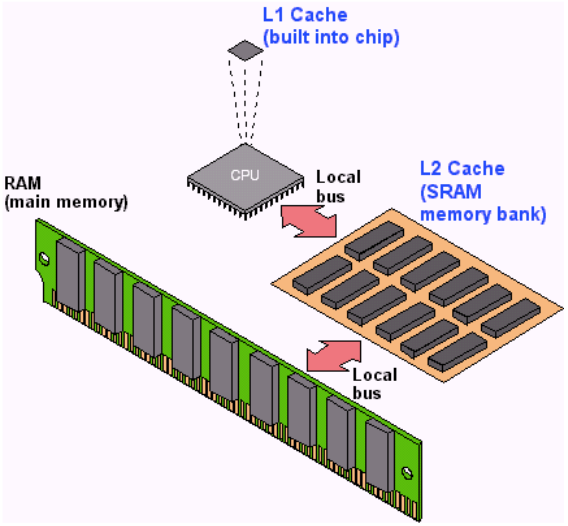


Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry
  - Otherwise, choose among entries in the set
    - Least-recently used (LRU)
    - Random

Multilevel Caches

- Primary cache: attached to CPU
  - Small, but very fast
- Level-2 cache: services misses from primary cache
  - Larger, slower, but still faster than main memory



- Main memory: services L-2 cache misses
- Some high-end systems include L-3 cache

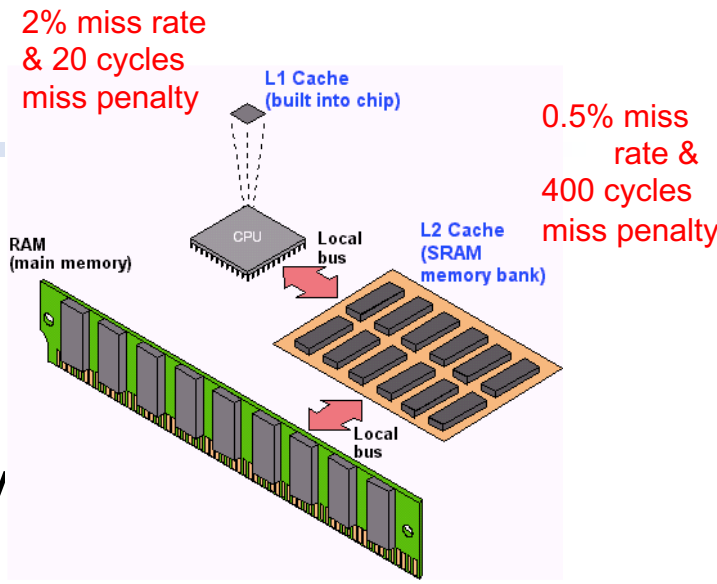
Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
    - Clock period = 1/4000000000 = 0.25 ns
  - Miss rate per 100 instructions = 2%
  - Main memory access time = 100ns
    - Or 100ns/0,25ns = 400 cycles (miss penalty)
- With just primary cache
  - Every miss in L1 will cost us 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Example (cont.)

- Now add L-1 cache
  - Access time to L2 = 20 cycles
    (a miss in L1 but a hit in L2 costs 20 cycles)
  - Global miss rate to main memory from L2 = 0.5%

(a miss in L2 and going to main memory, costs 400 cycles)



Adding L-1, 2% of misses goes to L2

Now only 0.5% of misses in L2 has to go to main memory

- CPI = 1 + (0.02 × 20 cycles)
  + (0.005 × 400 cycles) = 3.4

- Performance ratio = 9/3.4 = 2.6