CS152: Computer Systems Architecture Memory System and Caches

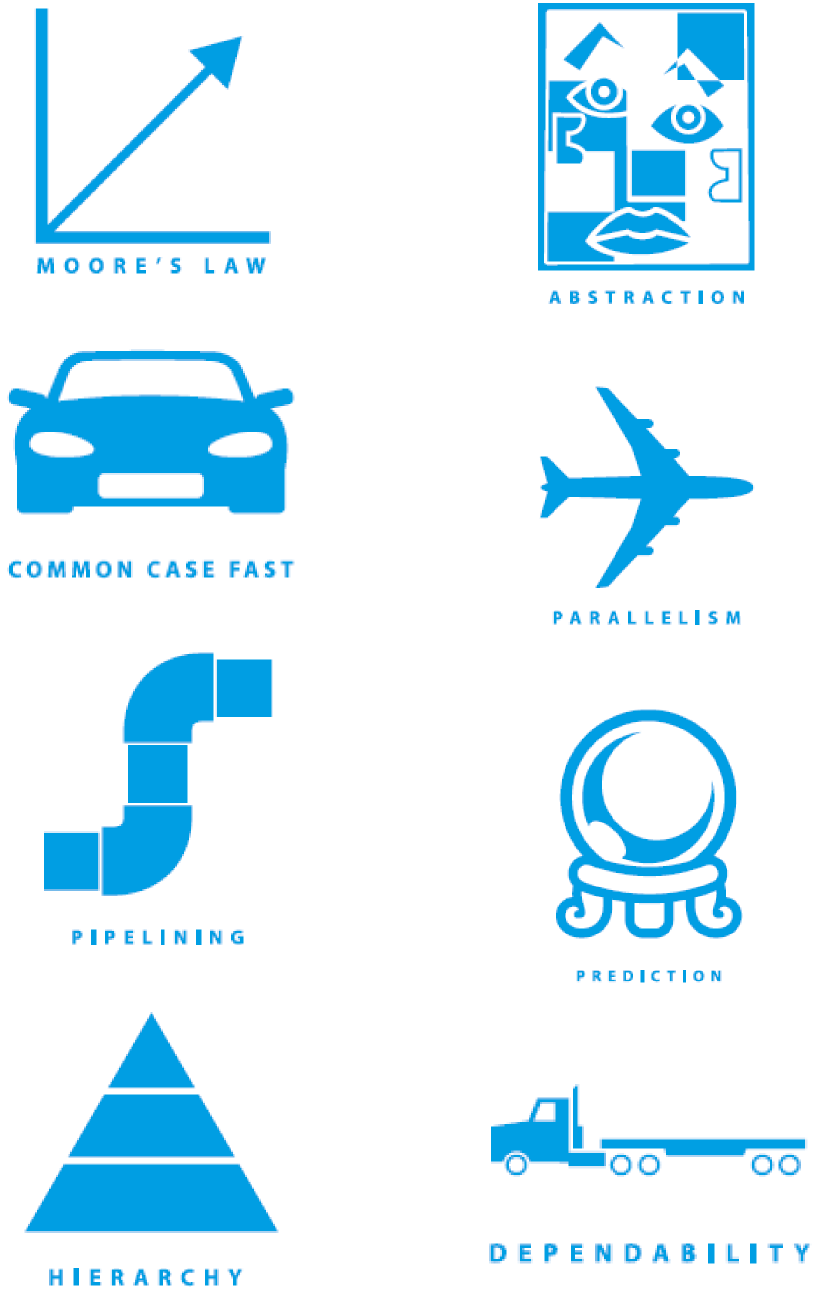
Sang-Woo Jun

Winter 2021

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Eight great ideas

- □ Design for Moore’s Law
- □ Use abstraction to simplify design
- □ Make the common case fast
- □ Performance via parallelism
- □ Performance via pipelining
- □ Performance via prediction
- □ Hierarchy of memories
- □ Dependability via redundancy



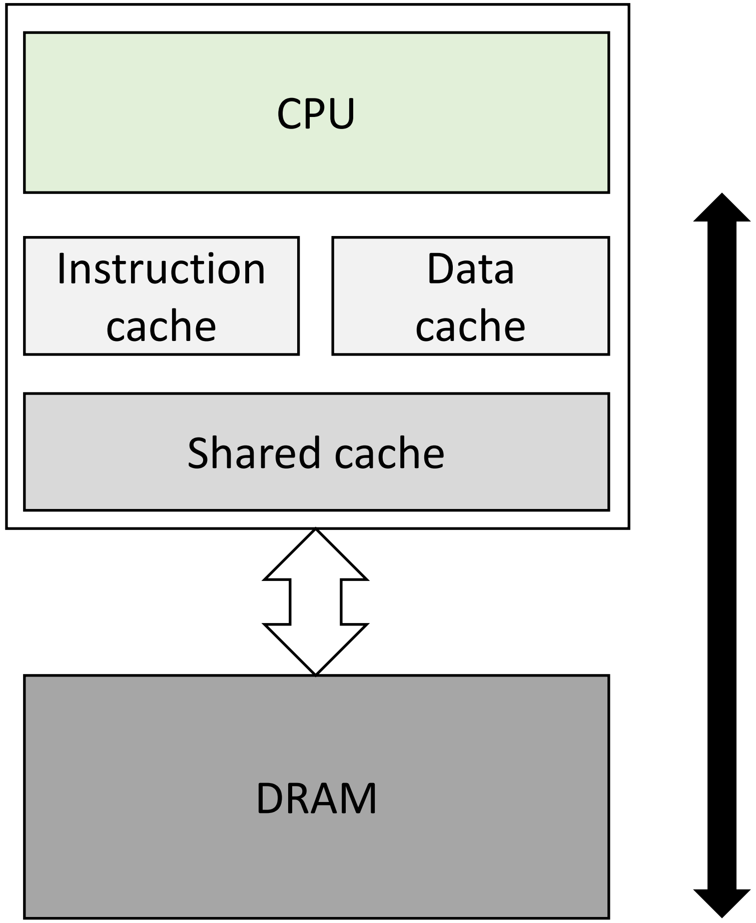
Caches are important

“There are only two hard things in computer science:

1. Cache invalidation,
2. Naming things,
3. and off-by-one errors”

Original quote (with only the first two points) by Phil Karlton
I couldn’t find joke source

A modern computer has a hierarchy of memory



Low latency (~1 cycle)
Small (KBs)
Expensive ($1000s per GB)

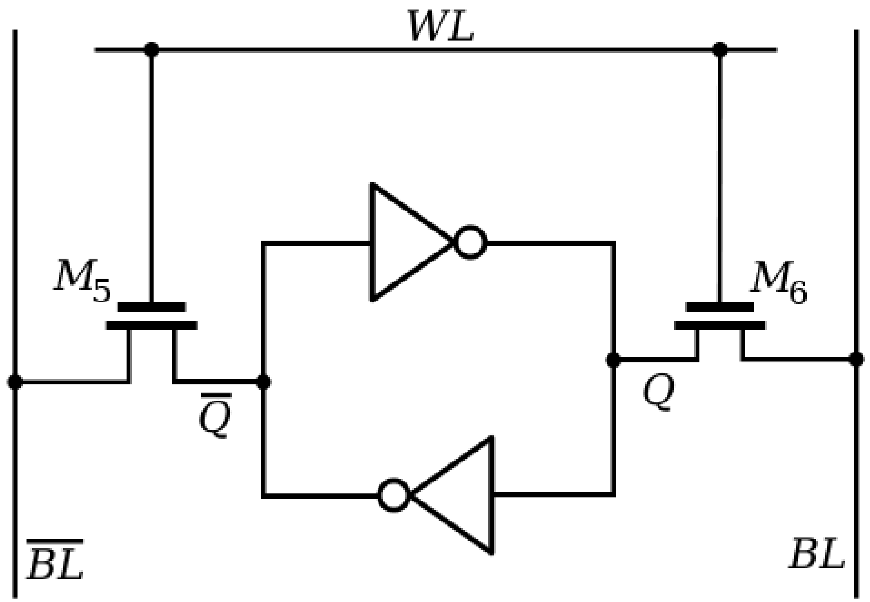
High latency (100s~1000s of cycles)
Large (GBs)
Cheap (<$5 per GB)

Cost prohibits having a lot of fast memory

Ideal memory:
As cheap and large as DRAM (Or disk!)
As fast as SRAM
…Working on it!

What causes the cost/performance difference? – SRAM

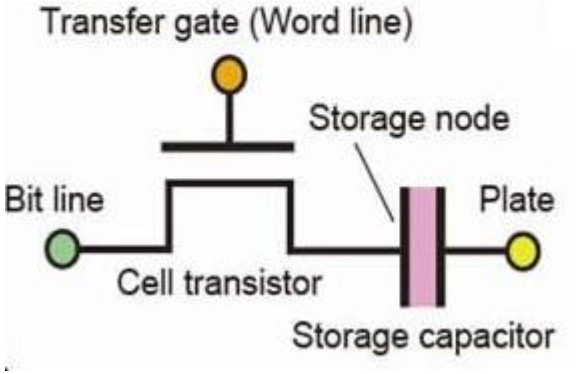
- SRAM (Static RAM) vs. DRAM (Dynamic RAM)
- SRAM is constructed entirely out of transistors
  - Accessed in clock-synchronous way, just like any other digital component
  - Subject to propagation delay, etc, which makes large SRAM blocks expensive and/or slow

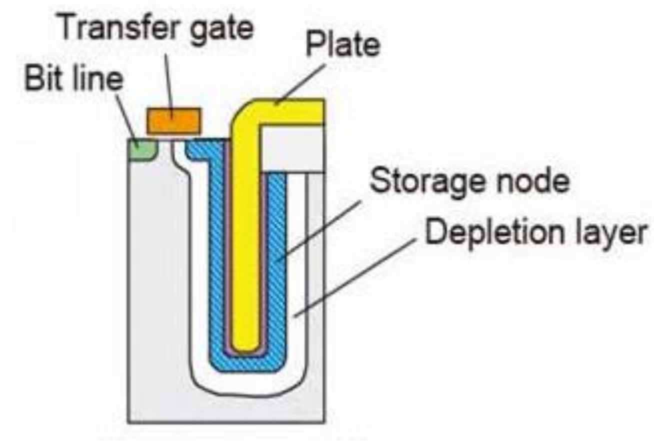


Source: Inductiveload, from commons.wikimedia.org

What causes the cost/performance difference? – DRAM

- ❑ DRAM stores data using a capacitor
  - o Very small/dense cell
  - o A capacitor holds charge for a short while, but slowly leaks electrons, losing data
  - o To prevent data loss, a controller must periodically read all data and write it back (“Refresh”)
    - Hence, “Dynamic” RAM
  - o Requires fab process separate from processor
- ❑ Reading data from a capacitor is high-latency
  - o EE topics involving sense amplifiers, which we won’t get into



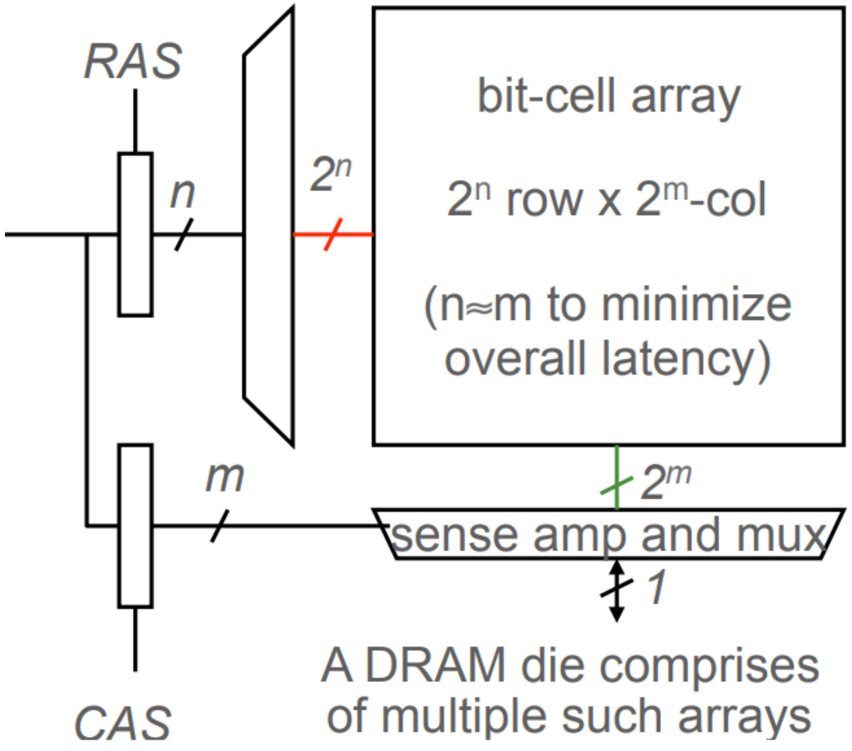


Note: Old, “trench capacitor” design

Source: Dailytech

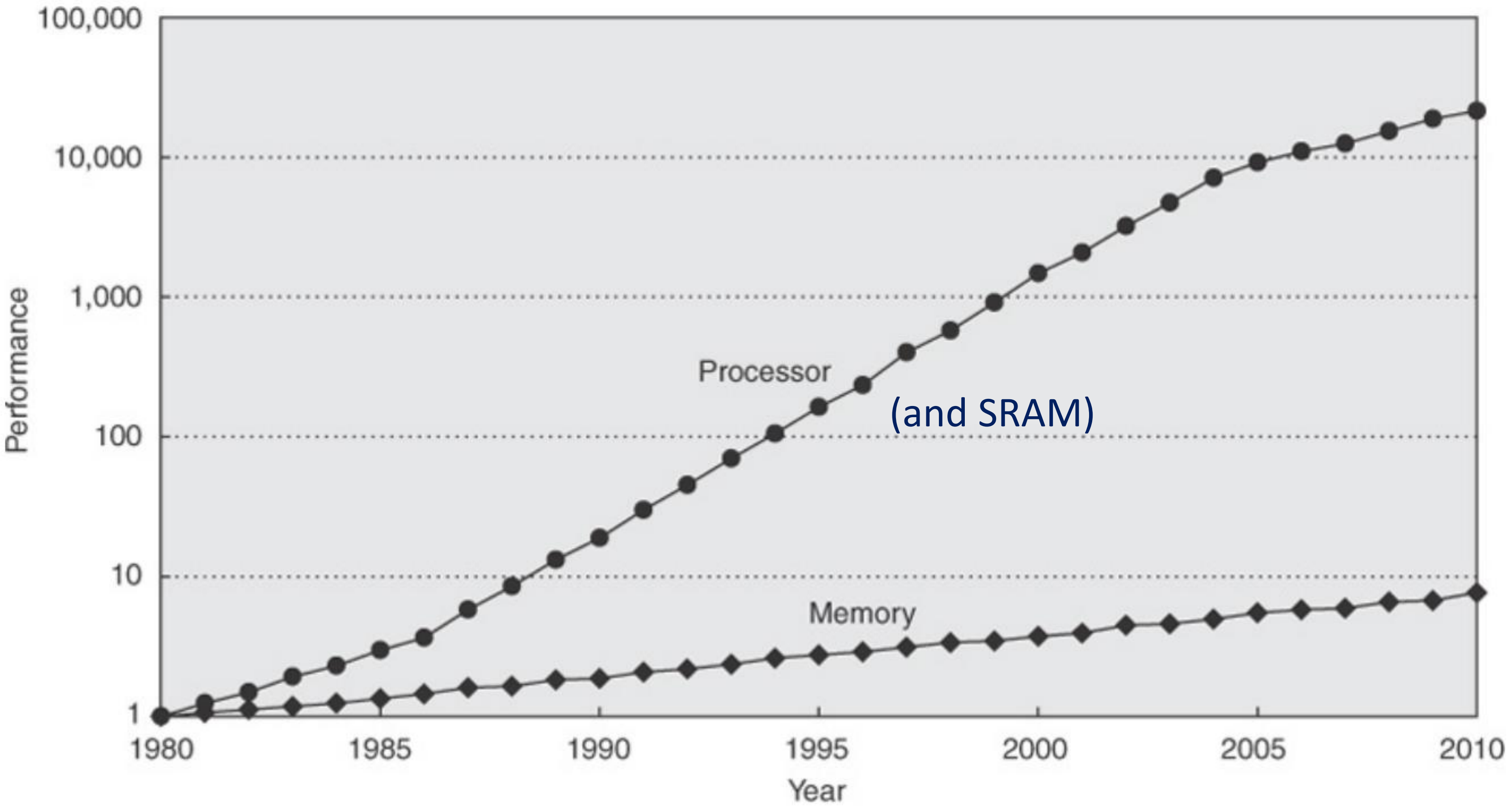
What causes the cost/performance difference? – DRAM

- ❑ DRAM is typically organized into a rectangle (rows, columns)
  - Reduces addressing logic, which is a high overhead in such dense memory
  - Whole row must be read whenever data in new row is accessed
  - As of 2020, typical row size ~8 KB
- ❑ Fast when accessing data in same row, order of magnitude slower when accessing small data across rows
  - Accessed row temporarily stored in DRAM “row buffer”



A DRAM die comprises of multiple such arrays

And the gap keeps growing



© 2007 Elsevier, Inc. All rights reserved.

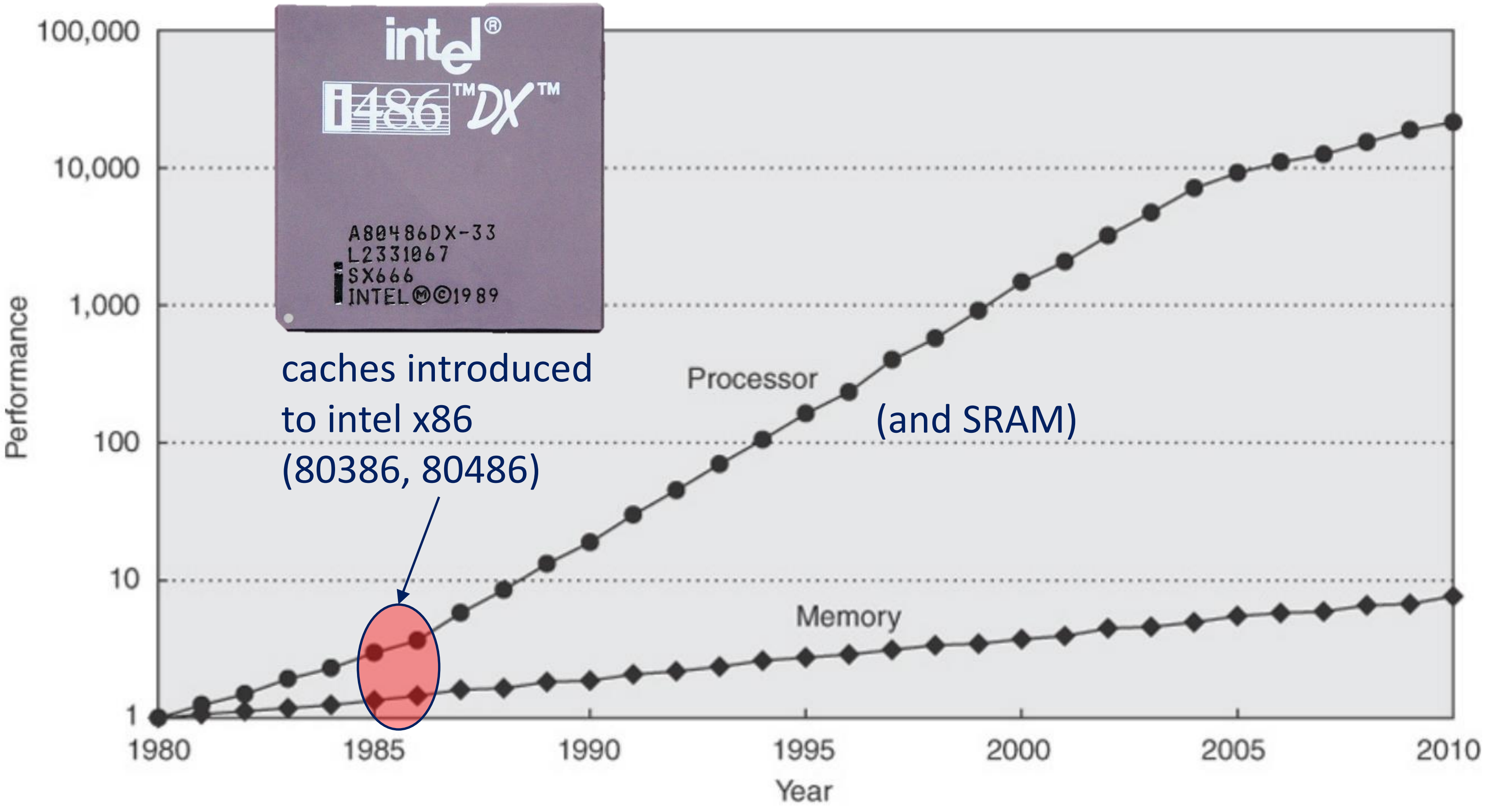
Goals of a memory system

- ❑ Performance at reasonable cost
  - o Capacity of DRAM, but performance of SRAM
- ❑ Simple abstraction
  - o CPU should be oblivious to type of memory
  - o Should not make software/compiler responsible for identifying memory characteristics and optimizing for them, as it makes performance not portable
    - Unfortunately this is not always possible, but the hardware does its best!

Introducing caches

- The CPU is (largely) unaware of the underlying memory hierarchy
  - The memory abstraction is a single address space
  - The memory hierarchy automatically stores data in fast or slow memory, depending on usage patterns
- Multiple levels of “caches” act as interim memory between CPU and main memory (typically DRAM)
  - Processor accesses main memory through the cache hierarchy
  - If requested address is already in the cache (address is “cached”, resulting in “cache hit”), data operations can be fast
  - If not, a “cache miss” occurs, and must be handled to return correct data to CPU

And the gap keeps growing



caches introduced to intel x86 (80386, 80486)

(and SRAM)

© 2007 Elsevier, Inc. All rights reserved.

Cache operation

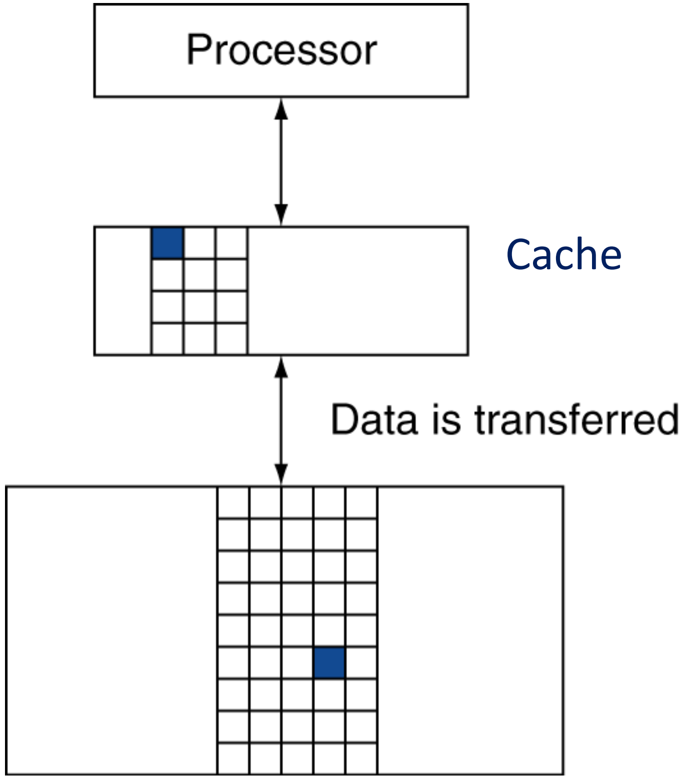
- ❑ One of the most intensely researched fields in computer architecture
- ❑ Goal is to somehow make to-be-accessed data available in fastest possible cache level at access time
    - ○ Method 1: Caching recently used addresses
        - Works because software typically has **"Temporal Locality"** : If a location has been accessed recently, it is likely to be accessed (reused) soon
    - ○ Method 2: Pre-fetching based on future pattern prediction
        - Works because software typically has **"Spatial Locality"** : If a location has been accessed recently, it is likely that nearby locations will be accessed soon
    - ○ Many, many more clever tricks and methods are deployed!

Average Memory Access Time = HitTime + MissRatio × MissPenalty

Basic cache operations

- ❑ Unit of caching: “Block” or “Cache line”
  - May be multiple words -- 64 Bytes in modern Intel x86
- ❑ If accessed data is present in upper level
  - Hit: access satisfied by upper level
- ❑ If accessed data is absent
  - Miss: block copied from lower level
    - Time taken: miss penalty
  - Then accessed data supplied from upper level

How does the memory system keep track of what is present in cache?



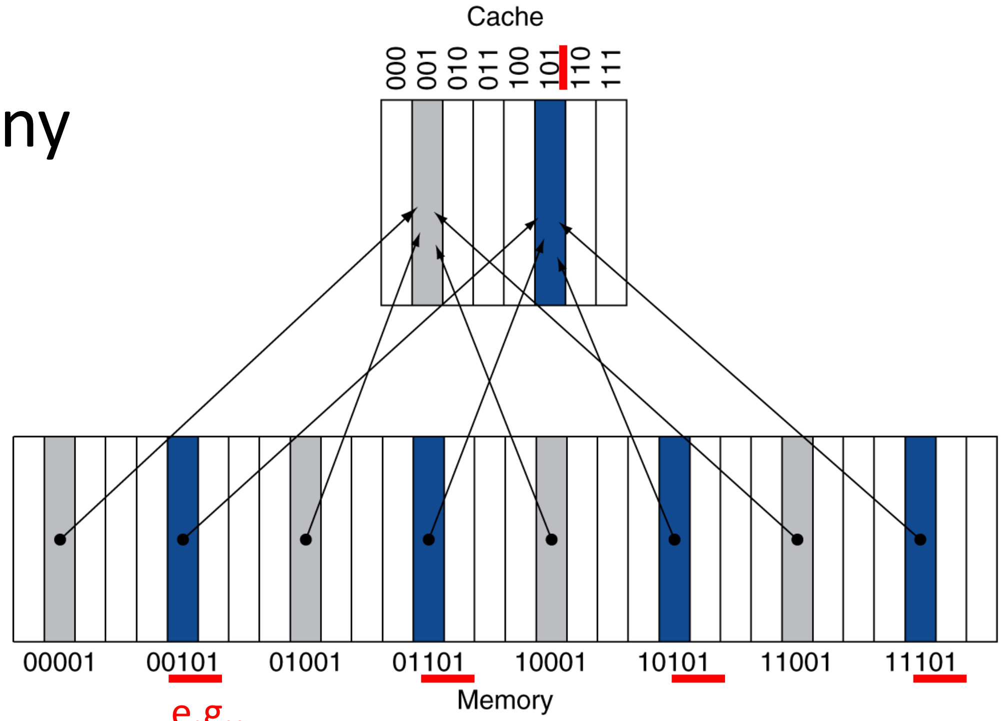
Main memory

A simple solution: “Direct Mapped Cache”

- ❑ Cache location determined by address
- ❑ Each block in main memory mapped on one location in cache memory (“Direct Mapped”)
  - ○ “Direct mapped”
- ❑ Cache is smaller than main memory, so many DRAM locations map to one cache location

(Cache address$_{\text{block}}$)
= (main memory address$_{\text{block}}$) mod (cache size$_{\text{block}}$)

Since cache size is typically power of two,
Cache address is lower bits of block address



e.g.,

Selecting index bits

- ❑ Why do we chose low order bits for index?
  - Allows consecutive memory locations to live in the cache simultaneously
  - Reduces likelihood of replacing data that may be accessed again in the near future
  - Helps take advantage of locality

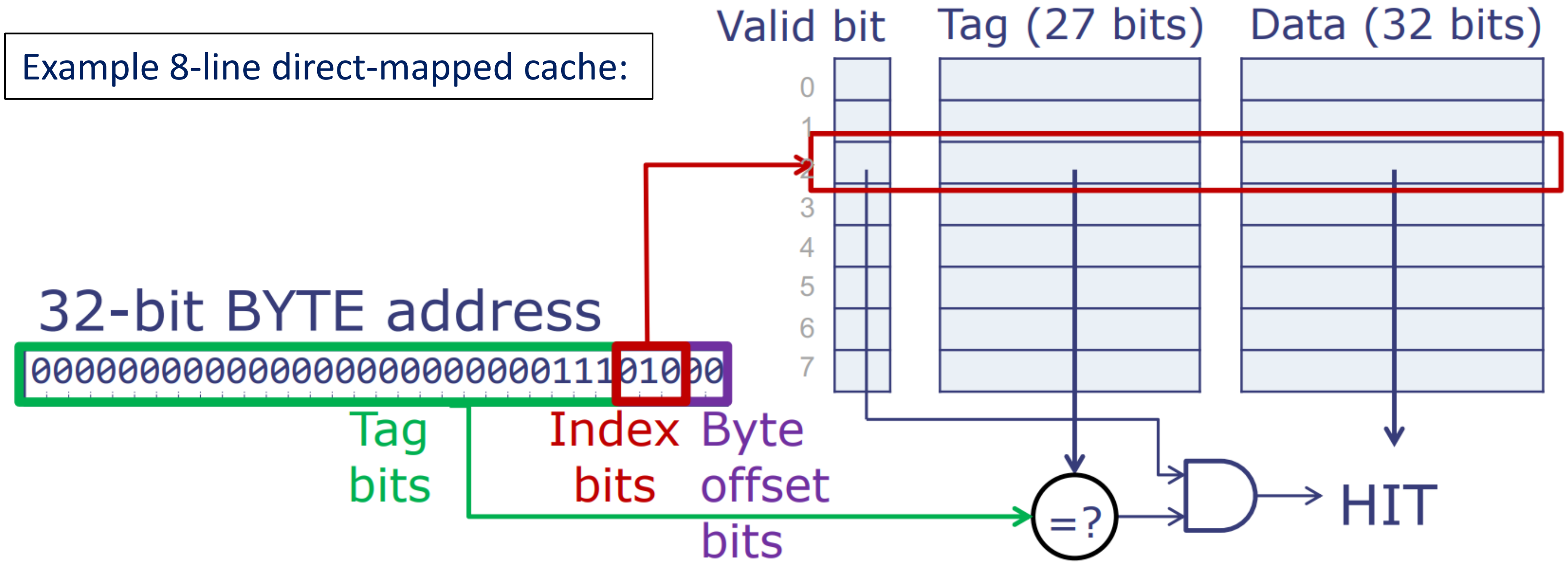
Tags and Valid Bits

- ❑ How do we know which particular block is stored in a cache location?
  - o Store block address as well as the data, compare when read
  - o Actually, only need the high-order bits (Called the “tag”)
- ❑ What if there is no data in a location?
  - o Valid bit: 1 = present, 0 = not present
  - o Initially 0

Direct Mapped Cache Access

- For cache with 2^W cache lines
  - Index into cache with W address bits (the index bits)
  - Read out valid bit, tag, and data
  - If valid bit == 1 and tag matches upper address bits, cache hit!

Example 8-line direct-mapped cache:



Direct Mapped Cache Access Example

- ❑ 64-line direct-mapped cache -> 64 indices -> 6 index bits
- ❑ Example 1: Read memory 0x400C

$$0x400C = \underbrace{0100\ 0000}_{\text{Tag: 0x40}}\ \underbrace{0000\ 11}_{\text{Index: 0x3}}\underbrace{00}_{\text{Byte offset: 0x0}}$$

-> **Cache hit!** Data read 0x42424242

- ❑ Example 2: Read memory 0x4008

$$0x4008 = \underbrace{0100\ 0000}_{\text{Tag: 0x40}}\ \underbrace{0000\ 10}_{\text{Index: 0x2}}\underbrace{00}_{\text{Byte offset: 0x0}}$$

-> **Cache miss! Tag mismatch**

	Valid bit	Tag (24 bits)	Data (32 bits)
0	1	0x000058	0xDEADBEEF
1	1	0x000058	0x00000000
2	1	0x000058	0x00000007
3	1	0x000040	0x42424242
4	0	0x000007	0x6FBA2381
	⋮	⋮	⋮
63	1	0x000058	0xF7324A32

Direct Mapped Cache Access Example

- 8-blocks, 1 word/block, direct mapped
- Initial state: All “valid” bits are set to invalid

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

Direct Mapped Cache Access Example

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

Cache miss! Main memory read to cache

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

Direct Mapped Cache Access Example

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

Cache miss! Main memory read to cache

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		

Direct Mapped Cache Access Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Cache hit! No main memory read

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		

Direct Mapped Cache Access Example

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

Cache misses result in main memory read

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		

Direct Mapped Cache Access Example

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

Cache collision results in eviction of old value

What if old value was written to?
Written data must be saved to main memory!

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		

Write Policies

- ❑ Write Through: Write is applied to cache, and applied immediately to memory
  - ○ + Simple to implement!
  - ○ - Wastes main memory bandwidth
- ❑ Write Back: Write is only applied to cache, write is applied only when evicted
  - ○ Cache line has another metadata bit “Dirty” to remember if it has been written
  - ○ + Efficient main memory bandwidth
  - ○ - Complex
  - ○ More common in modern systems

Write Back Example: Cache Hit/Miss

- ❑ 64-line direct-mapped cache -> 64 indices -> 6 index bits
- ❑ Write 0x9 to 0x480C
  - ○ $\underbrace{0100\ 1000}_{\text{Tag: 0x48}}\ \underbrace{0000\ 11}_{\text{Index: 0x3}}\underbrace{00}_{\text{Byte offset: 0x0}}$ **-> Cache hit!**
- ❑ Write 0x1 to 0x490C
  - ○ $\underbrace{0100\ 1001}_{\text{Tag: 0x49}}\ \underbrace{0000\ 11}_{\text{Index: 0x3}}\underbrace{00}_{\text{Byte offset: 0x0}}$ **-> Cache miss!** (Tag mismatch)

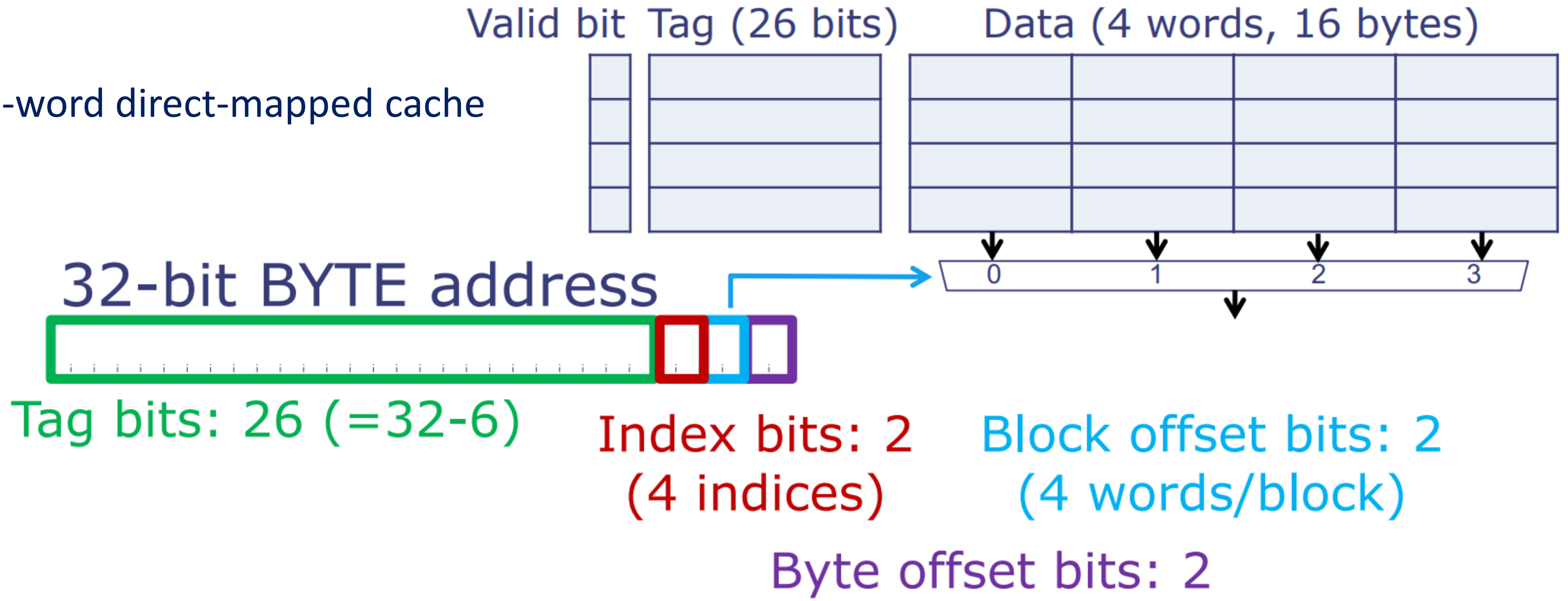
	V	D	Tag	Data
0	1	1		
1	1	0		
2	0	0		
3	1	1	0x49	0x1
⋮				
63	0	0		

Cache line 3 must be written to main memory, and then apply write to cache

Larger block (cache line) sizes

- ❑ Take advantage of spatial locality: Store multiple words per data line
  - ○ Always fetch entire block (multiple words) from memory
  - ○ Another advantage: Reduces size of tag memory!
  - ○ Disadvantage: Fewer indices in the cache -> Higher miss rate!

Example: 4-block, 16-word direct-mapped cache



Valid bit Tag (26 bits) Data (4 words, 16 bytes)

32-bit BYTE address

Tag bits: 26 (=32-6)

Index bits: 2
(4 indices)

Block offset bits: 2
(4 words/block)

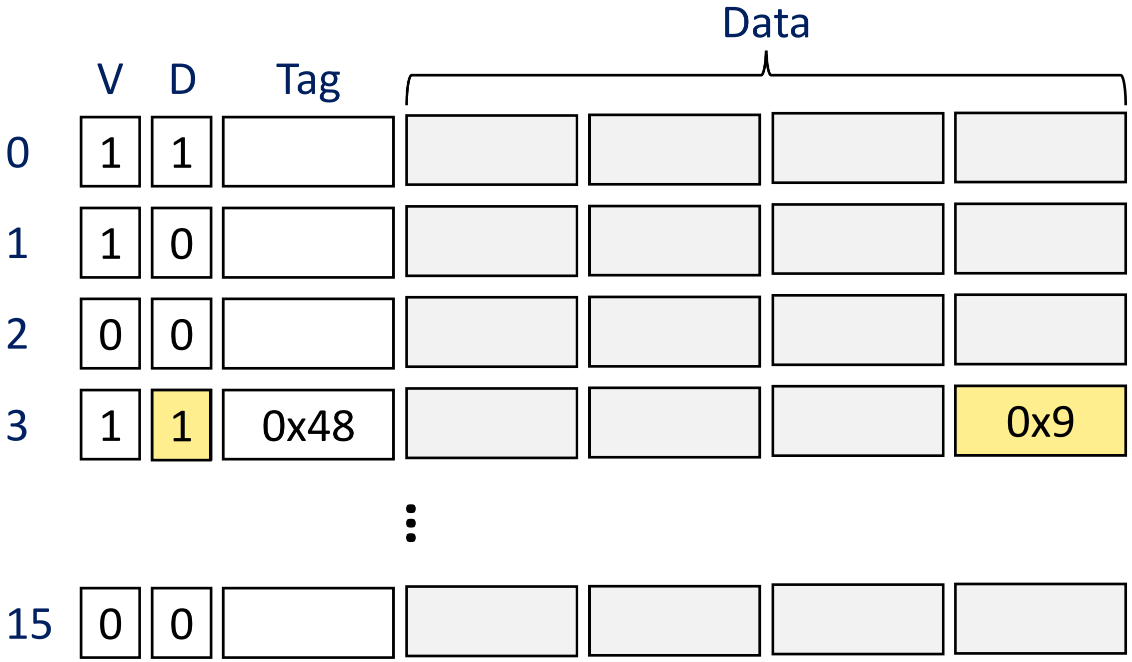
Byte offset bits: 2

Cache miss with larger block

- 64 elements with block size == 4 words
  - 16 cache lines, 4 index bits
- Write 0x9 to 0x483C
  - $\underbrace{0100\ 1000}_{\text{Tag: 0x48}}\ \underbrace{0011}_{\text{Index: 0x3}}\ \underbrace{11}_{\text{Block offset: 0x3}}00$

    -> Cache hit!
- Write 0x1 to 0x4938
  - $\underbrace{0100\ 1001}_{\text{Tag: 0x49}}\ \underbrace{0011}_{\text{Index: 0x3}}\ \underbrace{10}_{\text{Block offset: 0x2}}00$

    -> Cache miss!

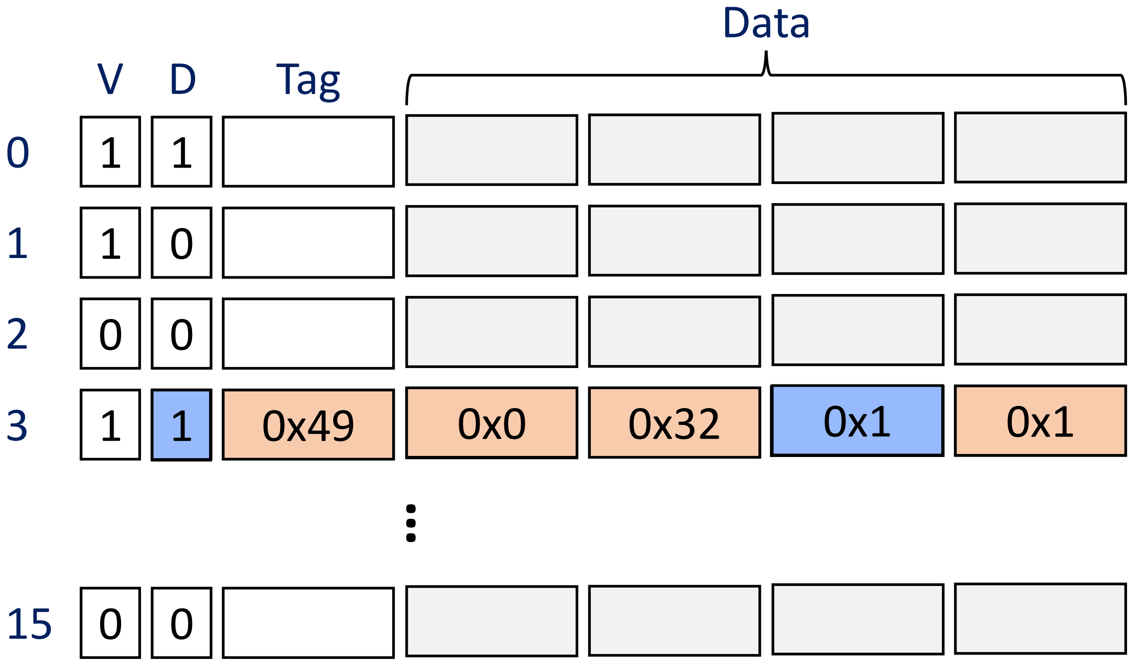


	V	D	Tag	Data			
0	1	1					
1	1	0					
2	0	0					
3	1	1	0x48				0x9
			⋮				
15	0	0					

Cache miss with larger block

- ❑ Write 0x1 to 0x4938
  - ○ $\underbrace{0100\ 1001}_{\text{Tag: 0x49}}\ \underbrace{0011}_{\text{Index: 0x3}}\ \underbrace{10}_{\text{Block offset: 0x2}}00$
- ❑ Since D == 1,
  - ○ Write cache line 3 to memory (All four words)
  - ○ Load cache line from memory (All four words)
  - ○ Apply write to cache

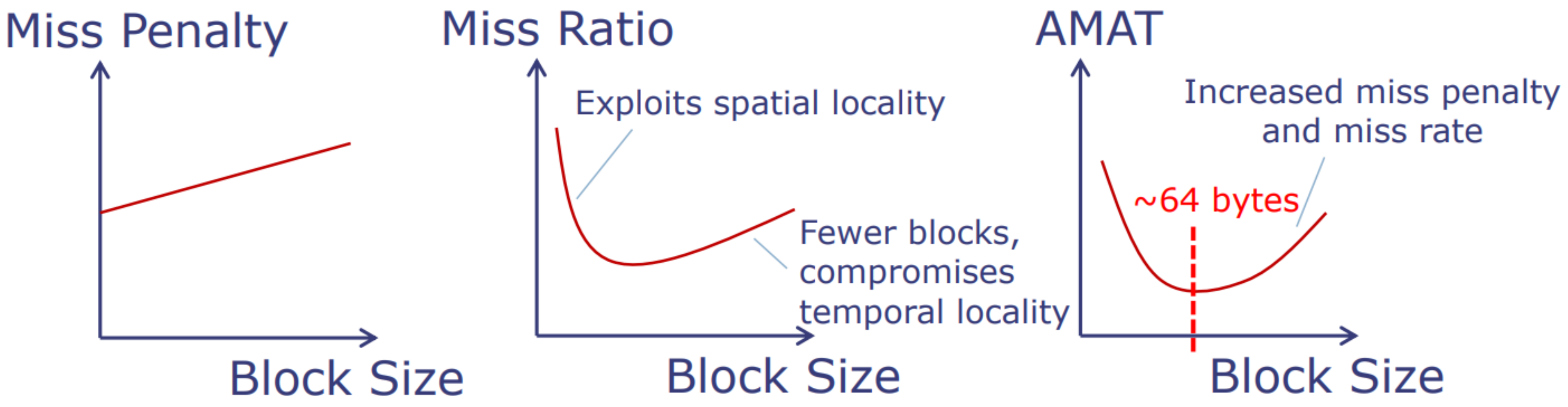
Writes/Reads four data elements just to write one!



	V	D	Tag	Data			
0	1	1					
1	1	0					
2	0	0					
3	1	1	0x49	0x0	0x32	0x1	0x1
			⋮				
15	0	0					

Block size trade-offs

- Larger block sizes…
  - Take advantage of spatial locality (also, DRAM is faster with larger blocks)
  - Incur larger miss penalty since it takes longer to transfer the block from memory
  - Can increase the average hit time and miss ratio
- AMAT = HitTime + MissPenalty*MissRatio



Looking back…

- ❑ Caches for high performance at low cost
  - ○ Exploits temporal locality in many programs
  - ○ Caches recently used data in fast, expensive memory
- ❑ Looked at “direct mapped” caches
  - ○ Cache slot to use was singularly determined by the address in main memory
  - ○ Uses tags and valid bits to correctly match data in cache and main memory
- ❑ Cache blocks (or “cache lines”) typically larger than a word
  - ○ Reduces tag size, better match with backing DRAM granularity
  - ○ Exploits spatial locality, up to a certain size (~64 bytes according to benchmarks)

Given a fixed space budget on the chip for cache memory, is this the most efficient way to manage it?

Direct-Mapped Cache Problem: Conflict Misses

- ❑ Assuming a 1024-line direct-mapped cache, 1-word cache line
- ❑ Consider steady state, after already executing the code once
  - What can be cached has been cached

- ❑ Conflict misses:
  - Multiple accesses map to same index!

We have enough cache capacity, just inconvenient access patterns

Loop A: Code at 1024, data at 37

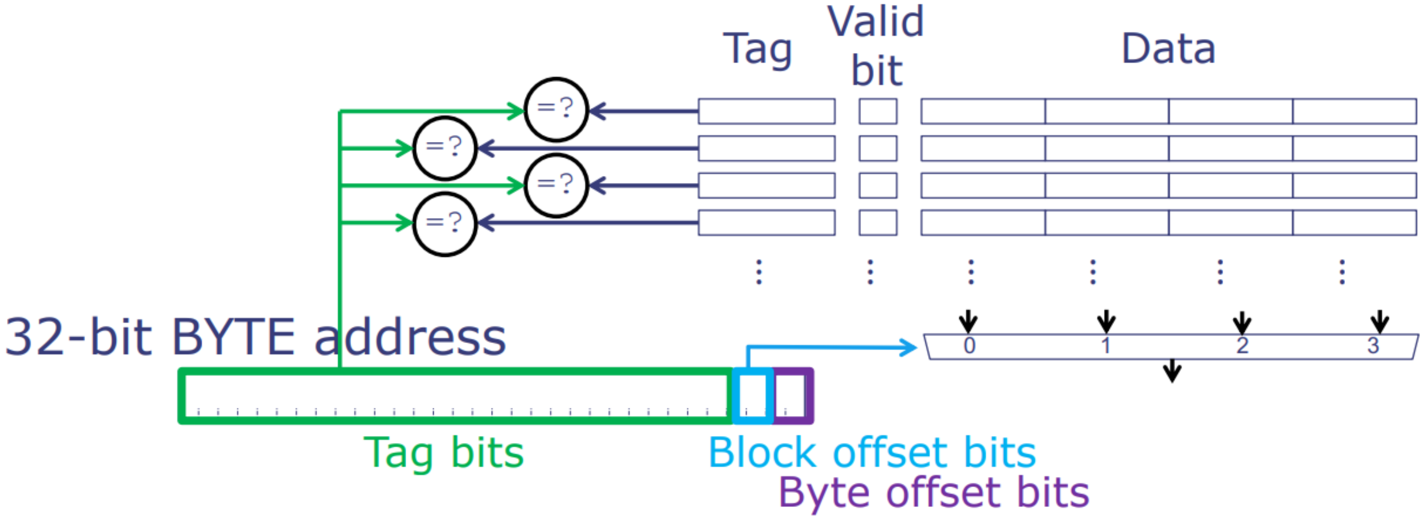
Word Address	Cache Line index	Hit/ Miss
1024	0	HIT
37	37	HIT
1025	1	HIT
38	38	HIT
1026	2	HIT
39	39	HIT
1024	0	HIT
37	37	HIT
…		

Loop B: Code at 1024, data at 2048

1024	0	MISS
2048	0	MISS
1025	1	MISS
2049	1	MISS
1026	2	MISS
2050	2	MISS
1024	0	MISS
2048	0	MISS
…		

Other extreme: “Fully associative” cache

- ❑ Any address can be in any location
  - ○ No cache index!
  - ○ Flexible (no conflict misses)
  - ○ Expensive: Must compare tags of all entries in parallel to find matching one
- ❑ Best use of cache space (all slots will be useful)
- ❑ But management circuit overhead is too large



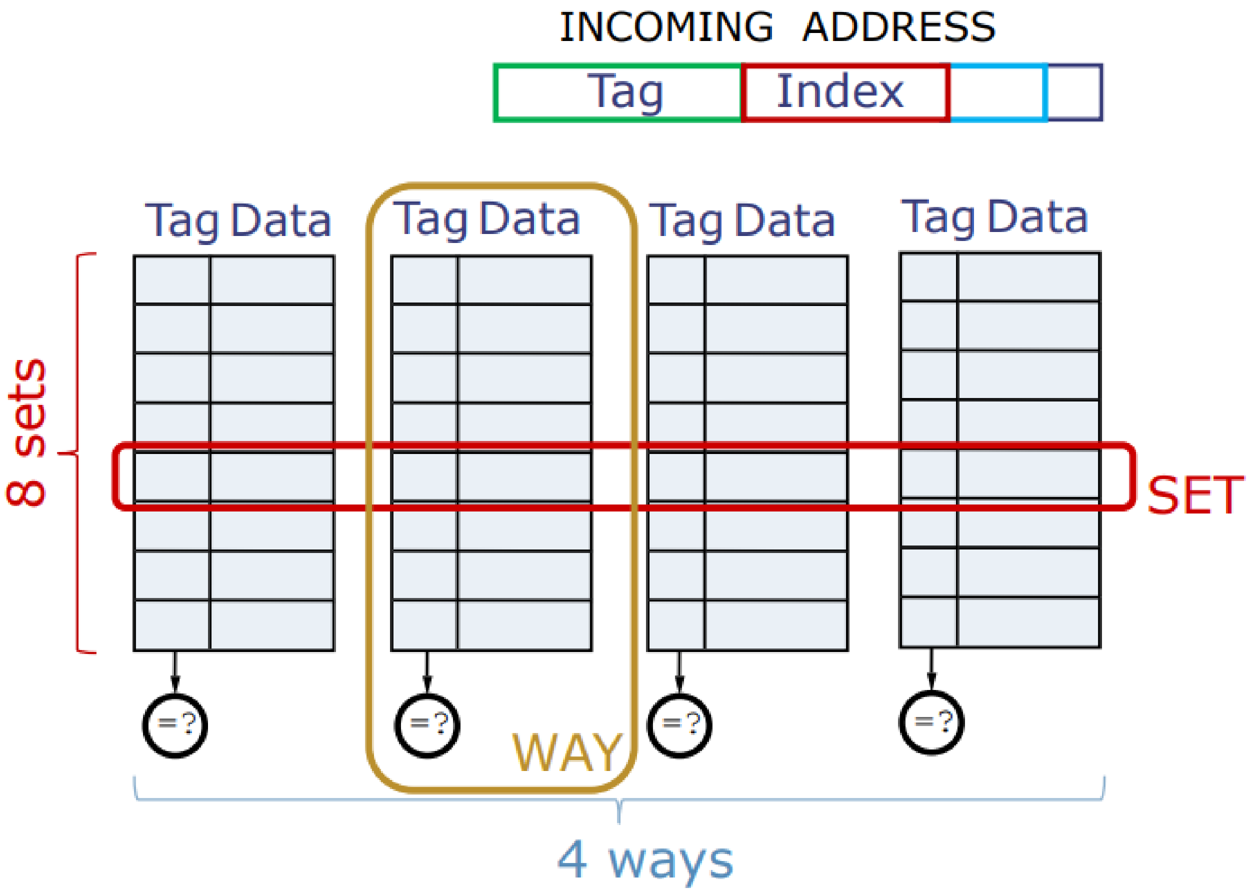
Three types of misses

- ❑ Compulsory misses (aka cold start misses)
  - ○ First access to a block
- ❑ Capacity misses
  - ○ Due to finite cache size
  - ○ A replaced block is later accessed again
- ❑ Conflict misses (aka collision misses)
  - ○ Conflicts that happen even when we have space left
  - ○ Due to competition for entries in a set
  - ○ Would not occur in a fully associative cache of the same total size

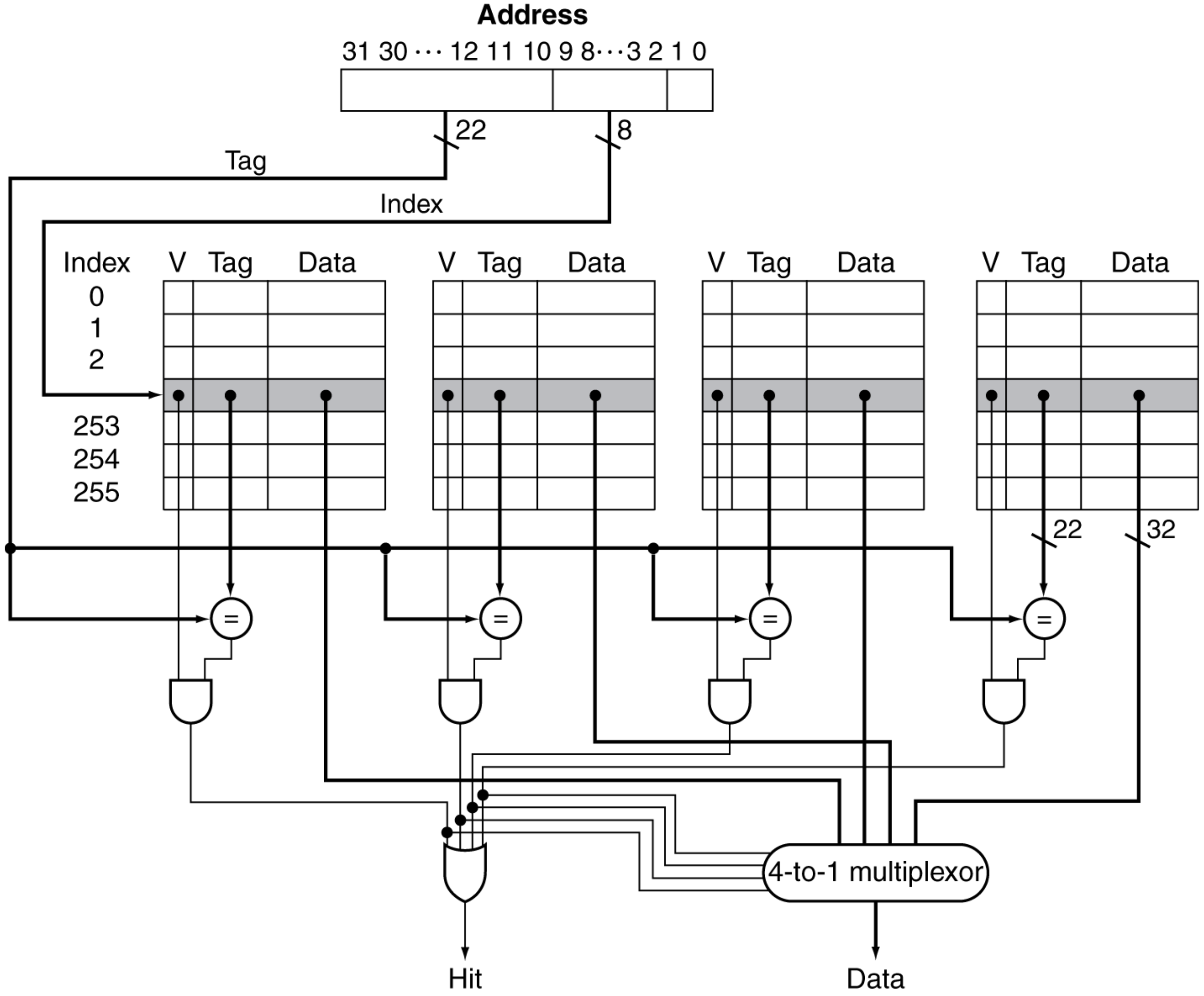
Empty space can always be used in a fully associative cache
(e.g., 8 KiB data, 32 KiB cache, but still misses? Those are conflict misses)

Balanced solution:
N-way set-associative cache

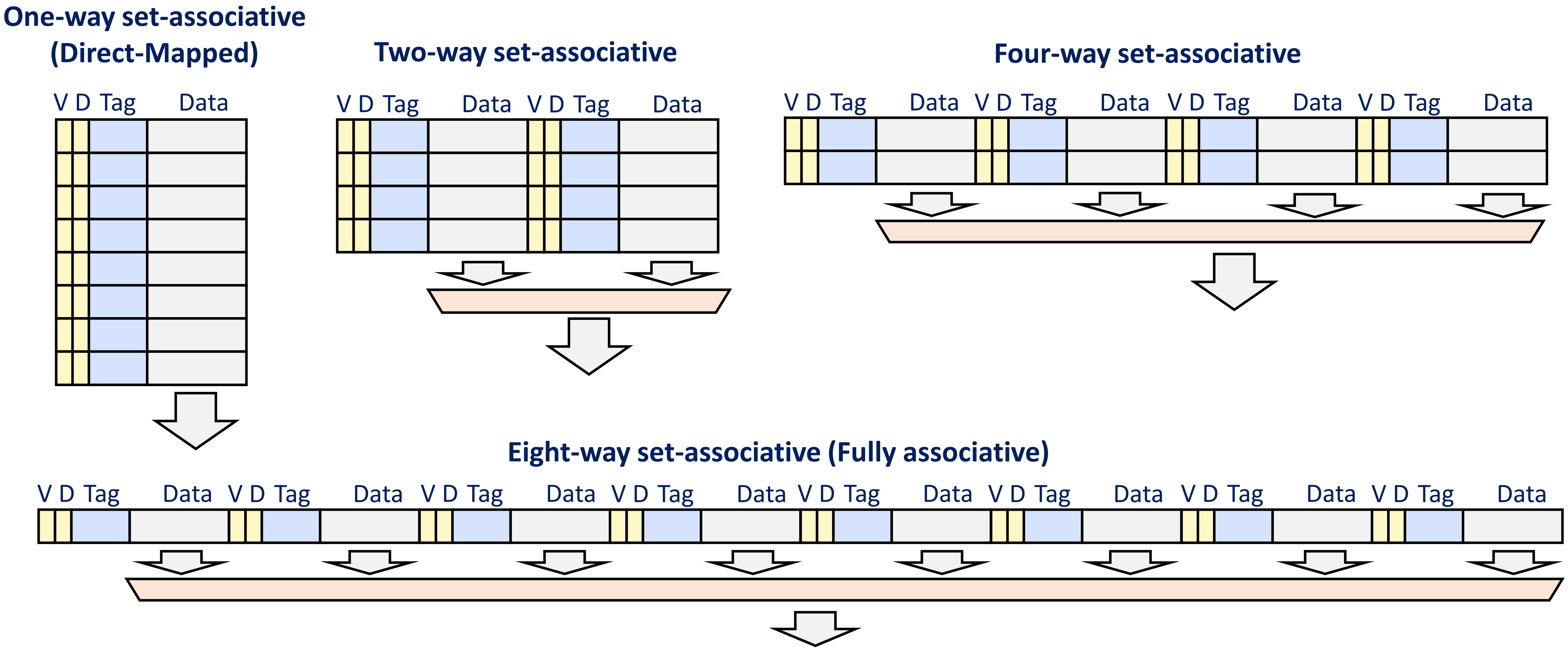
- ❑ Use multiple direct-mapped caches in parallel to reduce conflict misses
- ❑ Nomenclature:
  - ○ # Rows = # Sets
  - ○ # Columns = # Ways
  - ○ Set size = #ways = “set associativity” (e.g., 4-way -> 4 lines/set)
- ❑ Each address maps to only one set, but can be in any way within the set
- ❑ Tags from all ways are checked in parallel



Set-associative cache organization



Spectrum of associativity (For eight total blocks)


One-way set-associative
(Direct-Mapped)
V D Tag Data
Two-way set-associative
V D Tag Data V D Tag Data
Four-way set-associative
V D Tag Data V D Tag Data V D Tag Data V D Tag Data
Eight-way set-associative (Fully associative)
V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data V D Tag Data

Each “Data” is a cache line (~64 bytes), needs another mux layer to get actual word

Associativity example

- ❑ Compare caches with four elements
  - ○ Block access sequence: 0, 8, 0, 6, 8

- ❑ Direct mapped (Cache index = address mod 4)

Time ↓

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 (Cache index = address mod 2)

Time

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 (No more cache index!)

Time

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]	

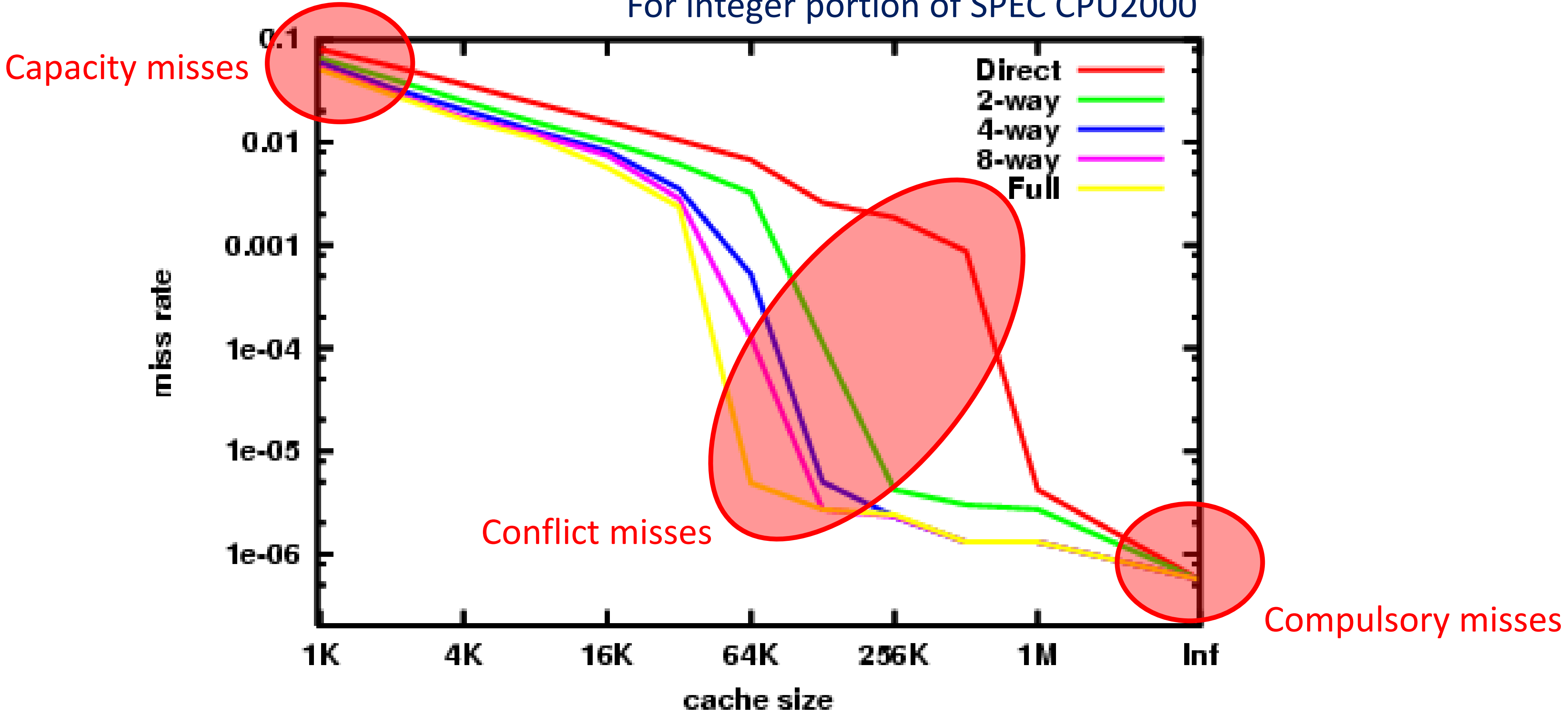
How Much Associativity?

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

How much associativity, how much size?

- ❑ Highly application-dependent!

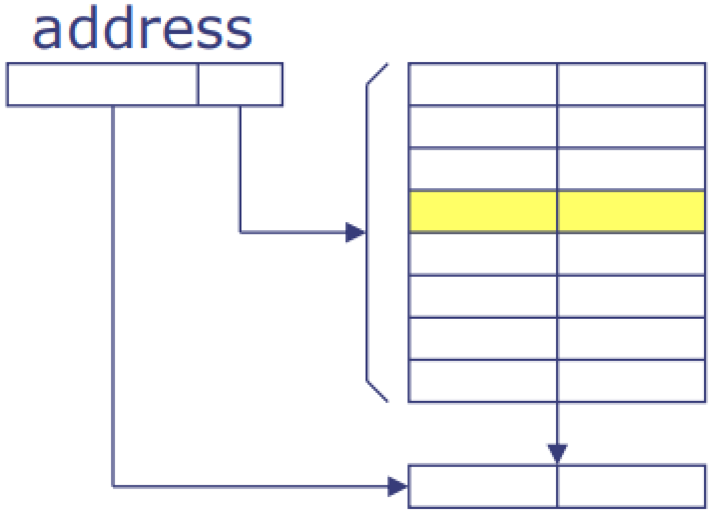
For integer portion of SPEC CPU2000



Piscione Pietro and Villardita Alessio, “Coherence and consistency models in multiprocessor architecture,” University of Pisa Computer Architecture, 2015

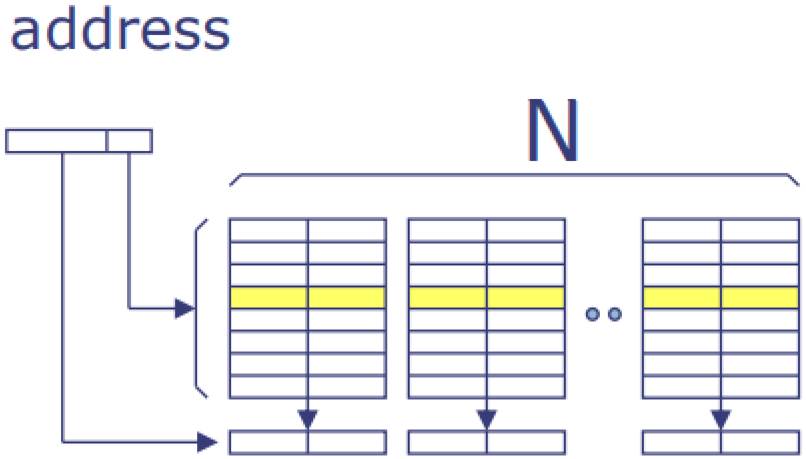
Associativity implies choices

Direct-mapped



Only one place an address can go
In case of conflict miss, old data is simply evicted

N-way set-associative



Multiple places an address can go
In case of conflict miss, which way should we evict?

What is our "***replacement policy***"?

Replacement policies

- ❑ Optimal policy (Oracle policy):
  - ○ Evict the line accessed furthest in the future
  - ○ Impossible: Requires knowledge of the future!
- ❑ Idea: Predict the future from looking at the past
  - ○ If a line has not been used recently, it's often less likely to be accessed in the near future (temporal locality argument)
- ❑ ***Least Recently Used (LRU):*** Replace the line that was accessed furthest in the past
  - ○ Works well in practice
  - ○ Needs to keep track of ordering, and discover oldest line quickly

Pure LRU requires complex logic: Typically implements cheap approximations of LRU

Other replacement policies

- ❑ LRU becomes very bad if working set becomes larger than cache size
    - o “for (i = 0 to 1025) A[i];”, if cache is 1024 elements large, every access is miss
- ❑ Some alternatives exist
    - o Effective in limited situations, but typically not as good as LRU on average
    - o Most recently used (MRU), First-In-First-Out (FIFO), random, etc …
    - o Sometimes used together with LRU

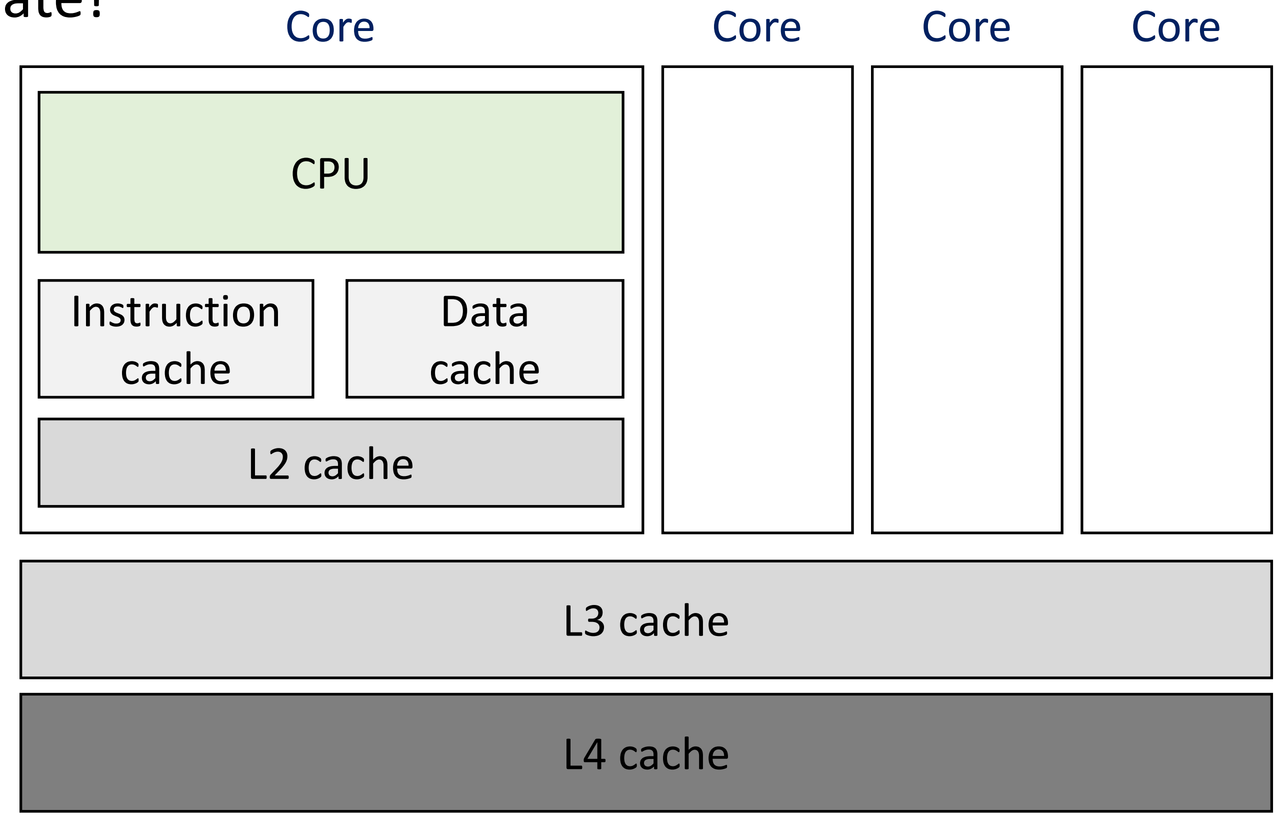
Performance improvements with caches

- ❑ Given CPU of CPI = 1, clock rate = 4GHz
  - ○ Main memory access time = 100ns
  - ○ Miss penalty = 100ns/0.25ns = 400 cycles
  - ○ CPI without cache = 400
- ❑ Given first-level cache with no latency, miss rate of 2%
  - ○ Effective CPI = 1 + 0.02 × 400 = 9
- ❑ Adding another cache (L2) with 5ns access time, miss rate of 0.5%
  - ○ Miss penalty = 5ns/0.25ns = 20 cycles
  - ○ New CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4

	Base	L1	L2
CPI Improvements	400	9	3.4
IPC improvements	0.0025	0.11	0.29
Normalized performance	1	44	118

Real-world: Intel Haswell i7

- Four layers of caches (two per-core layers, two shared layers)
  - Larger caches have higher latency
  - Want to achieve both speed and hit rate!
- The layers
  - L1 Instruction & L1 Data:
    32 KiB, 8-way set associative
  - L2: 256 KiB, 8-way set associative
  - L3: 6 MiB, 12-way set associative
  - L4: 128 MiB, 16-way set associative
    eDRAM!



Real-world: Intel Haswell i7

- ❑ Cache access latencies
  - o L1: 4 - 5 cycles
  - o L2: 12 cycles
  - o L3: ~30 - ~50 cycles
- ❑ For reference, Haswell as 14 pipeline stages

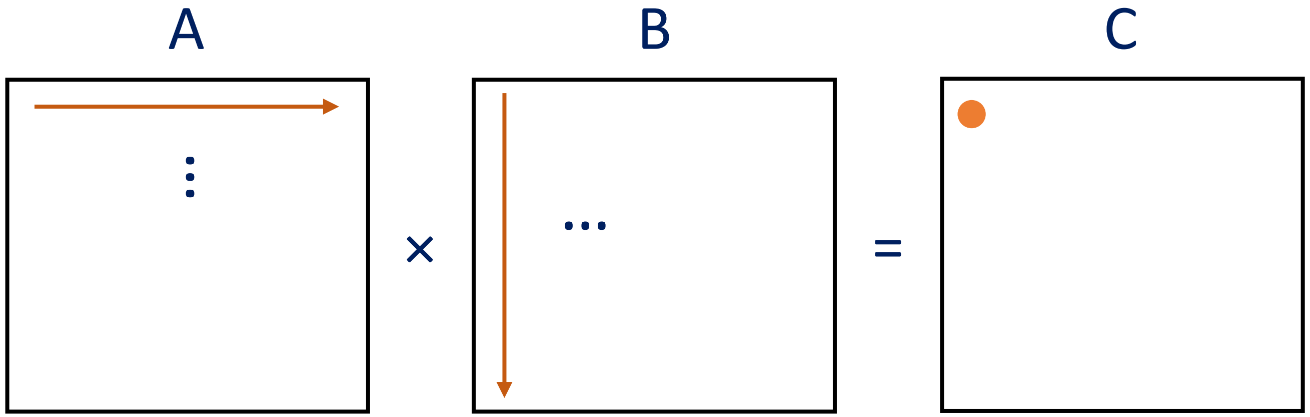
So far…

- What are caches and why we need them
- Direct-mapped cache
  - Write policies
  - Larger block size and implications
  - Conflict and other misses
- Set-associative cache
  - Replacement policies

Cache-aware software example: Matrix-matrix multiply

- Multiplying two NxN matrices (C = A × B)

```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * B[k][j]
```



2048*2048 on a i5-7400 @ 3 GHz = 63.19 seconds

is this fast?

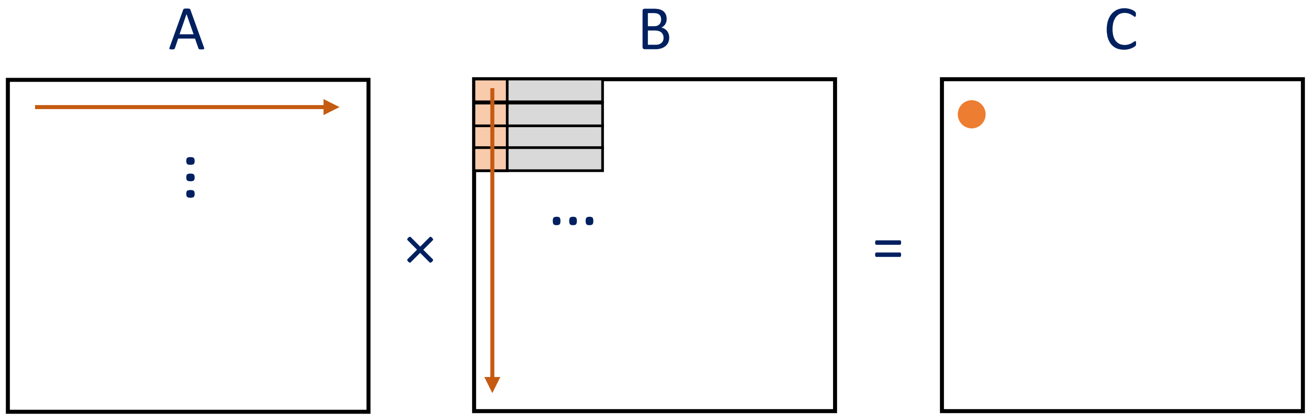
Whole calculation requires 2K * 2K * 2K = 8 Billion floating-point mult + add
At 3 GHz, ~5 seconds just for the math. Over 1000% overhead!

Assuming IPC=1, true numbers complicated due to superscalar

Overheads in matrix multiplication (1)

- Column-major access makes inefficient use of cache lines
  - A 64 Byte block is read for each element loaded from B
  - 64 bytes read from memory for each 4 useful bytes
- Shouldn't caching fix this? Unused bits should be useful soon!
  - 64 bytes x 2048 = 128 KB … Already overflows L1 cache (~32 KB)

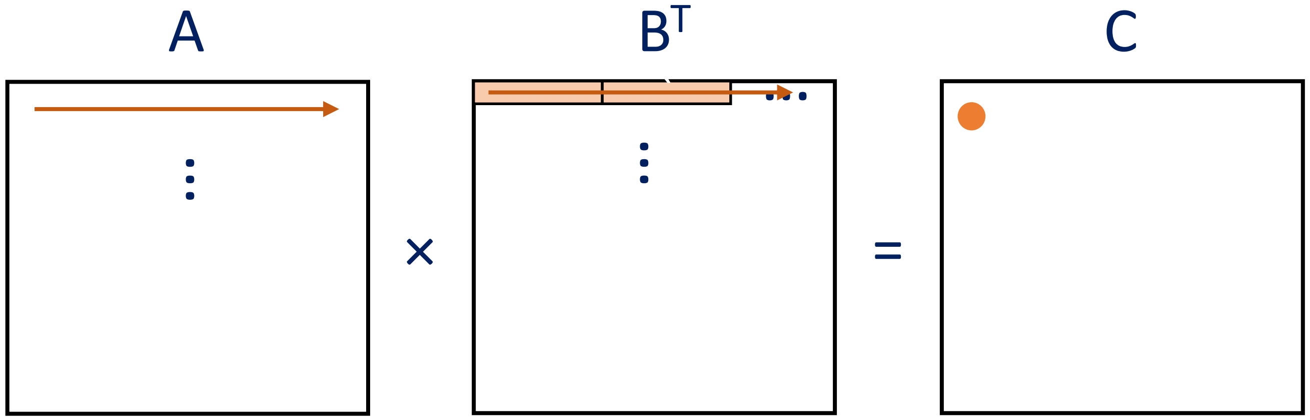
```
for (i = 0 to N)
    for (j = 0 to N)
        for (k = 0 to N)
            C[i][j] += A[i][k] * B[k][j]
```



Overheads in matrix multiplication (1)

- One solution: Transpose B to match cache line orientation
  - Does transpose add overhead? Not very much as it only scans B once
- Drastic improvements!
  - Before: 63.19s
  - After: 10.39s … 6x improvement!
  - But still not quite ~5s

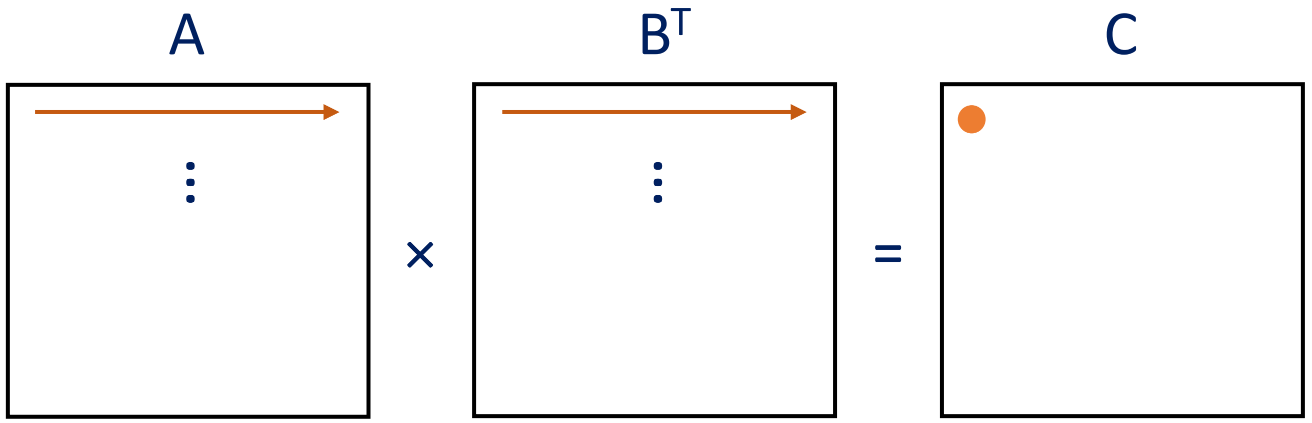
```
for (i = 0 to N)
   for (j = 0 to N)
      for (k = 0 to N)
         C[i][j] += A[i][k] * Bt[j][k]
```



Overheads in matrix multiplication (2)

- Both A and B read N times
  - A re-uses each row before moving on to next
  - B scans the whole matrix for each row of A
  - One row: 2048 * 4 bytes = 8192 bytes fits in L1 cache (32 KB)
  - One matrix: 2048 * 2048 * 4 bytes = 16 MB exceeds in L3 cache (6 MB shared across 4 cores)
  - No caching effect for B!

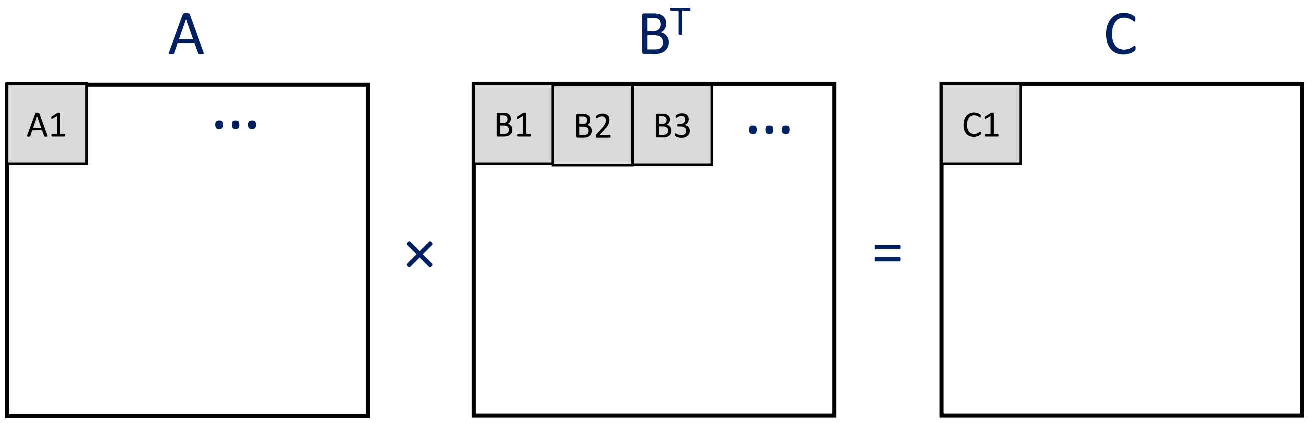
```
for (i = 0 to N)
    for (j = 0 to N)
        for (k = 0 to N)
            C[i][j] += A[i][k] * Bt[j][k]
```



Overheads in matrix multiplication (2)

- One solution: “Blocked” access
  - Assuming BxB fits in cache,
  - B is read only N/B times from memory
- Performance improvement!
  - No optimizations: 63.19s
  - After transpose: 10.39s
  - After transpose + blocking: 7.35

```
for (i = 0 to N/B)
  for (j = 0 to N/B)
    for (k = 0 to N/B)
      for (ii = 0 to B)
        for (jj = 0 to B)
          for (kk = 0 to B)
            C[i*B+ii][j*B+jj] += A[i*B+ii][k*B+kk] * Bt[j*B+jj][k*B+kk]
```



C1 sub-matrix = A1×B1 + A1×B2 + A1×B3 … A2×B1 …

Aside: Cache oblivious algorithms

- ❑ For sub-block size B × B -> N * N * (N/B) reads. What B do we use?
  - ○ Optimized for L1? (32 KiB for me, who knows for who else?)
  - ○ If B*B exceeds cache, sharp drop in performance
  - ○ If B*B is too small, gradual loss of performance
- ❑ Do we ignore the rest of the cache hierarchy?
  - ○ Say B optimized for L3,
B × B multiplication is further divided into T×T blocks for L2 cache
  - ○ T × T multiplication is further divided into U×U blocks for L1 cache
  - ○ … If we don’t, we lose performance
- ❑ Class of “cache-oblivious algorithms”

Typically recursive definition of data structures… topic for another day

Aside: Recursive Matrix Multiplication

$$\overset{C}{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}} = \overset{A}{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}} \times \overset{B}{\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiply-adds of (n/2) × (n/2) matrices
Recurse down until very small