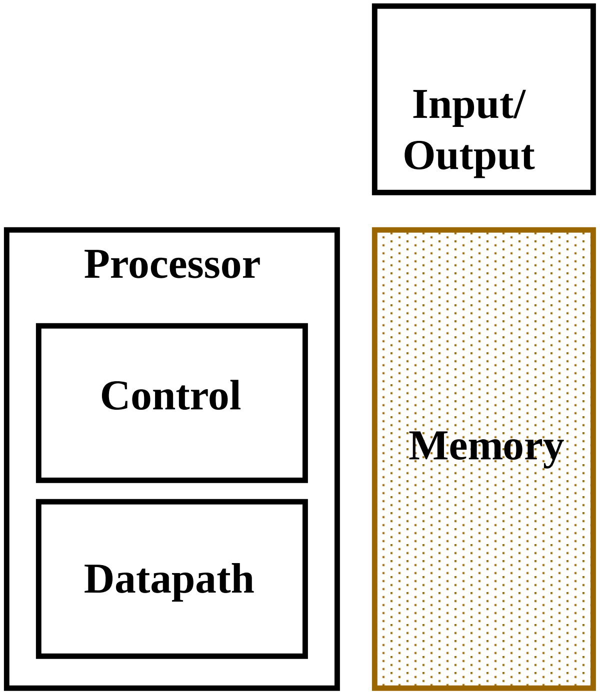
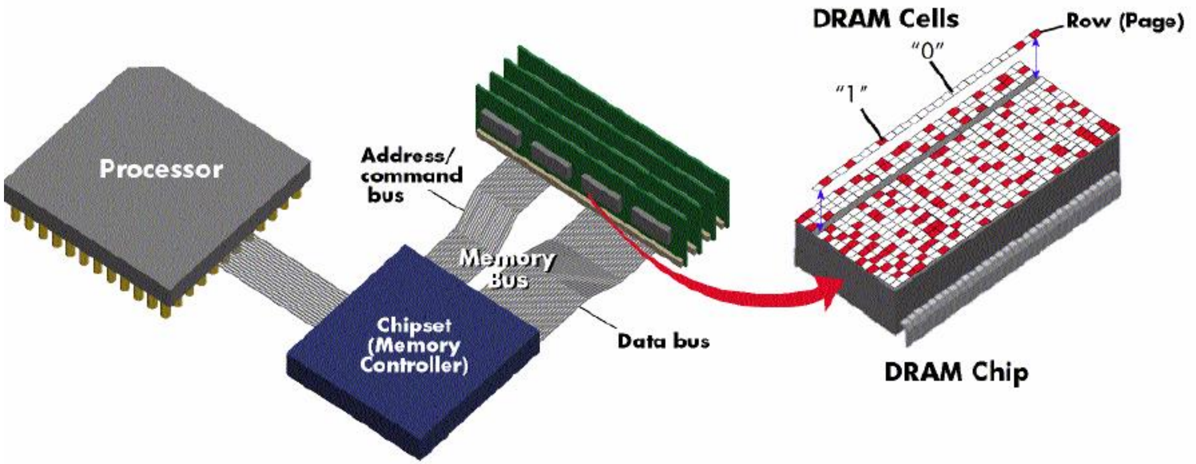
CSE331
Computer Organization
Memory Hierarchy and Cache Design

Lecture 12

The Big Picture: Where are We Now?

- The Five Classic Components of a Computer
- Memory is usually implemented as:
  - Dynamic Random Access Memory (DRAM) - for main memory
  - Static Random Access Memory (SRAM) - for cache





Technology Trends

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

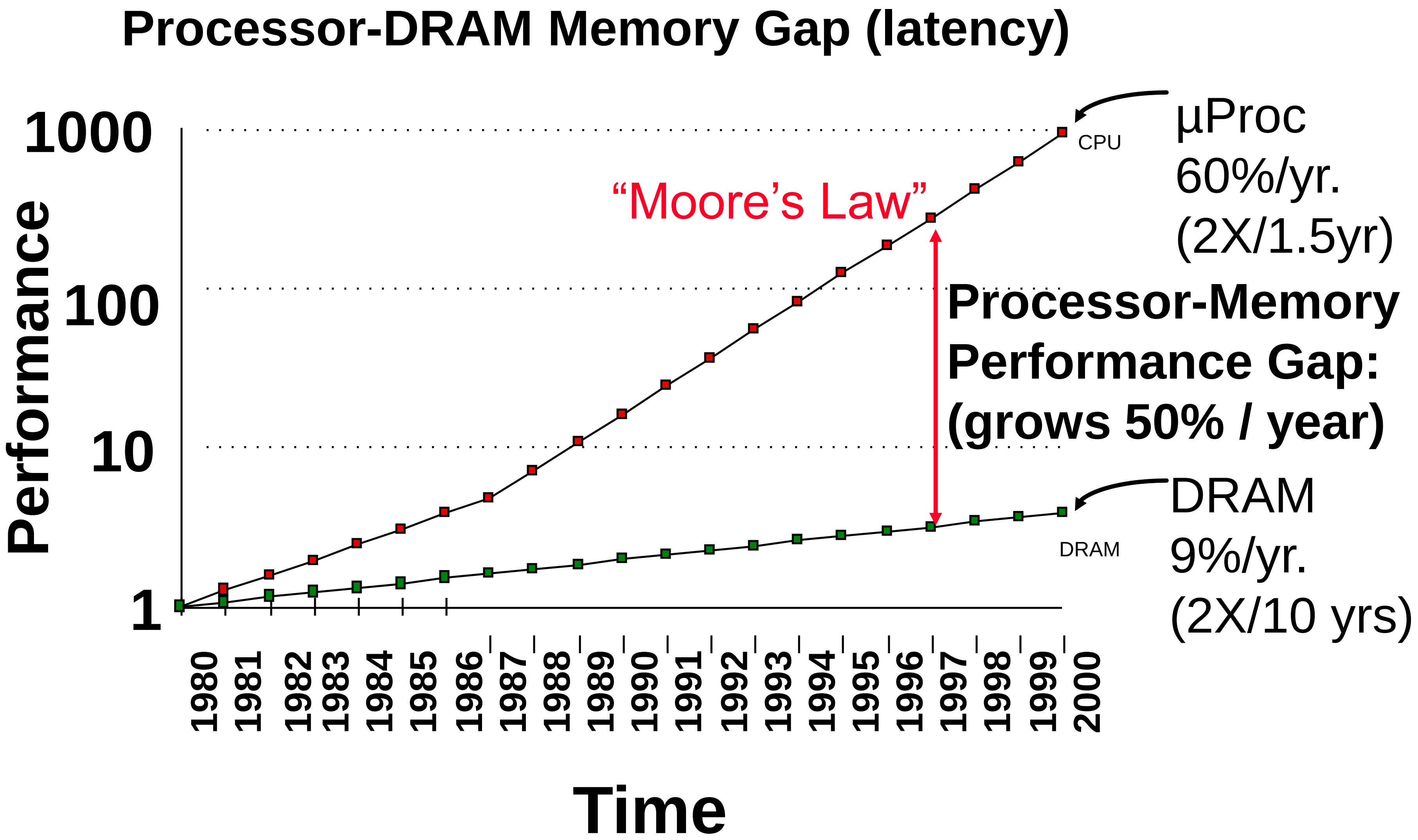
DRAM

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	100 ns
2001	1 Gb	80 ns

1000:1!

2:1!

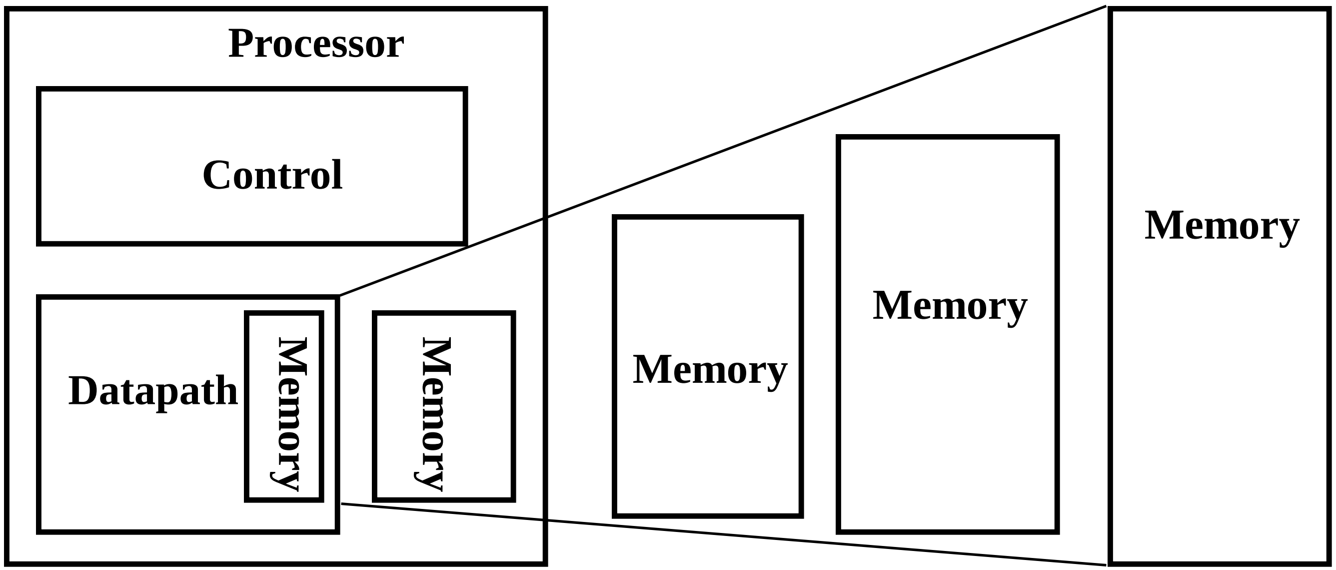
Who Cares About Memory?



Today’s Situation: Microprocessors

- Rely on caches to bridge gap
- Cache is a high-speed memory between the processor and main memory
- 1980: no cache in µproc;
  1997 2-level cache, on Alpha 21164 µproc

An Expanded View of the Memory System



Speed:	Fastest	Slowest
Size:	Smallest	Biggest
Cost:	Highest	Lowest

Taking Advantage of Locality

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

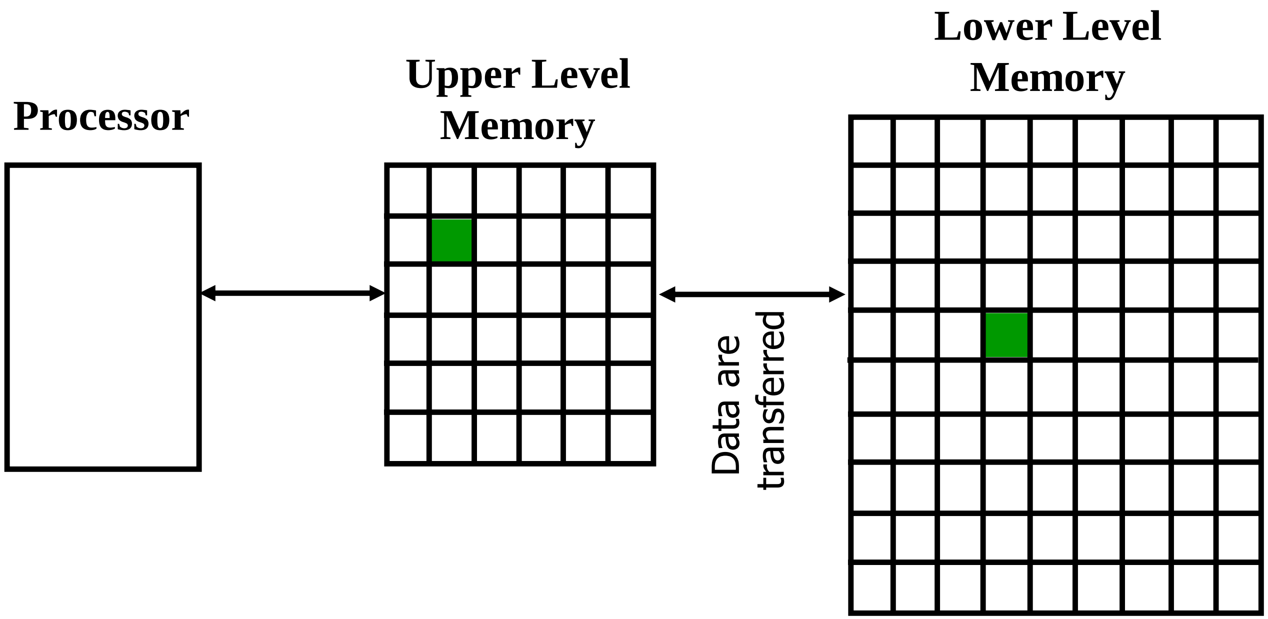
Memory Hierarchy: How Does it Work?

- Temporal Locality (Locality in Time):

  => Keep most recently accessed data items closer to the processor

- Spatial Locality (Locality in Space):

  => Move blocks consists of contiguous words to the upper levels

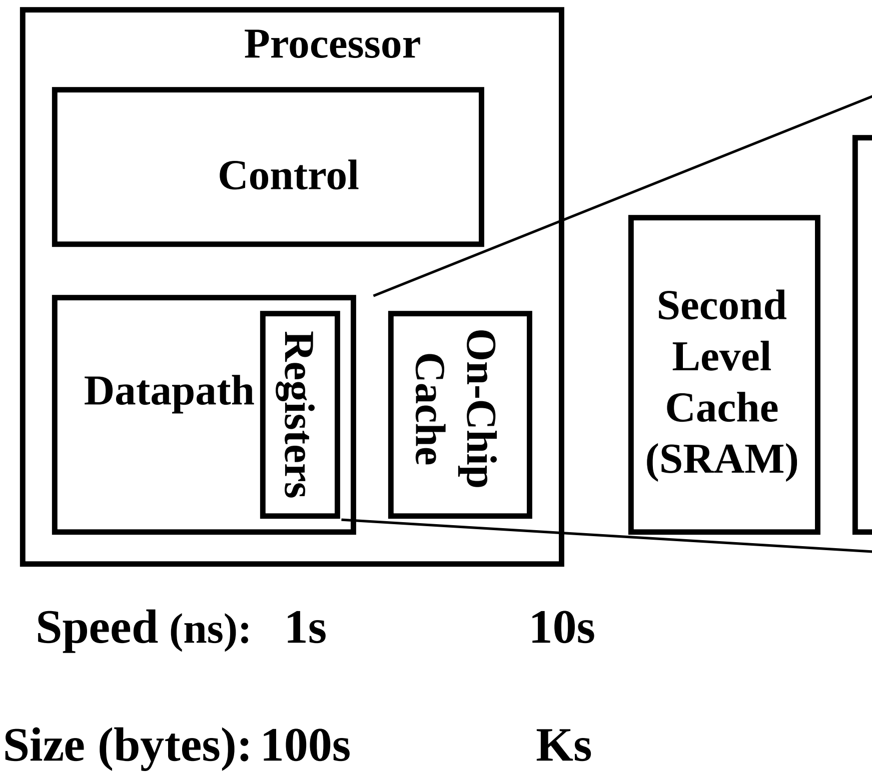


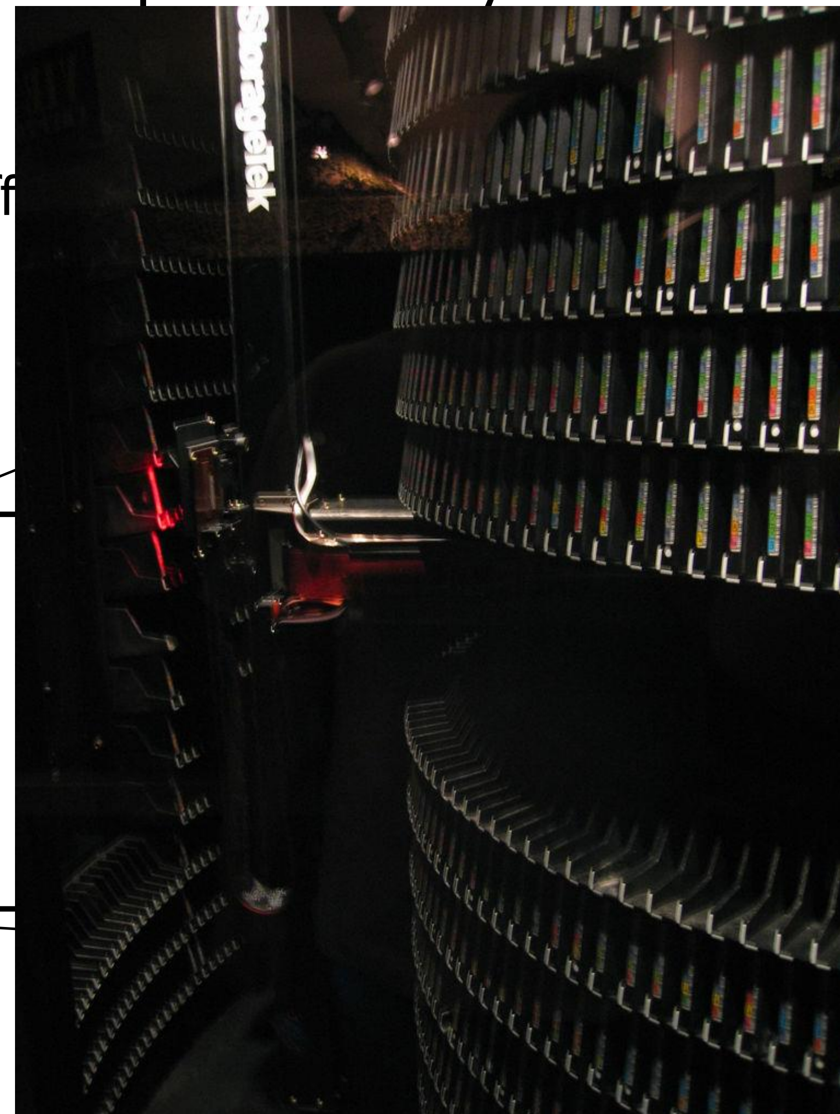
Memory Hierarchy: Terminology

- Hit: If the data requested by a processor appears in some block in the upper level.
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
  - Hit Rate: The fraction of memory access found in the upper level
- Miss: If the data is not found in the upper level.
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty

Memory Hierarchy of a Modern Computer System

- By taking advantage of the principle of locality:
  - Present the user with as much
    cheapest technology.
  - Provide access at the speed off





How is the hierarchy managed?

- Registers <-> Memory
  - by compiler (programmer?)
- cache <-> memory
  - by the hardware
- memory <-> disks
  - by the hardware and operating system (virtual memory)
  - by the programmer (files)

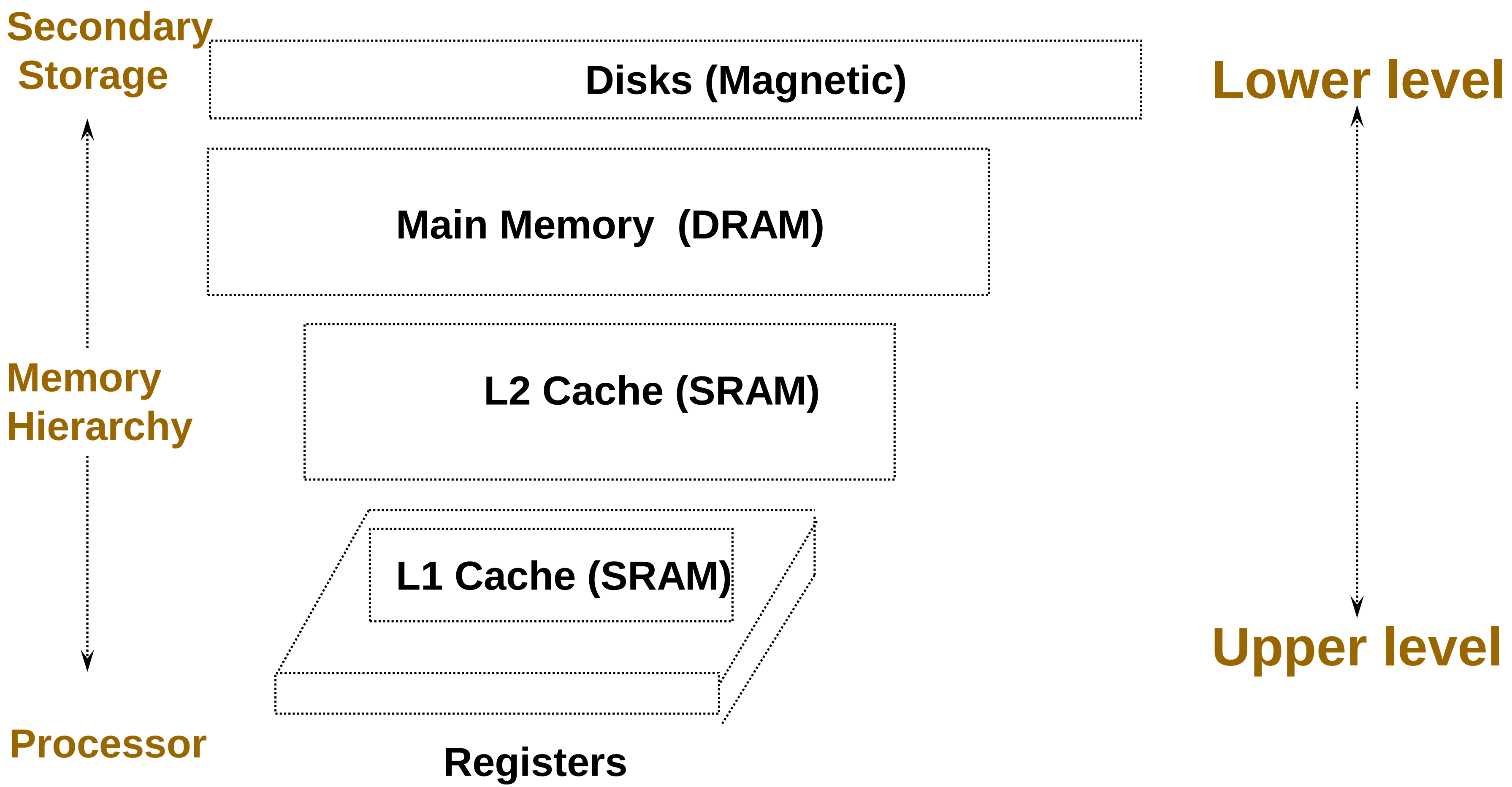
Memory Hierarchy Technology

- Random Access:
  - "Random" is good: access time is the same for all locations
  - DRAM: Dynamic Random Access Memory
    - High density, low power, cheap, slow
    - Dynamic: need to be "refreshed" regularly
  - SRAM: Static Random Access Memory
    - Low density, high power, expensive, fast
    - Static: content will last "forever" (until lose power)
- "Non-so-random" Access Technology:
  - Access time varies from location to location and from time to time
  - Examples: Disk, CDROM
- Sequential Access Technology: access time linear in location (e.g.,Tape)

General Principles of Memory

- Locality
  - *Temporal Locality* : referenced memory is likely to be referenced again soon (e.g. code within a loop)
  - *Spatial Locality* : memory close to referenced memory is likely to be referenced soon (e.g., data in a sequentially access array)
- Definitions
  - *Upper* : memory closer to processor
  - *Block* : minimum unit that is present or not present
  - *Block address* : location of block in memory
  - *Hit* : Data is found in the desired location
  - *Hit time* : time to access upper level
  - *Miss rate* : percentage of time item not found in upper level
- Locality + smaller HW is faster = memory hierarchy
  - *Levels* : each smaller, faster, more expensive/byte than level below
  - *Inclusive* : data found in upper level also found in the lower level

Memory Hierarchy



Differences in Memory Levels (2005)

Level	Memory Technology	Typical Size	Typical Access Time	Cost per Mbyte
Registers	D Flip-Flops	64 32-bit	2 -3 ns	N/A
L1 Cache (on chip)	SRAM	16 Kbytes	5 - 25 ns	\$100 - \$250
L2Cache (off chip)	SRAM	256 Kbytes	5 - 25 ns	\$100 - \$250
Main Memory	DRAM	256 Mbytes	60 - 120 ns	\$5 - \$10
Secondary Storage	Magnetic Disk	8 Gbytes	10 - 20 ms	\$0.10-\$0.20

Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB
- Dynamic RAM (DRAM)
  - 50ns – 70ns, $20 – $75 per GB
- Magnetic disk
  - 5ms – 20ms, $0.20 – $2 per GB
- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk

Four Questions for Memory Hierarchy Designers

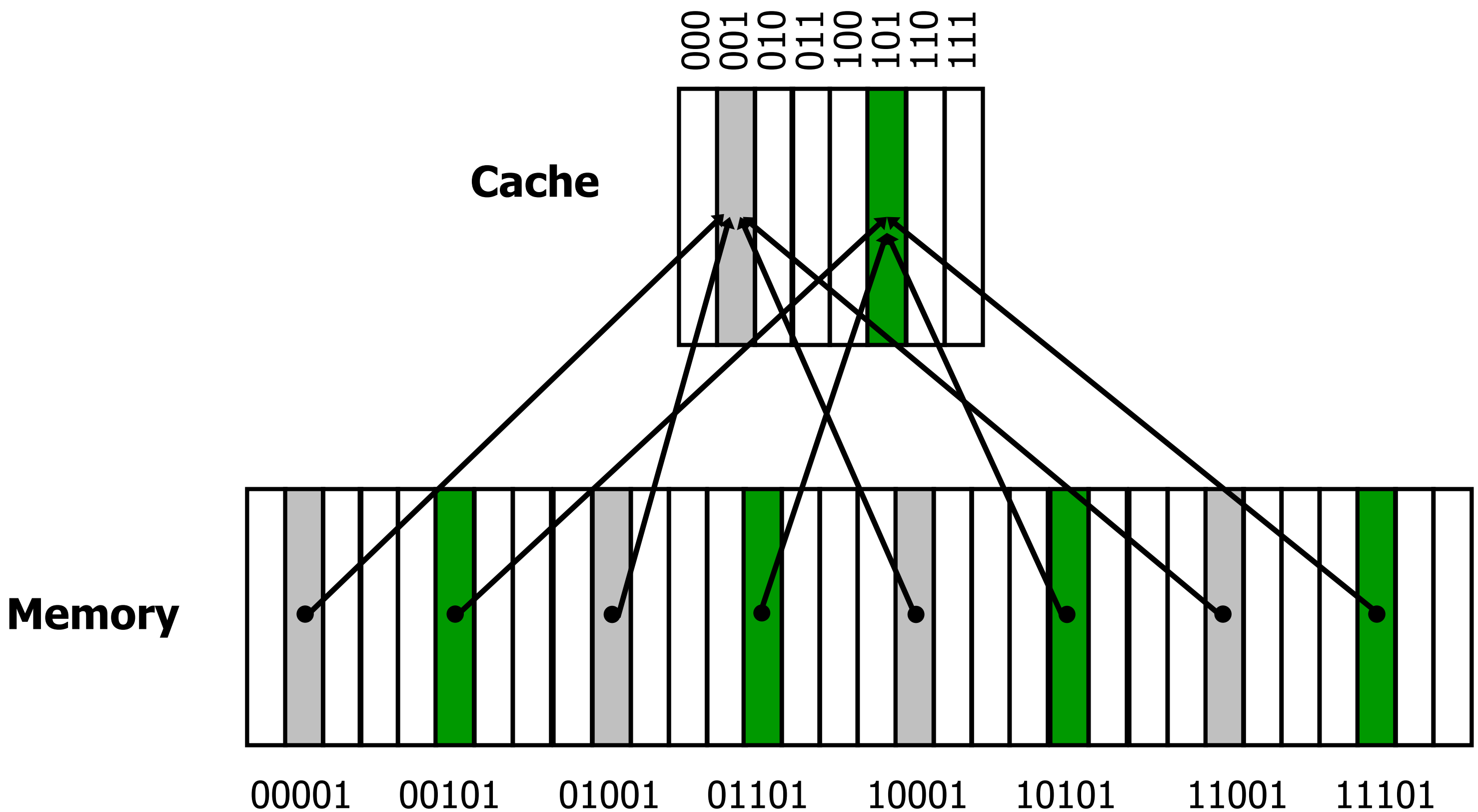
- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

Q1: Where can a block be placed?

- **Direct Mapped**: Each block has only one place that it can appear in the cache.
- **Fully associative**: Each block can be placed anywhere in the cache.
- **Set associative**: Each block can be placed in a restricted set of places in the cache.
  - If there are n blocks in a set, the cache is called n-way set associative
- What is the associativity of a direct mapped cache?

Direct Mapped Caches

- Mapping for direct mapped cache:
  (*Block address*) MOD (*Number of blocks in the cache*)



Cache Example

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

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

Cache Example

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

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

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

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	Hit	110
26	11 010	Hit	010

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	Hit	000

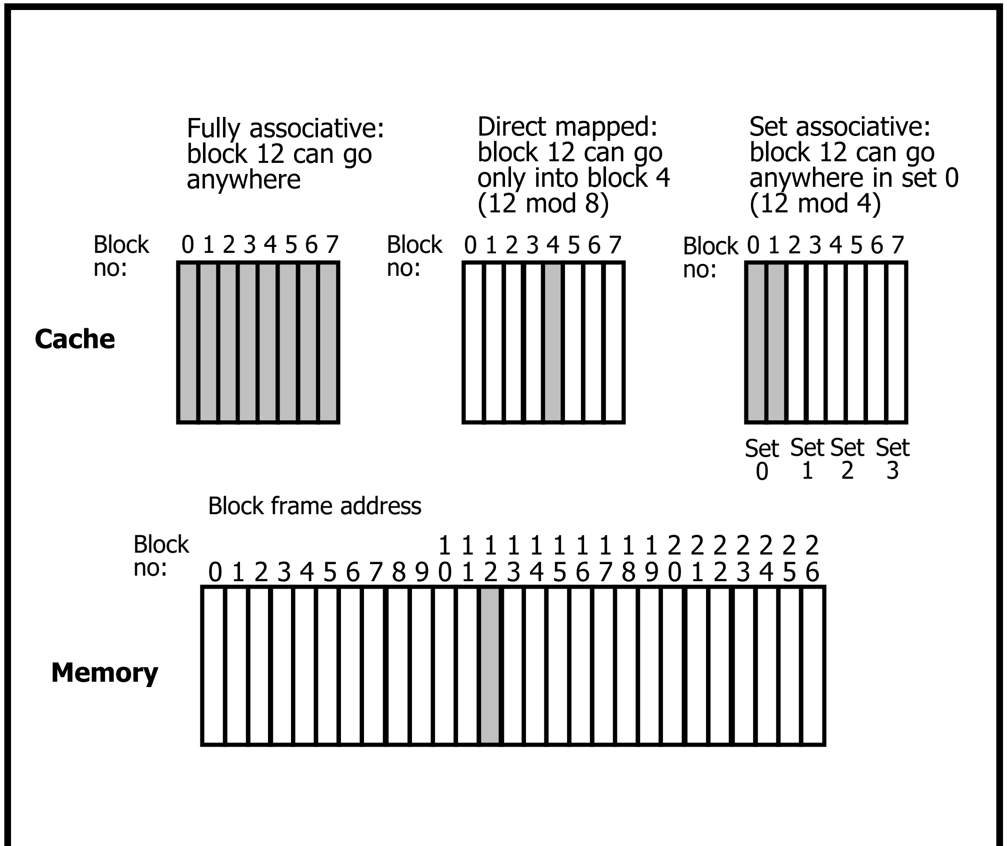
Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

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

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		

Associativity Examples



Cache size is 8 blocks
Where does word 12 from memory go?

Fully associative:
Block 12 can go anywhere

Direct mapped:
Block no. = (Block address) mod
(No. of blocks in cache)
Block 12 can go only into block 4
(12 mod 8 = 4)
=> Access block using lower 3 bits

2-way set associative:
Set no. = (Block address) mod
(No. of sets in cache)
Block 12 can go anywhere in set 0
(12 mod 4 = 0)
=> Access set using lower 2 bits

Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Q2: How Is a Block Found?

<table>
<tr><td colspan="2">Block address</td><td rowspan="2">Block offset</td></tr>
<tr><td>Tag</td><td>Index</td></tr>
</table>

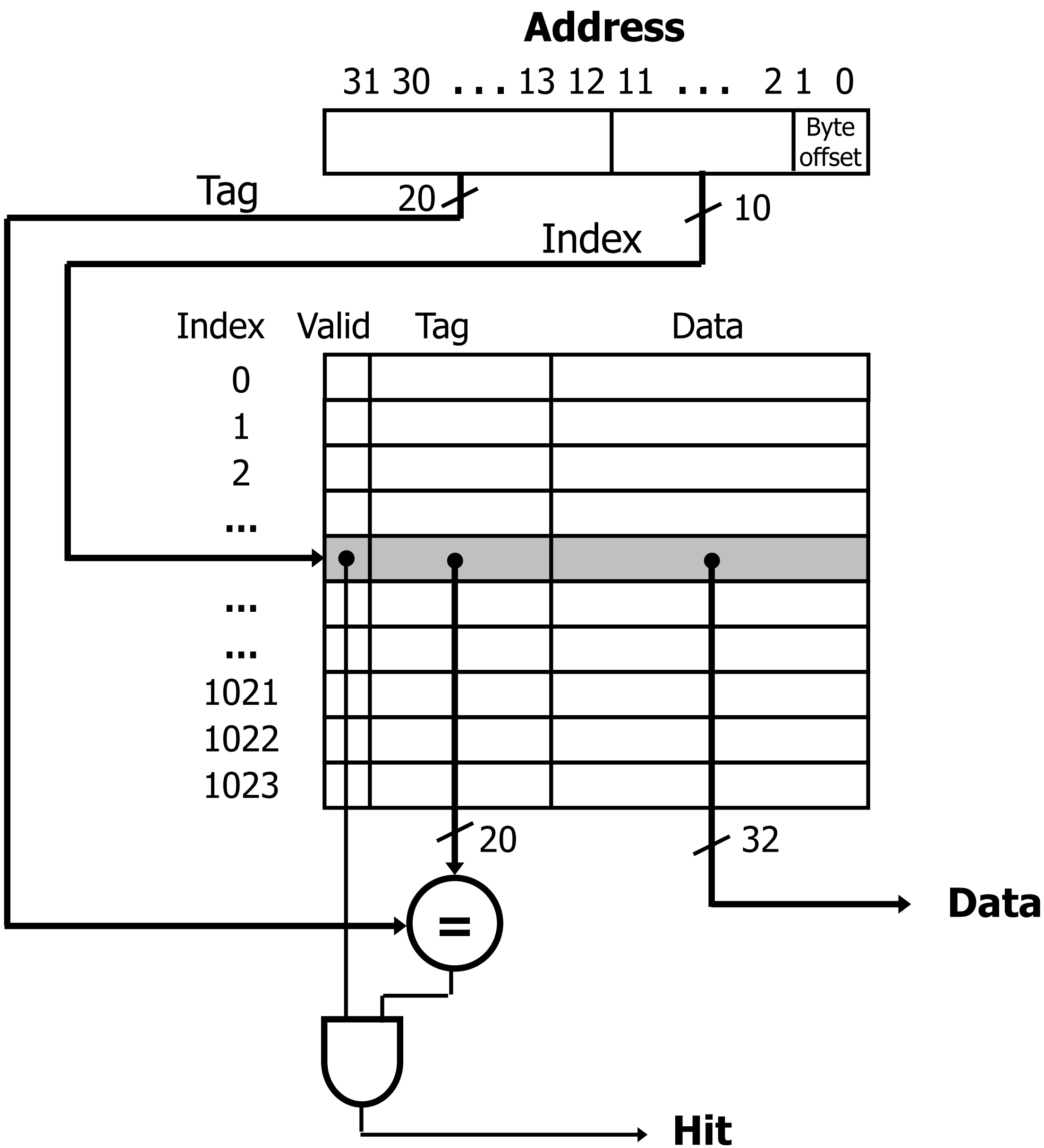
- The address can be divided into two main parts
  - **Block offset**: selects the data from the block

    offset size = $\log_2$(block size)
  - **Block address**: tag + index
    - index: selects set in cache

      index size = $\log_2$(#blocks/associativity)
    - tag: compared to tag in cache to determine hit

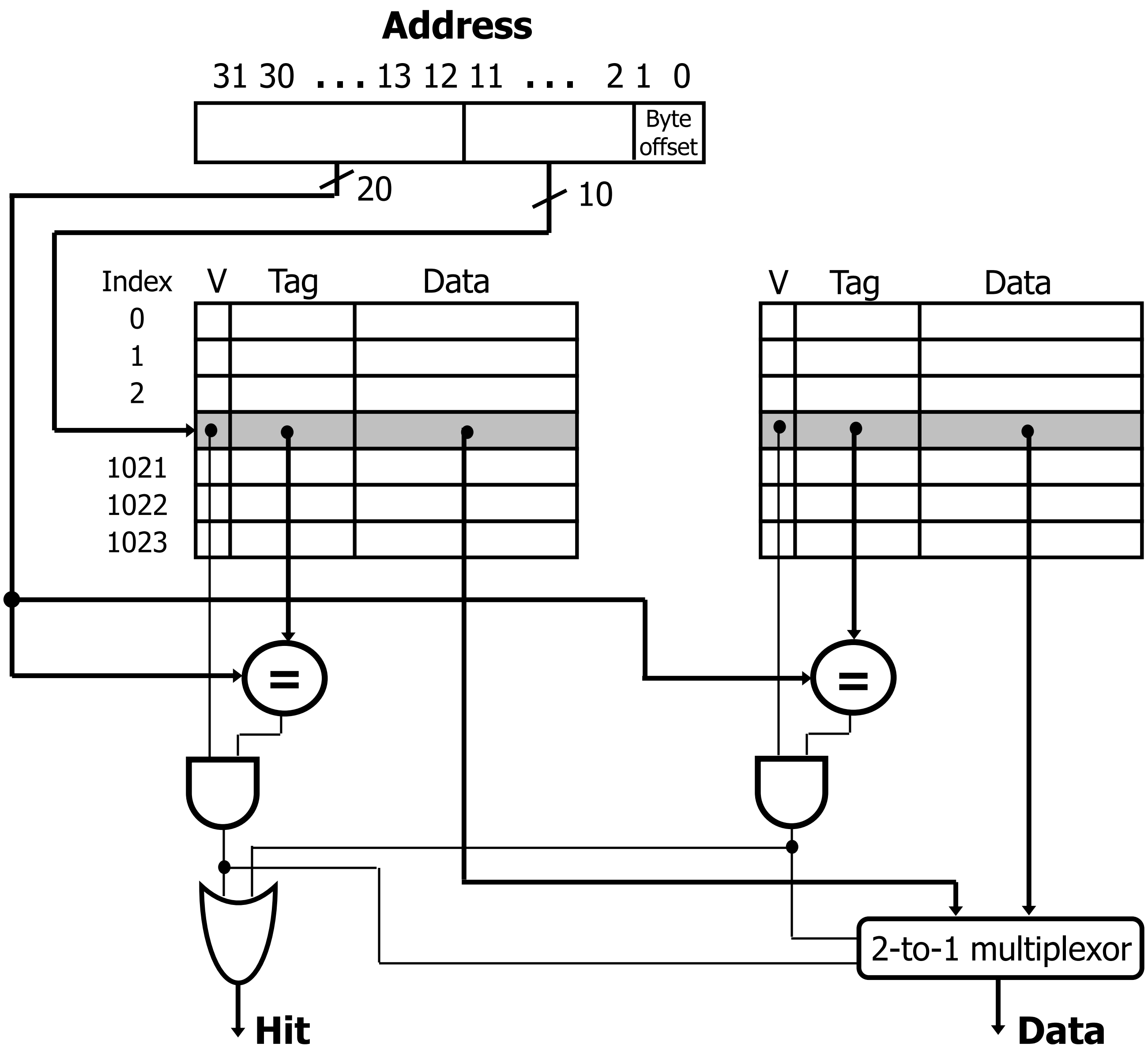
      tag size = address size - index size - offset size
- Each block has a valid bit that tells if the block is valid - the block is in the cache if the tags match and the valid bit is set.

A 4-KB Cache Using 1-word (4-byte) Blocks



- **Cache index** is used to select the block
- **Tag field** is used to compare with the value of the tag filed of the cache
- **Valid** bit indicates if a cache block have valid information

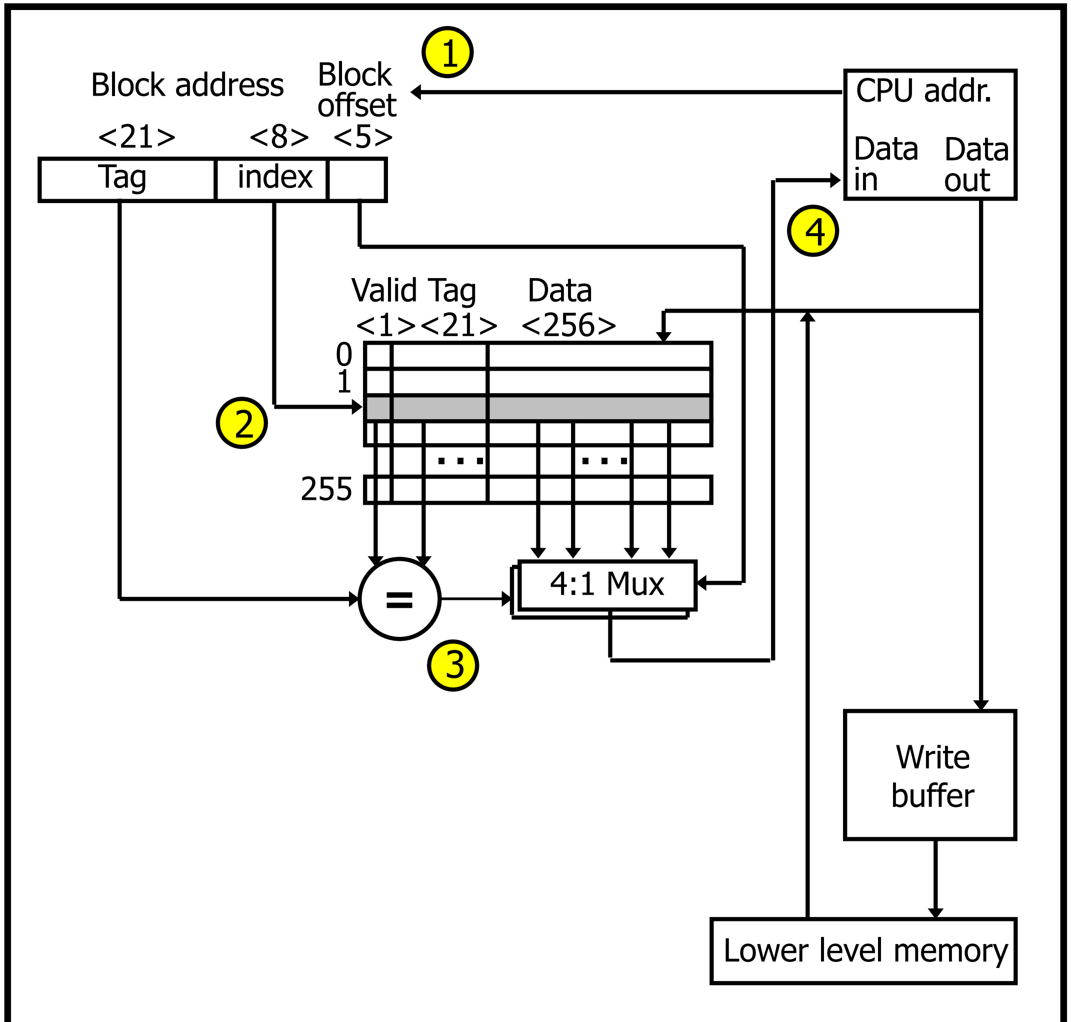
Two-way Set-associative Cache



Example: Alpha 21064 Data Cache

- The data cache of the Alpha 21064 has the following features
  - 8 KB of data
  - 32 byte blocks
  - Direct mapped placement
  - Write through (no-write allocate, 4-block write buffer)
  - 34 bit physical address composed of
    - 5 bit block offset
    - 8 bit index
    - 21 bit tag

Example: Alpha 21064 Data Cache



A cache read has 4 steps

(1) The address from the cache is divided into the tag, index, and block offset

(2) The index selects block

(3) The address tag is compared with the tag in the cache, the valid bit is checked, and data to be loaded is selected

(4) If the valid bit is set, the data is loaded into the processor

If there is a write, the data is also sent to the write buffer

Q3: Which Block Should be Replaced on a Miss?

- Easy for Direct Mapped - only on choice
- Set Associative or Fully Associative:
  - Random - easier to implement
  - Least Recently Used (the block has been unused for the longest time) - harder to implement
- Miss rates for caches with different size, associativity and replacement algorithm.

Associativity:	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

For caches with low miss rates, random is almost as good as LRU.

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Q4: What Happens on a Write?

- Write through: The information is written to both the block in the cache and to the block in the lower-level memory.

- Write back: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty? (add a dirty bit to each block)

Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = $1 + 0.1 \times 100 = 11$
- 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
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first

Pros and Cons of each:

- Write through
  - Read misses cannot result in writes to memory,
  - Easier to implement
  - Always combine with write buffers to avoid memory latency
- Write back
  - Less memory traffic
  - Perform writes at the speed of the cache

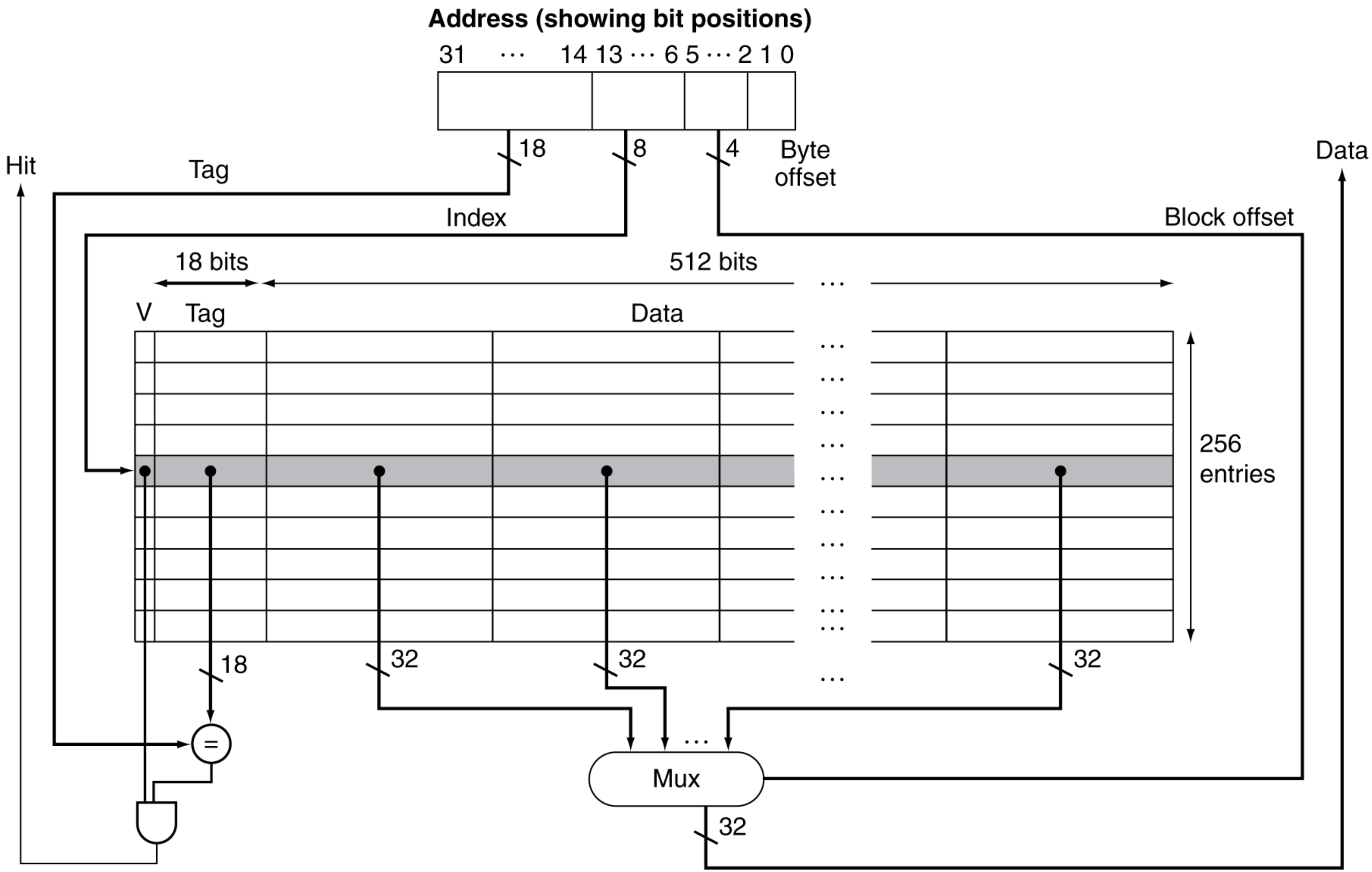
Q4: What Happens on a Write? CONT'D

- Since data does not have to be brought into the cache on a write miss, there are two options:
  - Write allocate
    - The block is brought into the cache on a write miss
    - Used with write-back caches
    - Hope subsequent writes to the block hit in cache
  - No-write allocate
    - The block is modified in memory, but not brought into the cache
    - Used with write-through caches
    - Writes have to go to memory anyway, so why bring the block into the cache

Example: Intrinsity FastMATH

- Embedded MIPS processor
  - 12-stage pipeline
  - Instruction and data access on each cycle
- Split cache: separate I-cache and D-cache
  - Each 16KB: 256 blocks × 16 words/block
  - D-cache: write-through or write-back
- SPEC2000 miss rates
  - I-cache: 0.4%
  - D-cache: 11.4%
  - Weighted average: 3.2%

Example: Intrinsity FastMATH



Calculating Bits in Cache

- How many total bits are needed for a direct- mapped cache with 64 KBytes of data and one word blocks, assuming a 32-bit address?
  - 64 Kbytes = 16 K words = 2^14 words = 2^14 blocks
  - block size = 4 bytes => offset size = 2 bits,
  - #sets = #blocks = 2^14 => index size = 14 bits
  - tag size = address size - index size - offset size = 32 - 14 - 2 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 32 + 16 + 1 = 49
  - bits in cache = #blocks x (bits/block) = 2^14 x 49 = **98 Kbytes**
- How many total bits would be needed for a 4-way set associative cache to store the same amount of data
  - block size and #blocks does not change
  - #sets = #blocks/4 = (2^14)/4 = 2^12 => index size = 12 bits
  - tag size = address size - index size - offset = 32 - 12 - 2 = 18 bits
  - bits/block = data bits + tag bits + valid bit = 32 + 18 + 1 = 51
  - bits in cache = #blocks x (bits/block) = 2^14 x 51 = **102 Kbytes**
- Increase associativity => increase bits in cache

Calculating Bits in Cache

- How many total bits are needed for a direct-mapped cache with 64 KBytes of data and 8 word blocks, assuming a 32-bit address?
  - 64 Kbytes = 2^14 words = (2^14)/8 = 2^11 blocks
  - block size = 32 bytes => offset size = 5 bits,
  - #sets = #blocks = 2^11 => index size = 11 bits
  - tag size = address size - index size - offset size = 32 - 11 - 5 = 16 bits
  - bits/block = data bits + tag bits + valid bit = 8x32 + 16 + 1 = 273 bits
  - bits in cache = #blocks x (bits/block) = 2^11 x 273 = 68.25 Kbytes
- Increase block size => decrease bits in cache

Summary

- CPU-Memory gap is major performance obstacle for achieving high performance
- Memory hierarchies
  - Take advantage of program locality
  - Closer to processor => smaller, faster, more expensive
  - Further from processor => bigger, slower, less expensive
- 4 questions for memory hierarchy
  - Block placement, block identification, block replacement, and write strategy
- Cache parameters
  - Cache size, block size, associativity