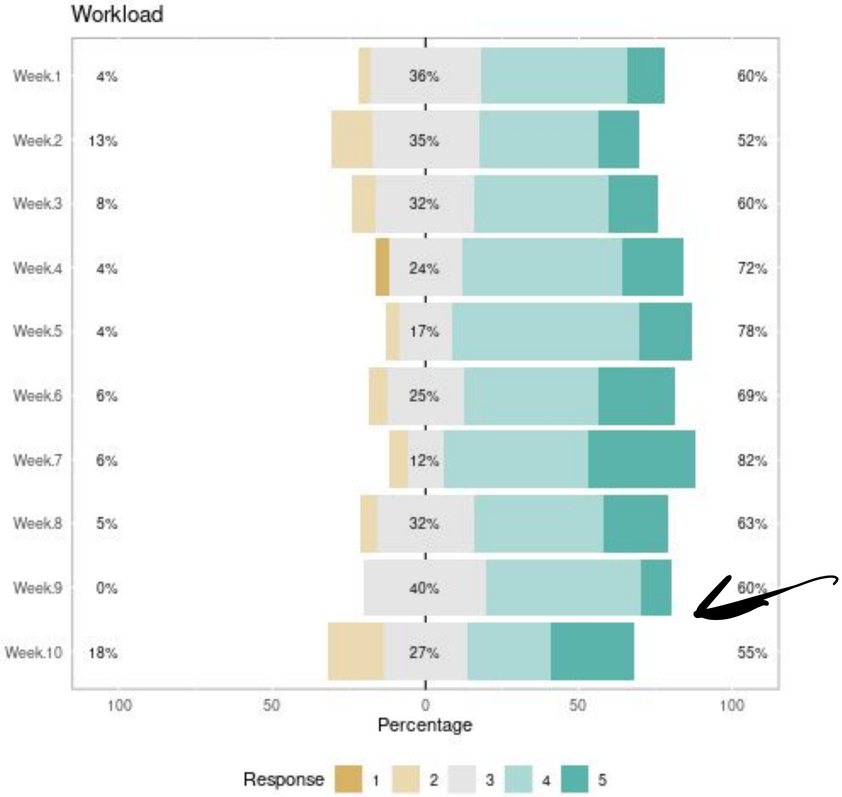
0x13 - Memory and Caches

ENGR 3410: Computer Architecture

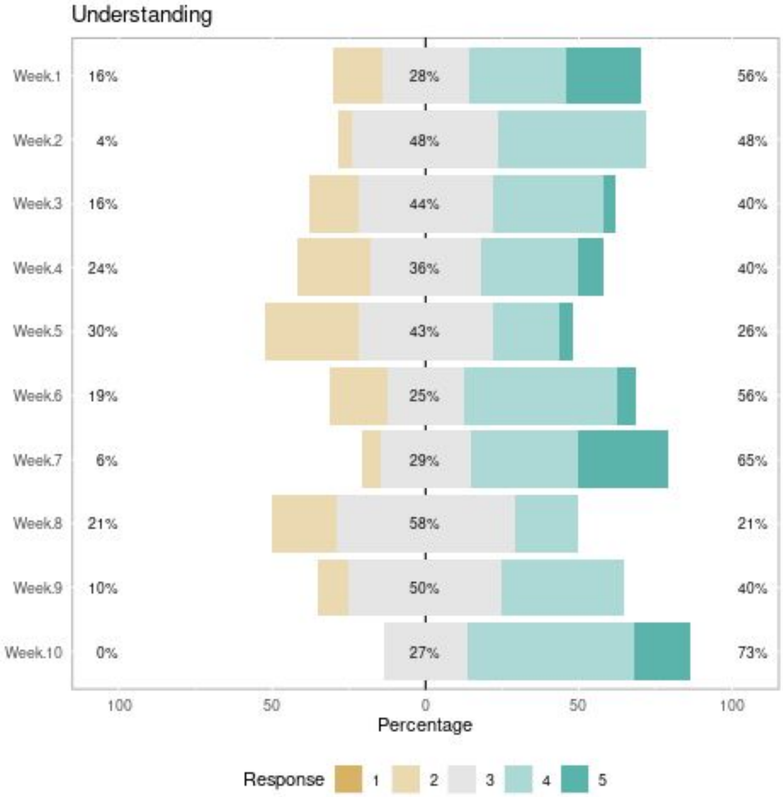
Jon Tse

Fall 2020

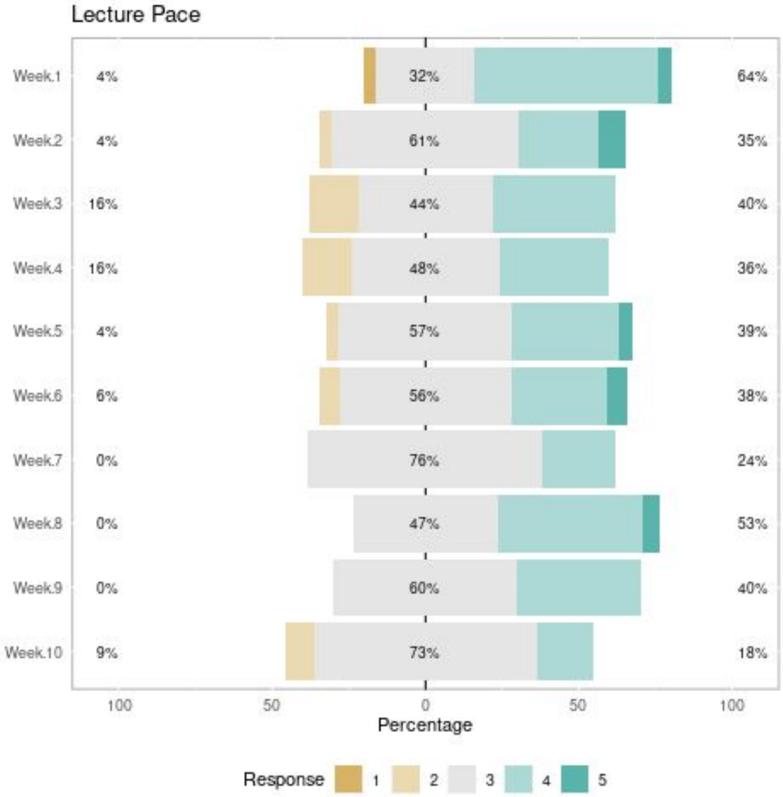
Feedback - Workload



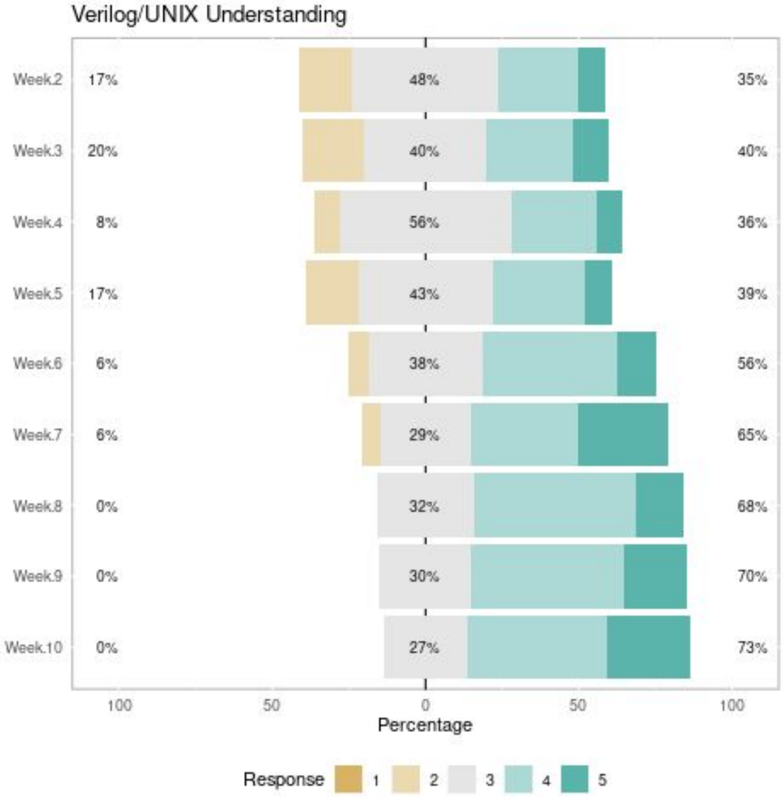
Feedback - Understanding



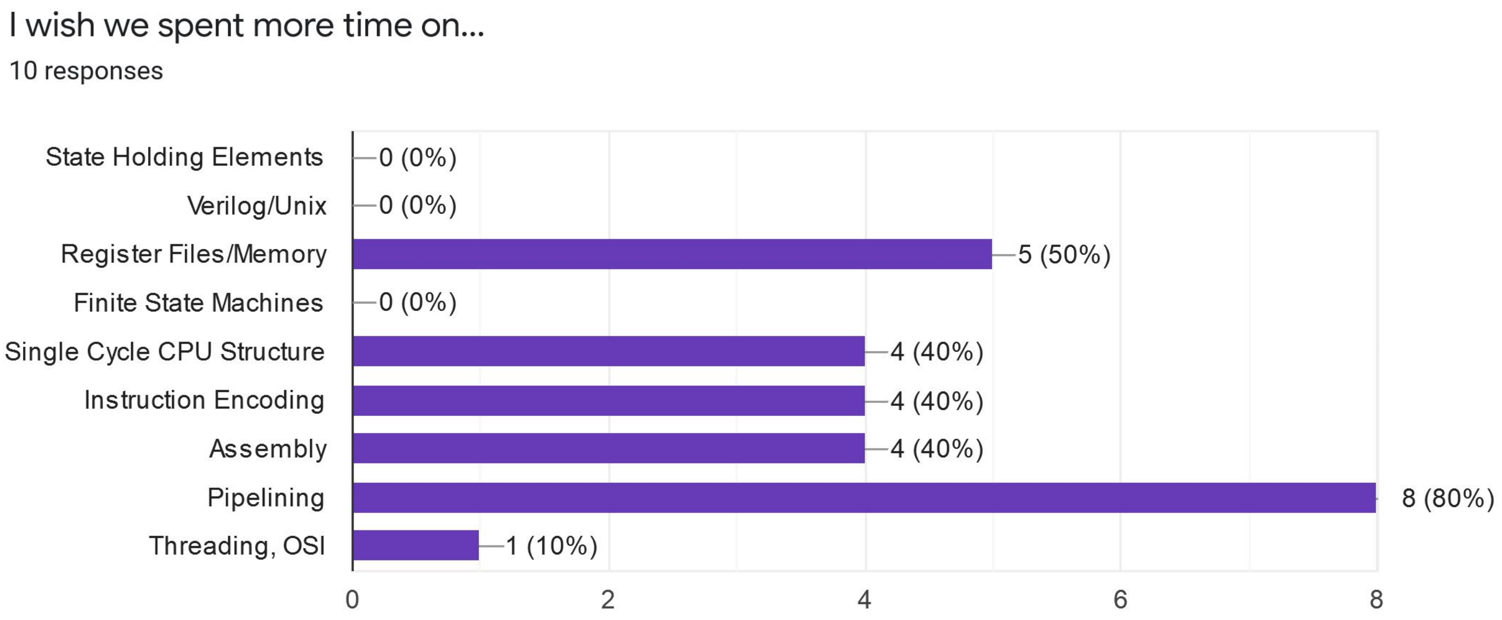
Feedback - Pace



Feedback - Tools



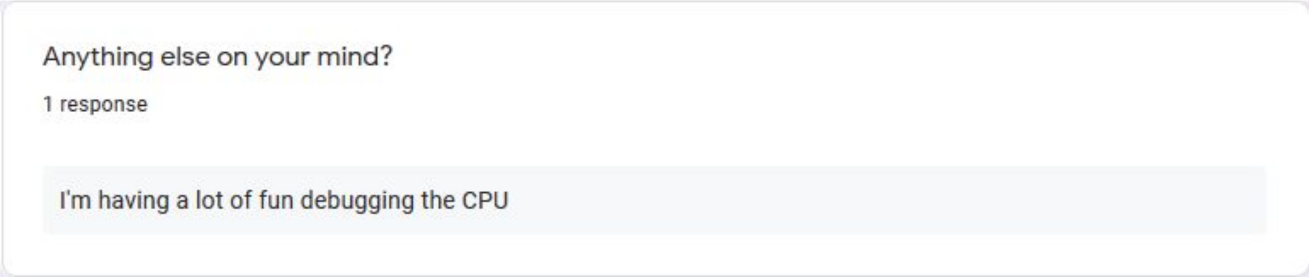
Feedback - More Time...



I wish we spent more time on...

10 responses

Feedback - Top of Mind?



Anything else on your mind?

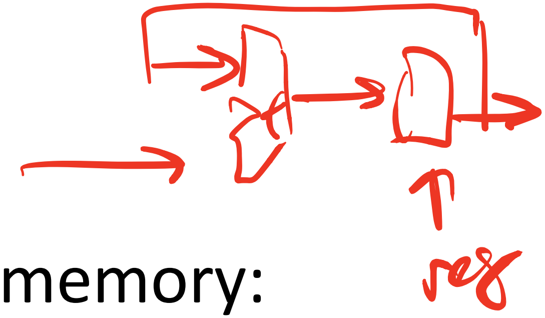
1 response

I'm having a lot of fun debugging the CPU

Final Project

- NINJAs will reduce hours, but…
- You can ask a NINJA to be your “advisor!”
- Reach out, they want to learn with you!

Review



- So far, we have three types of memory:
  - Registers/DFF (between pipeline stages)
  - Register File
  - Black box memory for loads and stores

- Today we'll investigate the internals of the memory system

Goals

- As always, computer architects care about: Performance – Area – Energy - Design Time
- For memory, you can have
  - Fast but Expensive (large area)
  - Slow but Cheap
- Our goal – design the best of both worlds

Static Random Access Memory (SRAM)

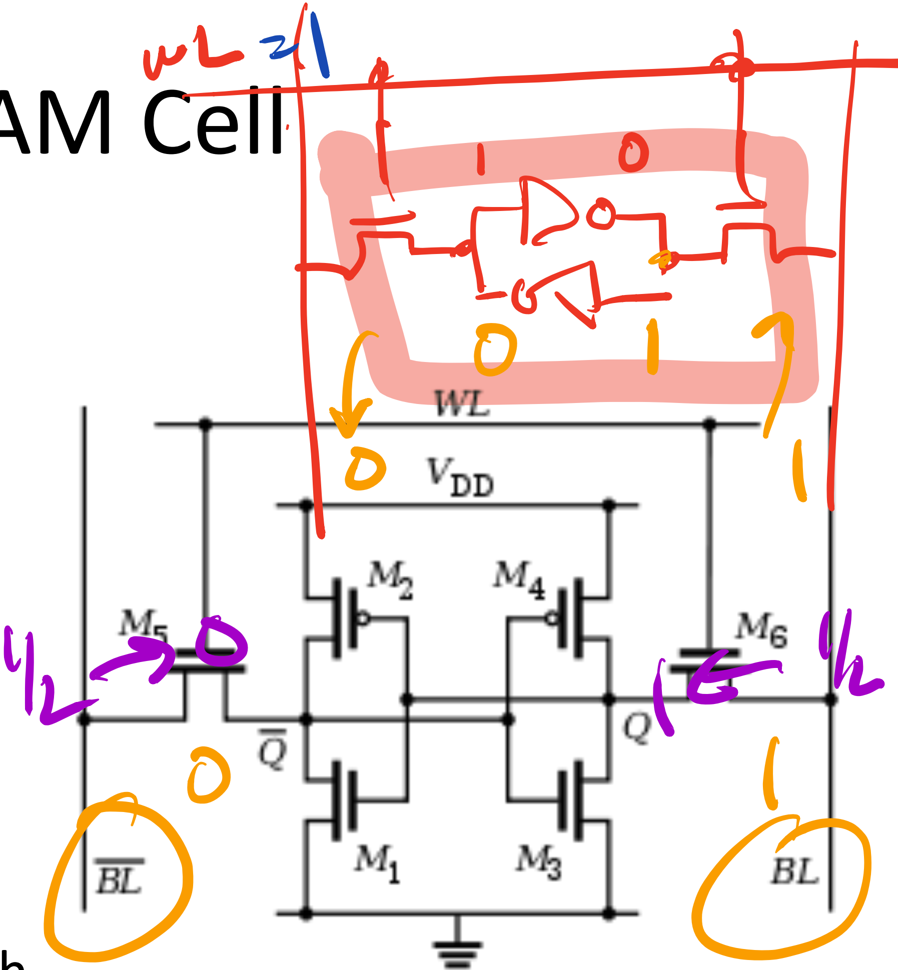
- Like a register file, but:
  - Only one Port (unified read/write)
  - Many more words (greater depth)

- Different Cell Construction
  - Smaller than a D-Flip Flop

- Different Word Select Construction
  - Individual cells are weak, so we add circuitry at the perimeter to amplify during a read

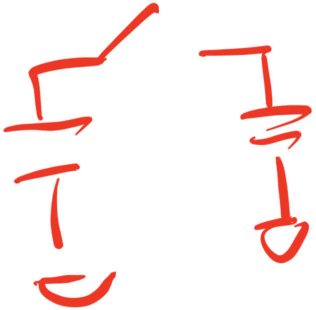
SRAM Cell

- **M1-M3** store bit weakly
- **W**ord **L**ine accesses cell
  - Open M5, M6
- **Bit L**ines
  - Read bit (weak signal)
  - Write bit (strong signal)
- Cross-coupled like a latch



Dynamic

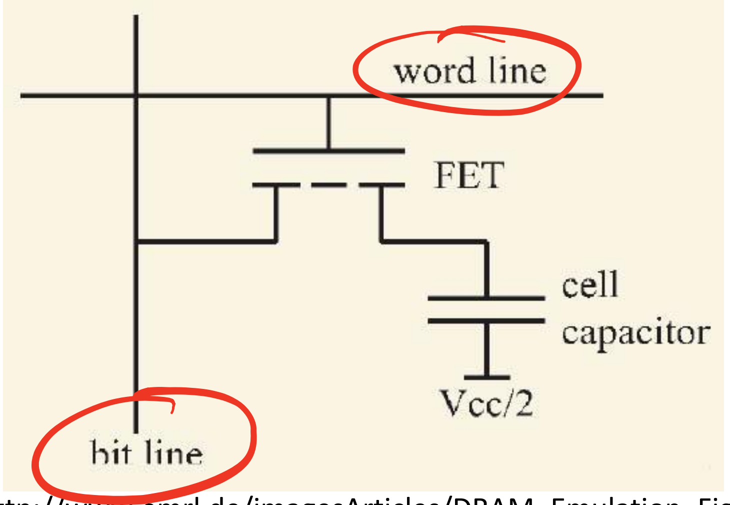
DRAM Cell



- Capacitor stores bit
  - For a while
  - Must be "refreshed"

- FET controls access

- Smaller than SRAM
- Slower than SRAM

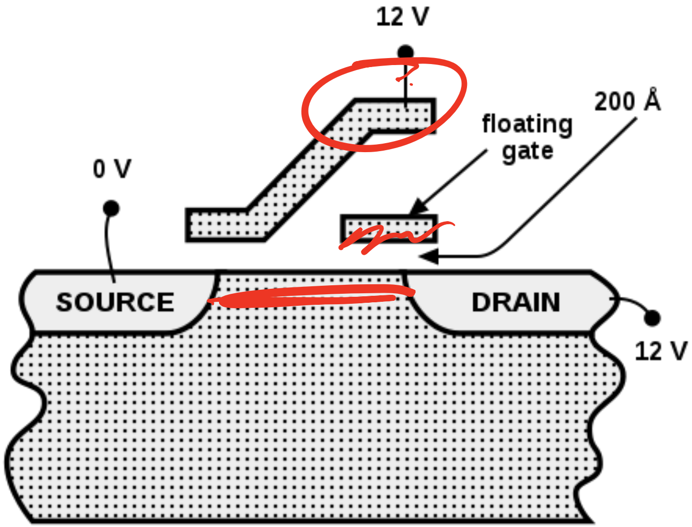


http://www.emrl.de/imagesArticles/DRAM_Emulation_Fig2.jpg

Flash Cell

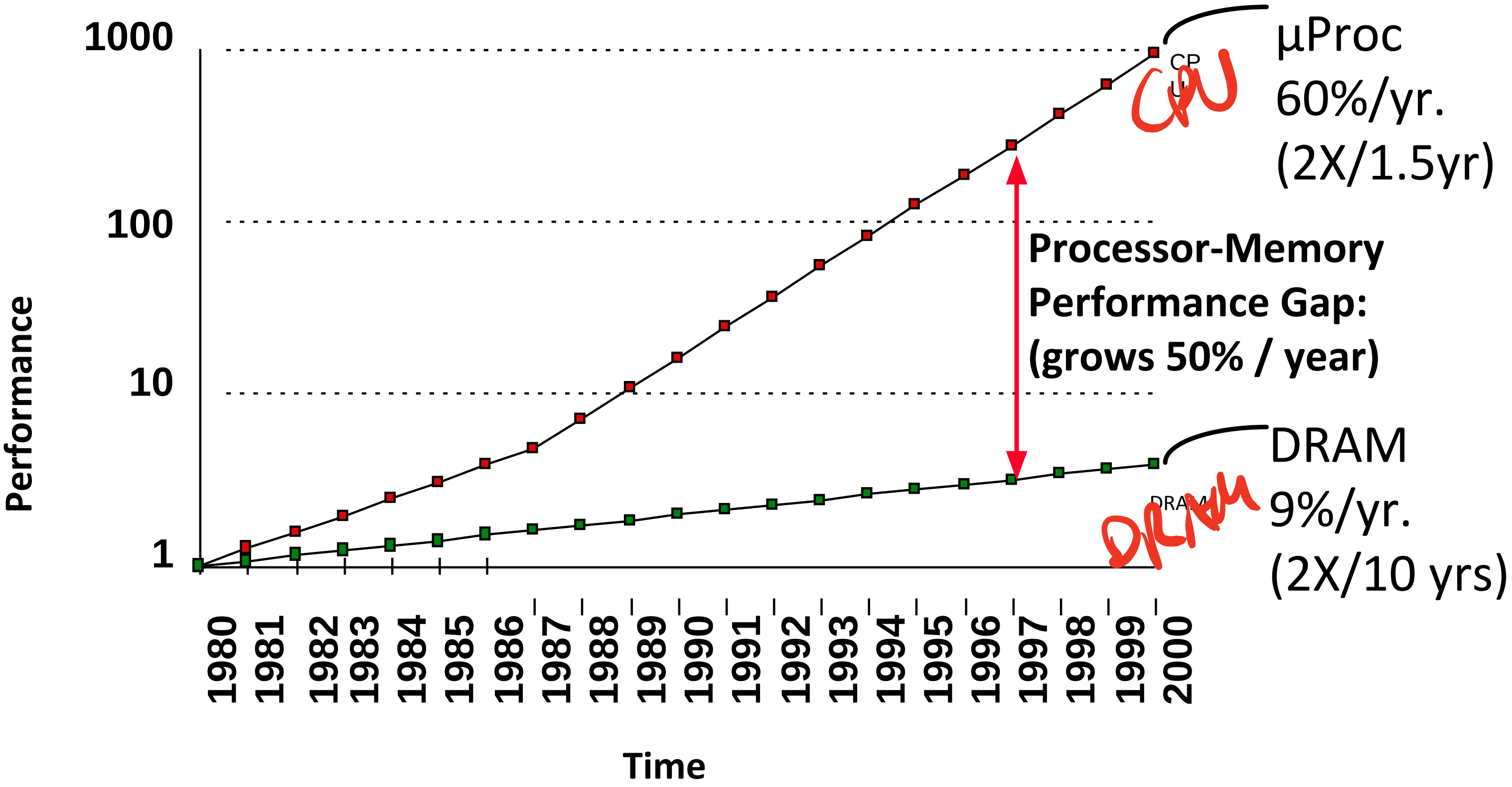
- Transistor with floating gate stores bit
  - Nonvolatile!
  - Limited lifespan
- Possible to have multiple bits per cell
- Requires elevated voltage for program/erase

Programming via hot electron injection



Technology Trends

- Processor-DRAM Memory Gap (latency)



µProc
60%/yr.
(2X/1.5yr)

Processor-Memory Performance Gap: (grows 50% / year)

DRAM
9%/yr.
(2X/10 yrs)

Technology Trends – Modern Story

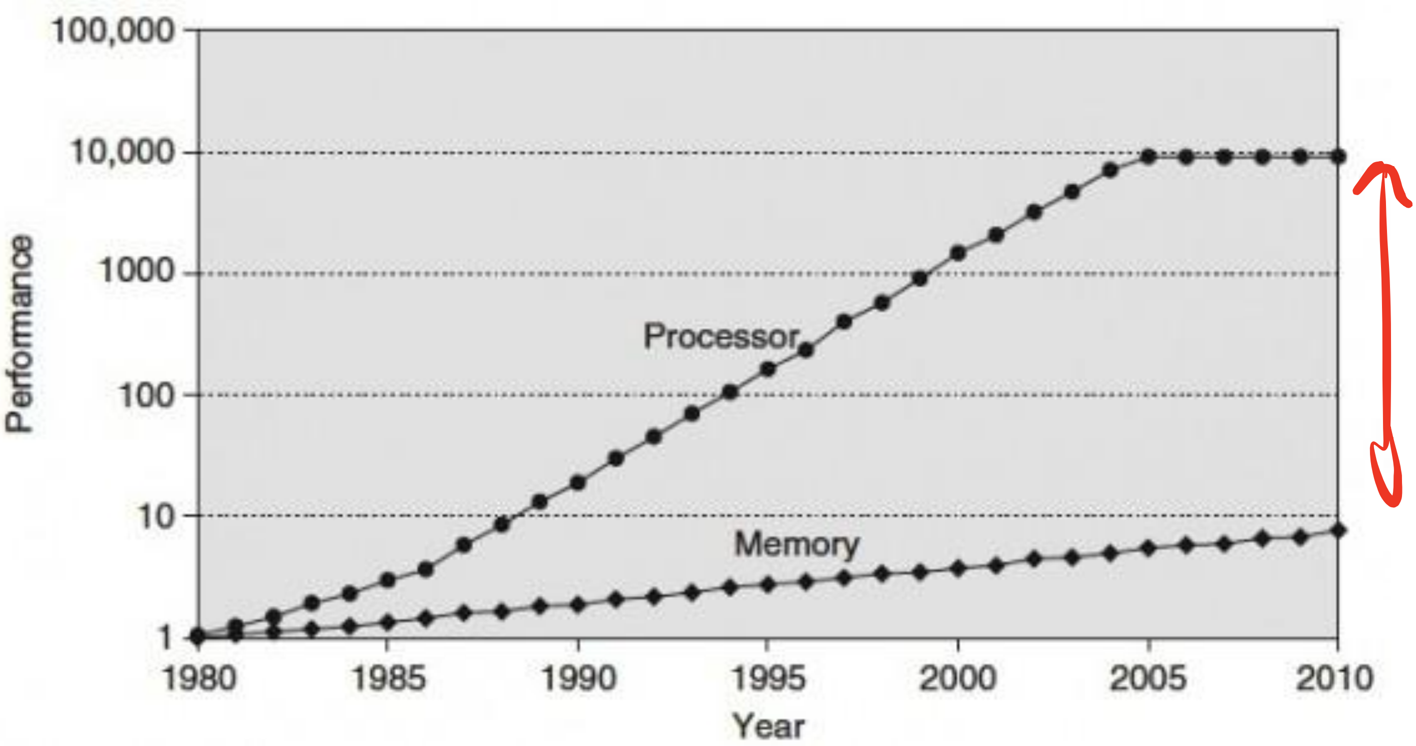


Figure from Hennessy and Patterson's Computer Architecture: A Quantitative Approach

The Problem

- The Von Neumann Bottleneck "Memory Wall"
  - Logic gets faster
  - Memory capacity gets larger
  - **Memory speed is not keeping up with logic**

- How do we cope?

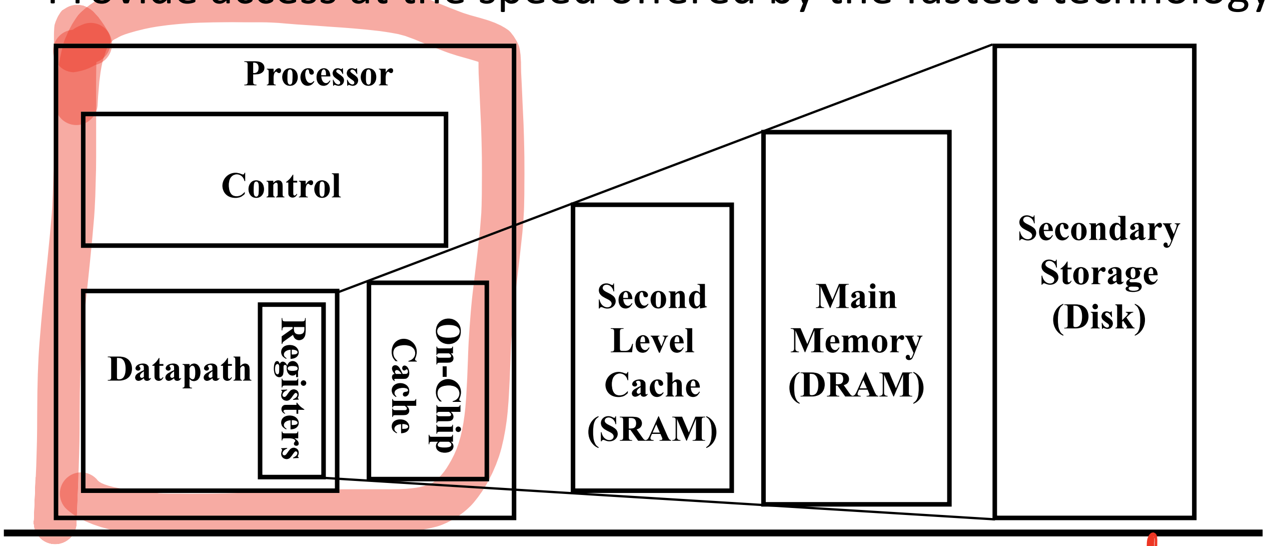
Fast, Big, Cheap: Pick 3

- Design Philosophy
  - Use a hybrid approach that uses aspects of both
  - Lots of slow + cheap
  - Small amount of fast + expensive

- “Cache”
  - Make the common case fast
  - Keep frequently used things in a small amount of fast/expensive memory

The Solution

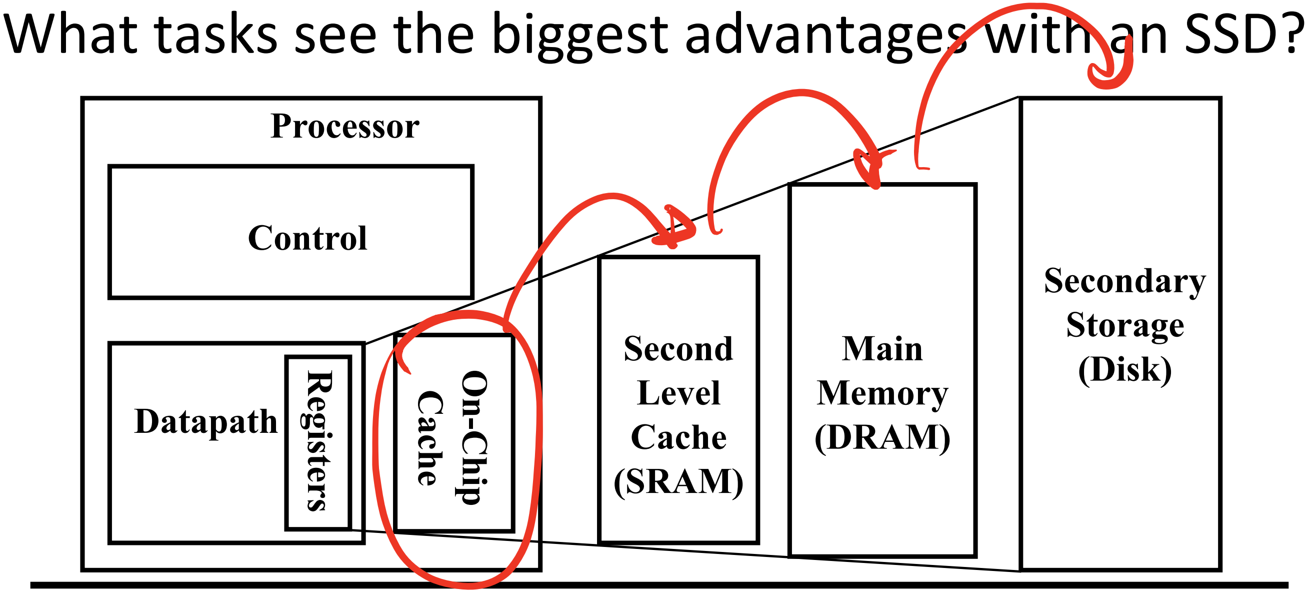
- By taking advantage of the principle of locality:
  - Provide as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



	DFF	SRAM	DRAM	Magnetic
Type	Regfile	Cache	Main Memory	Disk
Speed	< 1ns	< 10ns	~60ns	10 ms
Size	100B	KB-MB	GB	TB

SSD

- 100-1000x faster access time than HDD
- 3-8x faster read/write **rate**
- What tasks see the biggest advantages with an SSD?



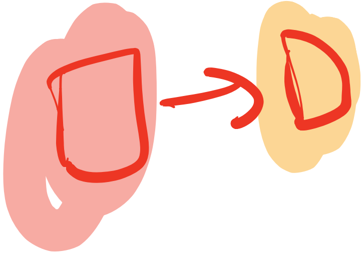
Type	Regfile	Cache	Main Memory	SSD
Speed	< 1ns	< 10ns	~60ns	0.1 ms
Size	100B	KB-MB	GB	TB

Cache Terminology

- **Hit**: Data appears in that level
  - **Hit rate** – percent of accesses hitting in that level
  - **Hit time** – Time to access this level
    - Hit time = Access time + Time to determine hit/miss
- **Miss:** Data does not appear in that level and must be fetched from lower level
  - **Miss rate** – percent of misses at that level = (1 – hit rate)
  - **Miss penalty** – Overhead in getting data from a lower level
    - Miss penalty = Lower level access time + Replacement time + Time to deliver to processor

- Miss penalty is usually MUCH larger than the hit time

Cache Access Time



- Average access time
  - Access time = (hit time) + (miss penalty)x(miss rate)
  - Want low miss rate & low hit time, since miss penalty is large
- Average Memory Access Time (AMAT)
  - Apply average access time to entire hierarchy.

Split Caches

- Often separate Instruction and Data Caches
  - Harvard Architecture: Split I & D
  - von Neumann Architecture: Unified

- Higher bandwidth
- Optimize to usage
- Slightly higher miss rate: each cache is smaller

Handling A Cache Miss

reg

L1

L2

L3

SRAM

DRAM

- Data Miss
  1. Stall the pipeline (freeze following instructions)
  2. Instruct memory to perform a read and wait
  3. Return the result from memory and allow the pipeline to continue
- Instruction Miss
  1. Send the original PC to the memory
  2. Instruct memory to perform a read and wait (no write enables)
  3. Write the result to the appropriate cache line
  4. Restart the instruction

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	
L2	10 cycles	90%	
Main Memory	50 cycles	99%	
Disk	50,000 cycles	100%	

- Note: Numbers are **local** hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	
L2	10 cycles	90%	
Main Memory	50 cycles	99%	
Disk	50,000 cycles	100%	50,000

- Note: Numbers are **local** hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	
L2	10 cycles	90%	
Main Memory	50 cycles	99%	50 + .01 * 50000 = 550
Disk	50,000 cycles	100%	50,000

- Note: Numbers are **local** hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	
L2	10 cycles	90%	10 + .1 * 550 = 65
Main Memory	50 cycles	99%	50 + .01 * 50000 = 550
Disk	50,000 cycles	100%	50,000

- Note: Numbers are **local** hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	1 + .05 * 65 = 4.25
L2	10 cycles	90%	10 + .1 * 550 = 65
Main Memory	50 cycles	99%	50 + .01 * 50000 = 550
Disk	50,000 cycles	100%	50,000

- Note: Numbers are **local** hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)

How do we get those Hit Rates?

- There's no such thing as a free lunch
  - If access was totally random, we'd be out of luck
  - But we have knowledge a priori about programs

- Locality
  - Temporal Locality – If an item has been accessed recently, it will tend to be accessed again soon
  - Spatial Locality – If an item has been accessed recently, nearby items will tend to be accessed soon

Example

```
char *index = string;
while (*index != 0) { /* C strings end in 0 */
   if (*index >= ‘a’ && *index <= ‘z’)
      *index = *index +(‘A’ - ‘a’);
   index++;
}
```

- What does this code do?

- What type(s) of locality does it have?

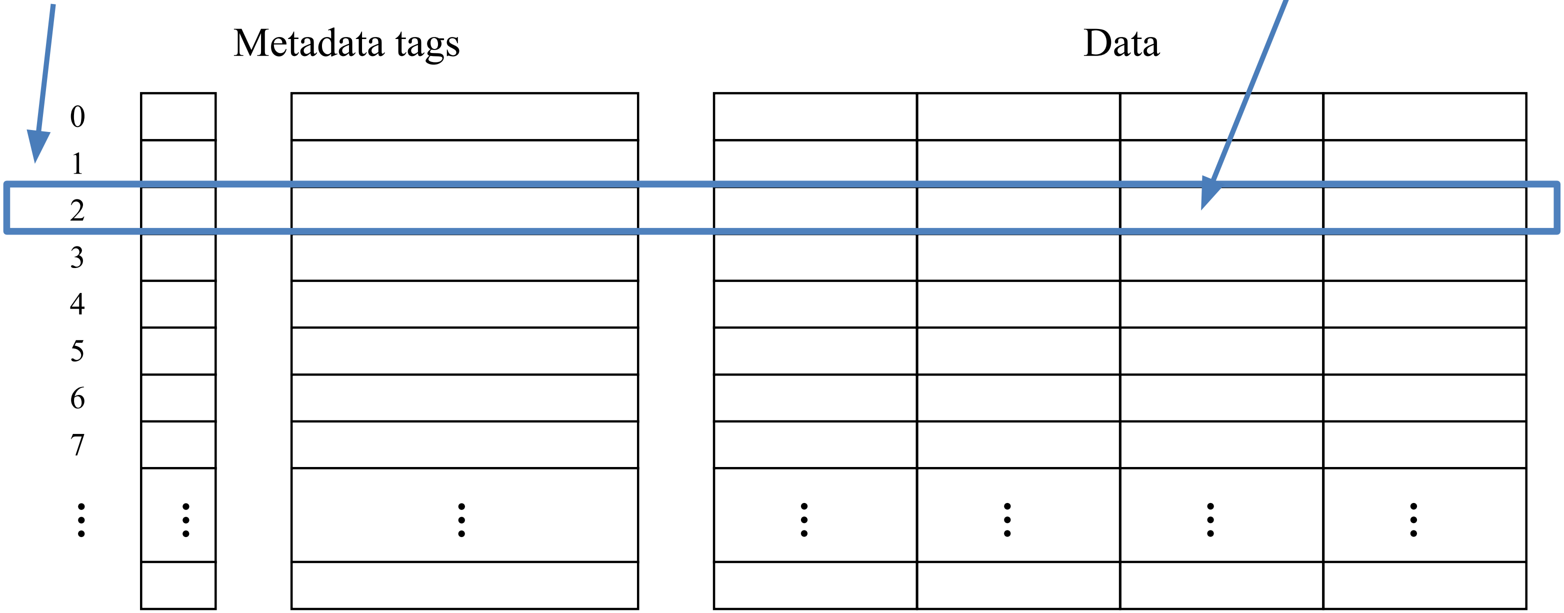
Exploiting Locality

- Temporal locality
  - Keep more recently accessed items closer to the processor
  - When we must evict items to make room for new ones, attempt to keep more recently accessed items

- Spatial locality
  - When there is a cache miss, move blocks of multiple contiguous words into cache

Whose Line is it, Anyway?

How do we map System Memory into Cache?



Cache line

Each block may contain multiple memory words (spatial locality)

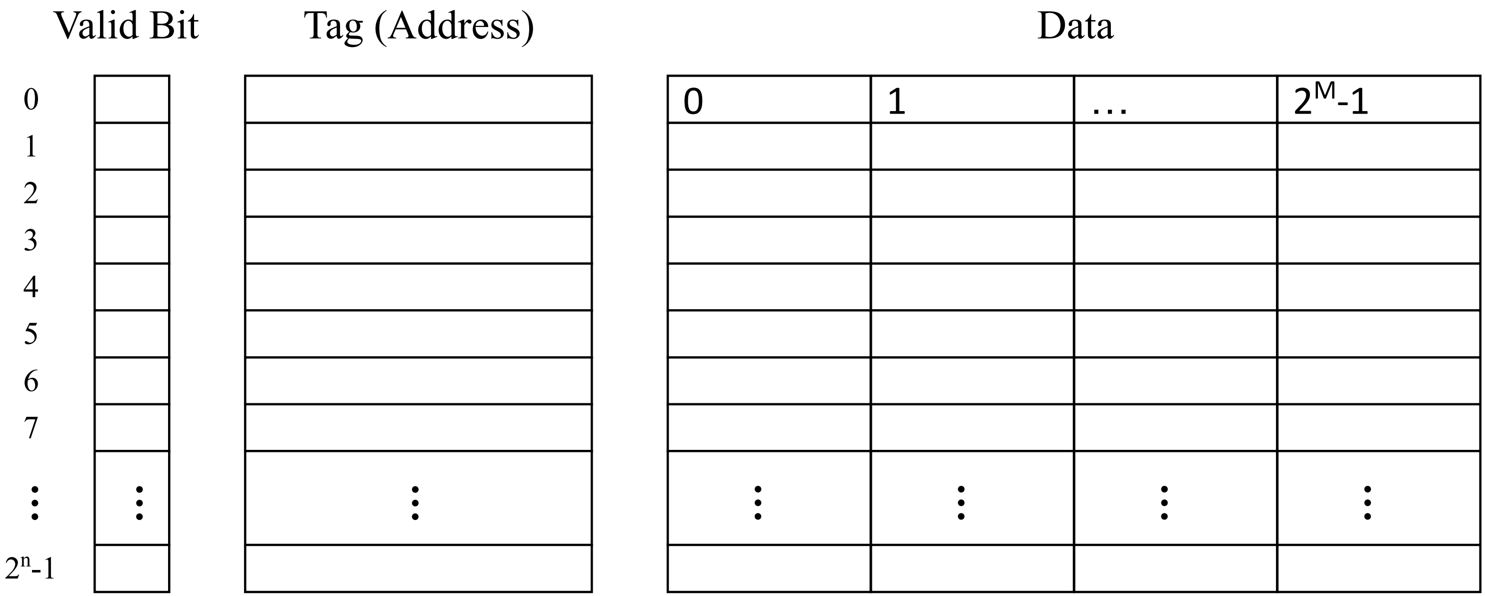
Design questions

Assume a fixed amount of storage for your cache design (data + metadata). Propose a cache organization, answering the following questions:

- When moving data from memory into cache, where (i.e. which cache line) do we put it?
- How do we detect whether data for a given memory address is in the cache or not?
- What happens when we run out of cache space?
- How many memory words should go into each cache block?

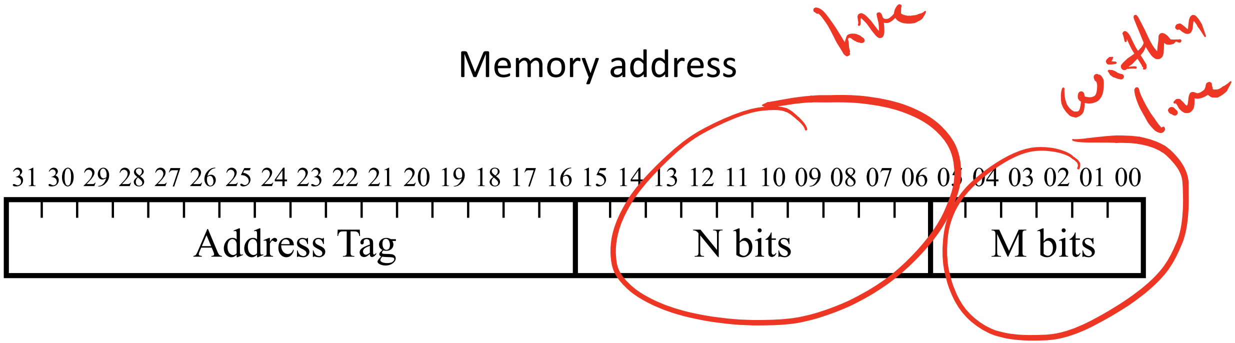
Example Cache Design

- 2^N **Blocks** of Data 2^M bytes wide
- **Tag** indicates what is stored in each block



Idea One: Direct Mapping

- Each address maps to a single cache line
- Lower M bits are address within a block
- Next N bits are the cache line address
- Remainder are the address Tag



Memory address

line

within line

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Address Tag

N bits

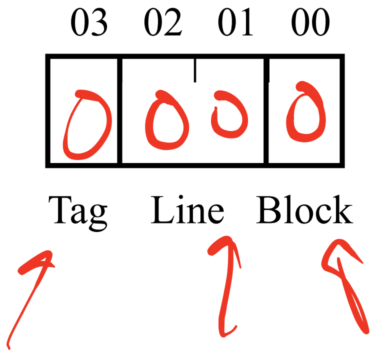
M bits

Direct Mapped Cache

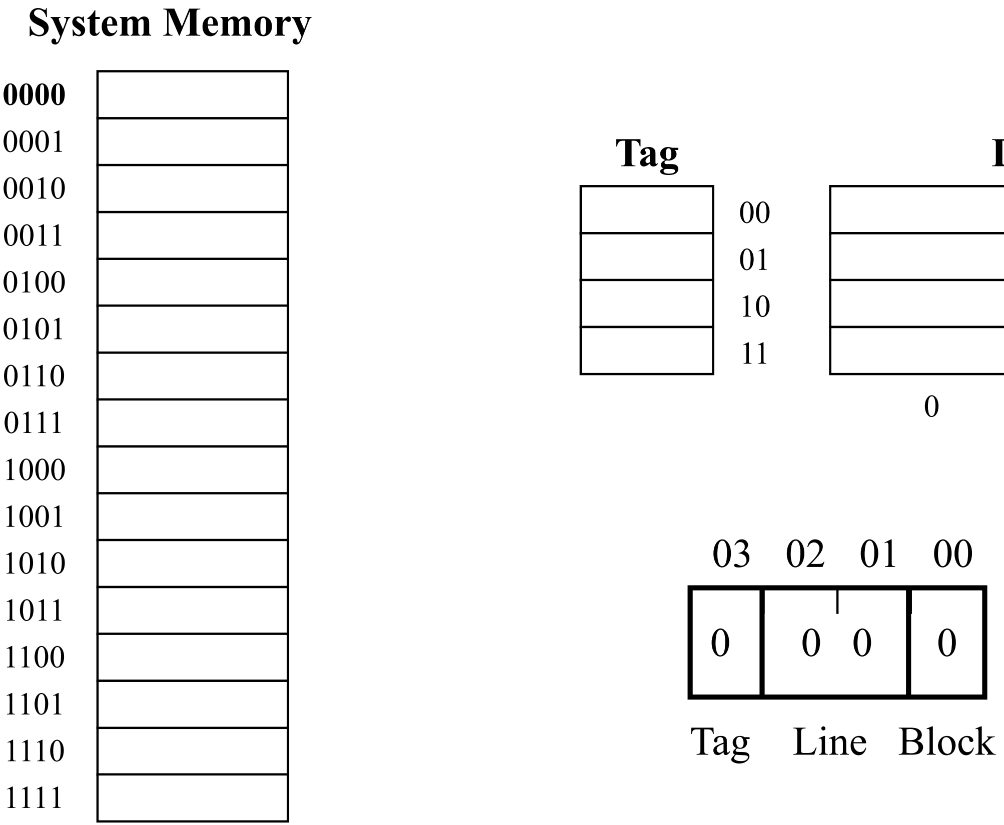
System Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

Tag		Data	
	00		
	01		
	10		
	11		
		0	1



Direct Mapped Cache



System Memory

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Tag **Data**

00
01
10
11

0 1

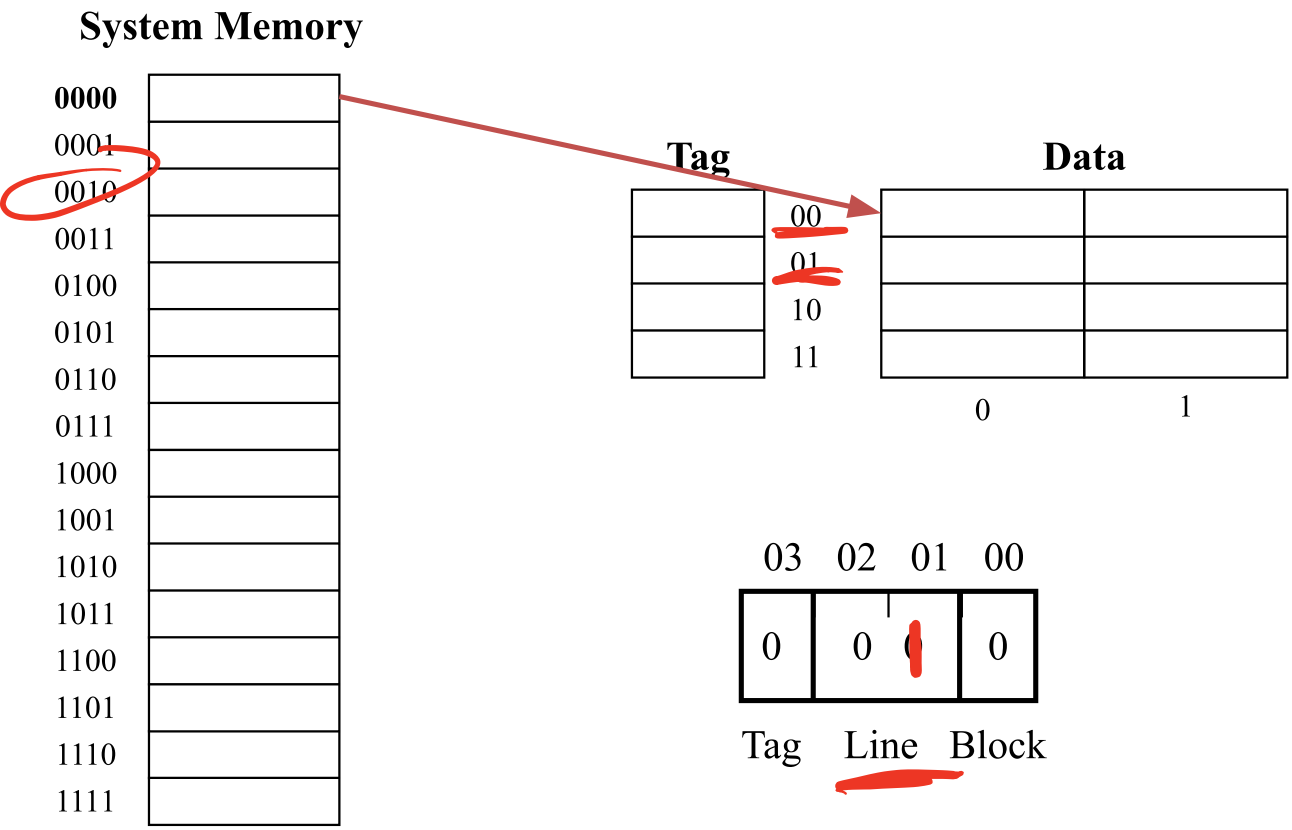
03 02 01 00

0 0 0 0

Tag Line Block

Memory address split into Tag, Line, and Block (sizes N,M depend on sizes chosen for cache)

Direct Mapped Cache



System Memory

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Tag

Data

00
01
10
11

0 1

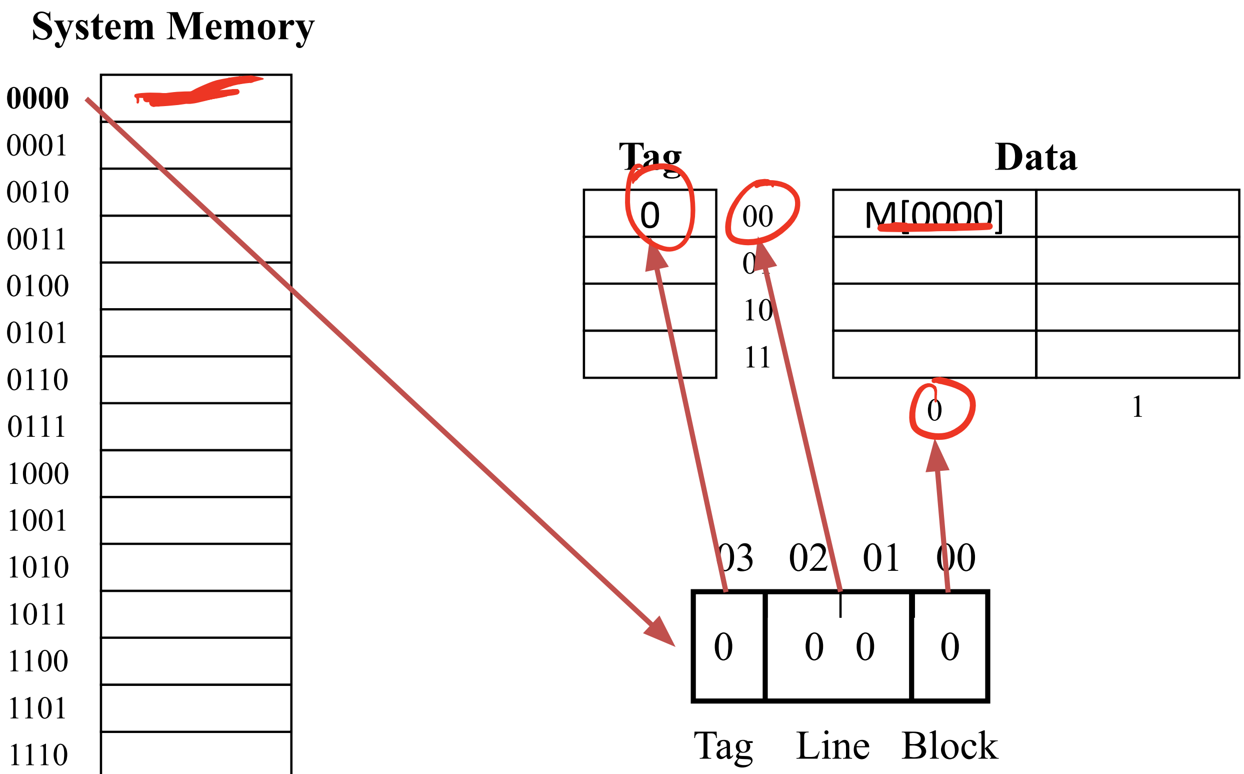
03 02 01 00

0 0 ~~0~~ 1 0

Tag Line Block

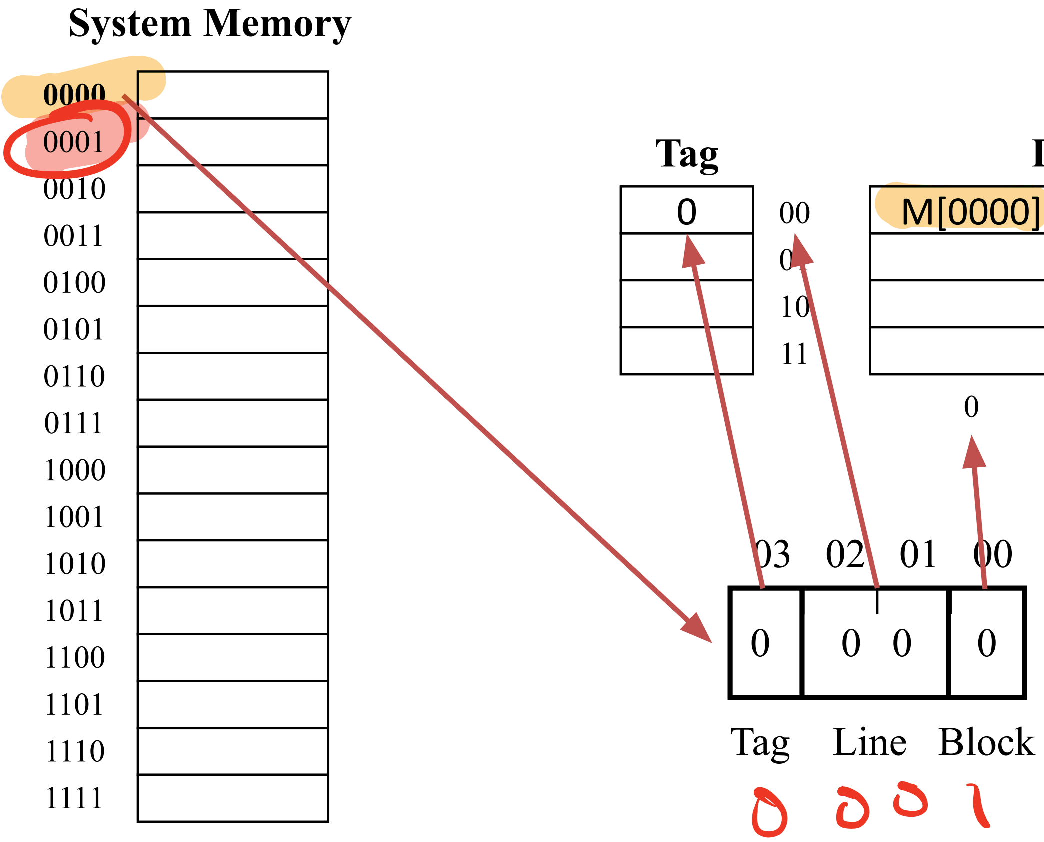
Line portion of address determines which cache line to use (direct mapped)

Direct Mapped Cache



Requested data loaded from system memory into cache, tag metadata set

Direct Mapped Cache



System Memory

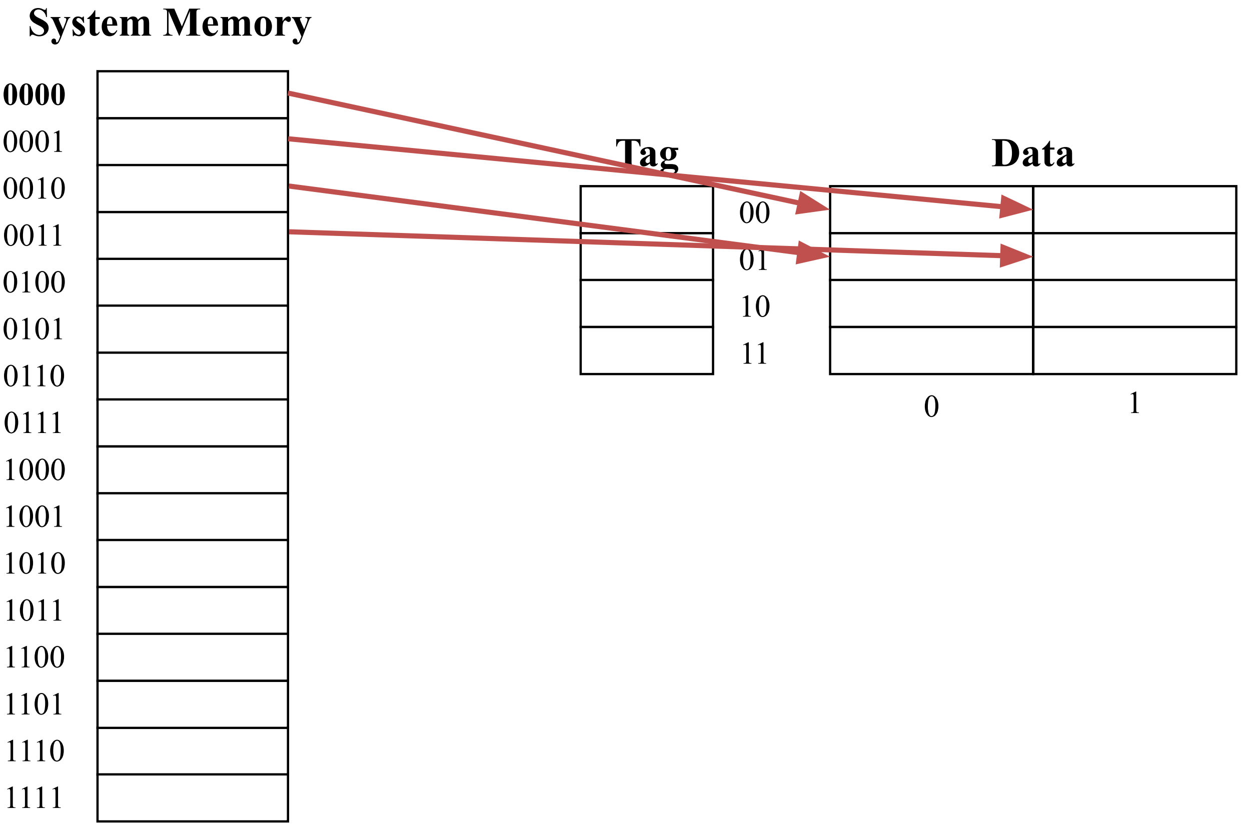
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Tag		Data	
0	00	M[0000]	M[0001]
	01		
	10		
	11		
		0	1

03	02 01	00
0	0 0	0
Tag	Line	Block
0	0 0	1

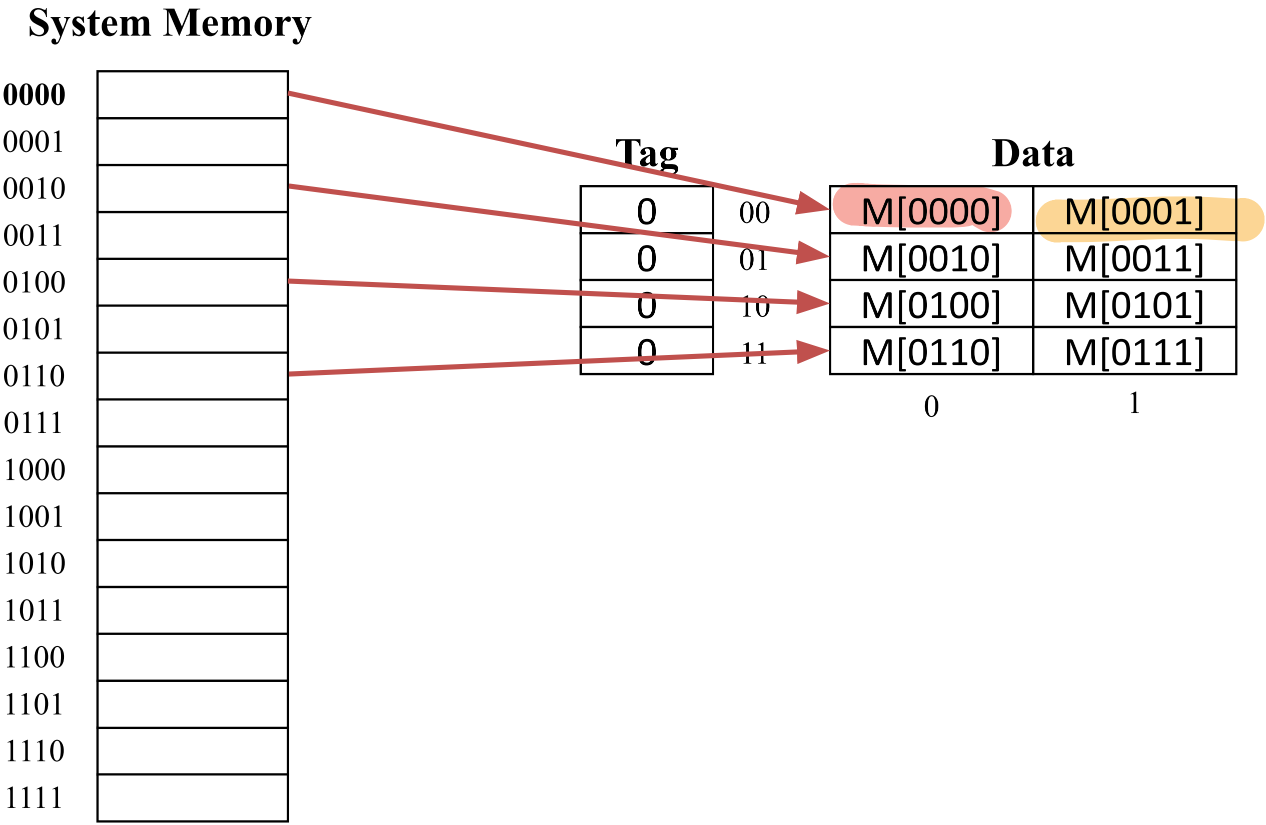
Entire cache block is loaded into cache when any element is read (spatial locality)

Direct Mapped Cache



Only one possible cache line location for each memory address (direct mapping)

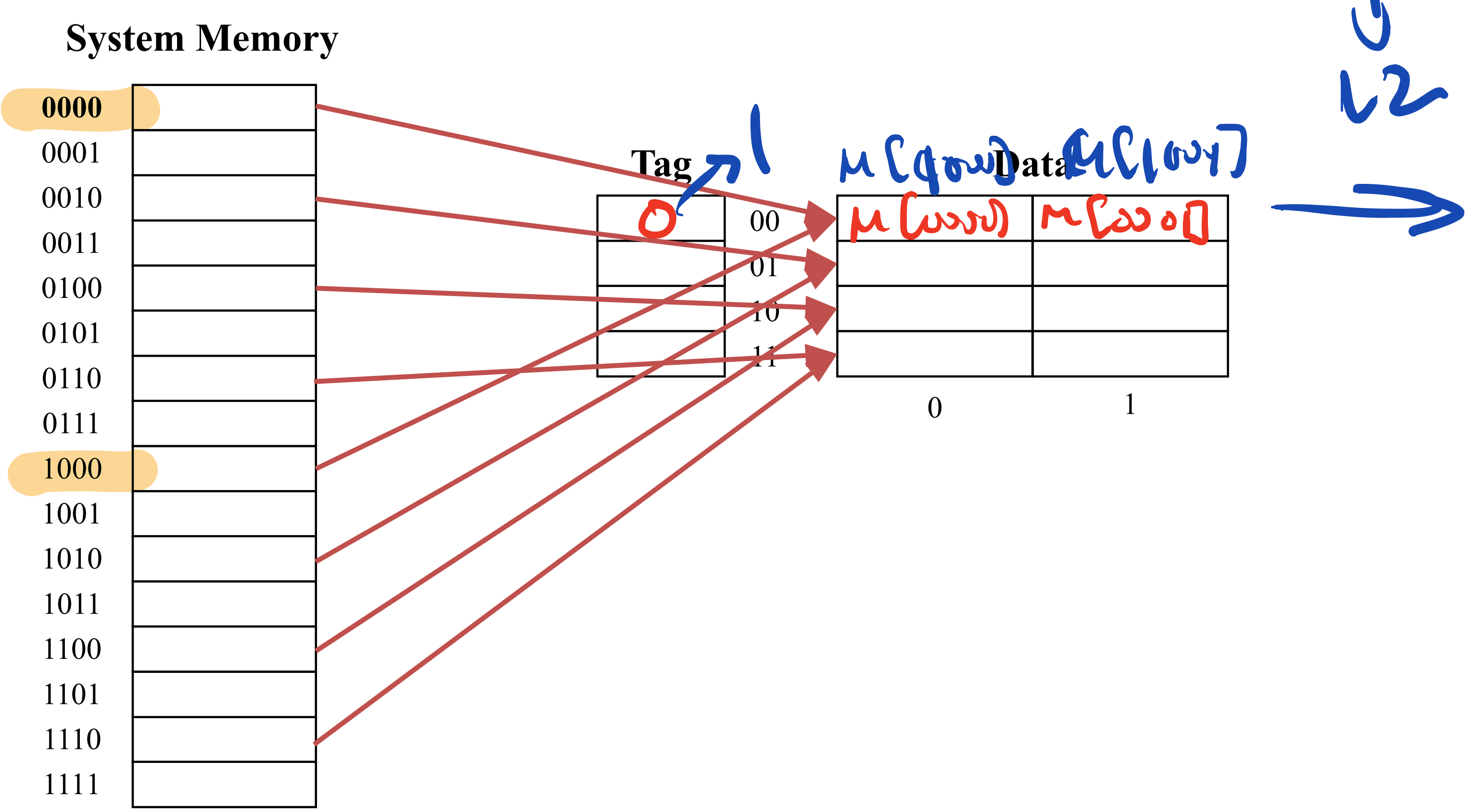
Direct Mapped Cache



Tag		Data	
0	00	M[0000]	M[0001]
0	01	M[0010]	M[0011]
0	10	M[0100]	M[0101]
0	11	M[0110]	M[0111]
		0	1

Cache filled (sequential accesses yield 4 misses, 4 hits)

Direct Mapped Cache

C*
↓


System Memory

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Tag

Data

00
01
10
11

0
1

Conflict: cache is smaller than memory, so these addresses map to the same cache lines

Direct Mapping

- Simple Control Logic
- Great Spatial Locality
- Awful Temporal Locality

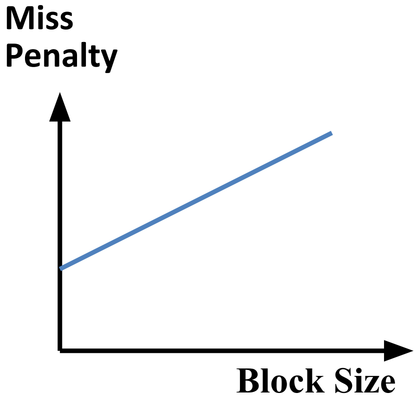
- When does this fail in a direct mapped cache?

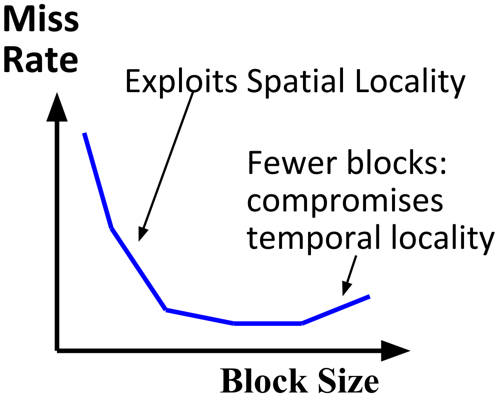
```
char *image1, *image2;

int stride;
for(int i=0;i<size;i+=stride) {
   diff +=abs(image1[i] - image2[i]);
}
```

Block Size Tradeoff

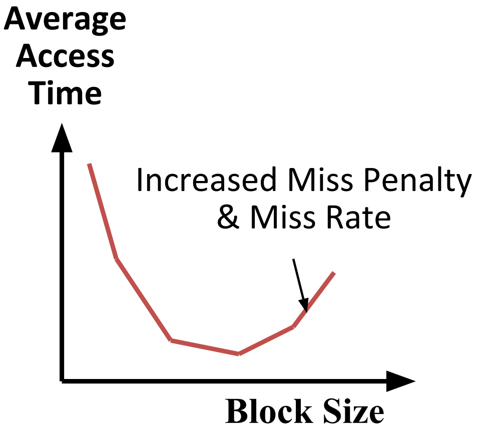
- With **fixed cache size**, increasing Block size(M):
  - Worse Miss penalty: longer to fill block
    - Can be partially be hidden by sending the requested data first
  - Better Spatial Locality
  - Worse Temporal Locality





Exploits Spatial Locality

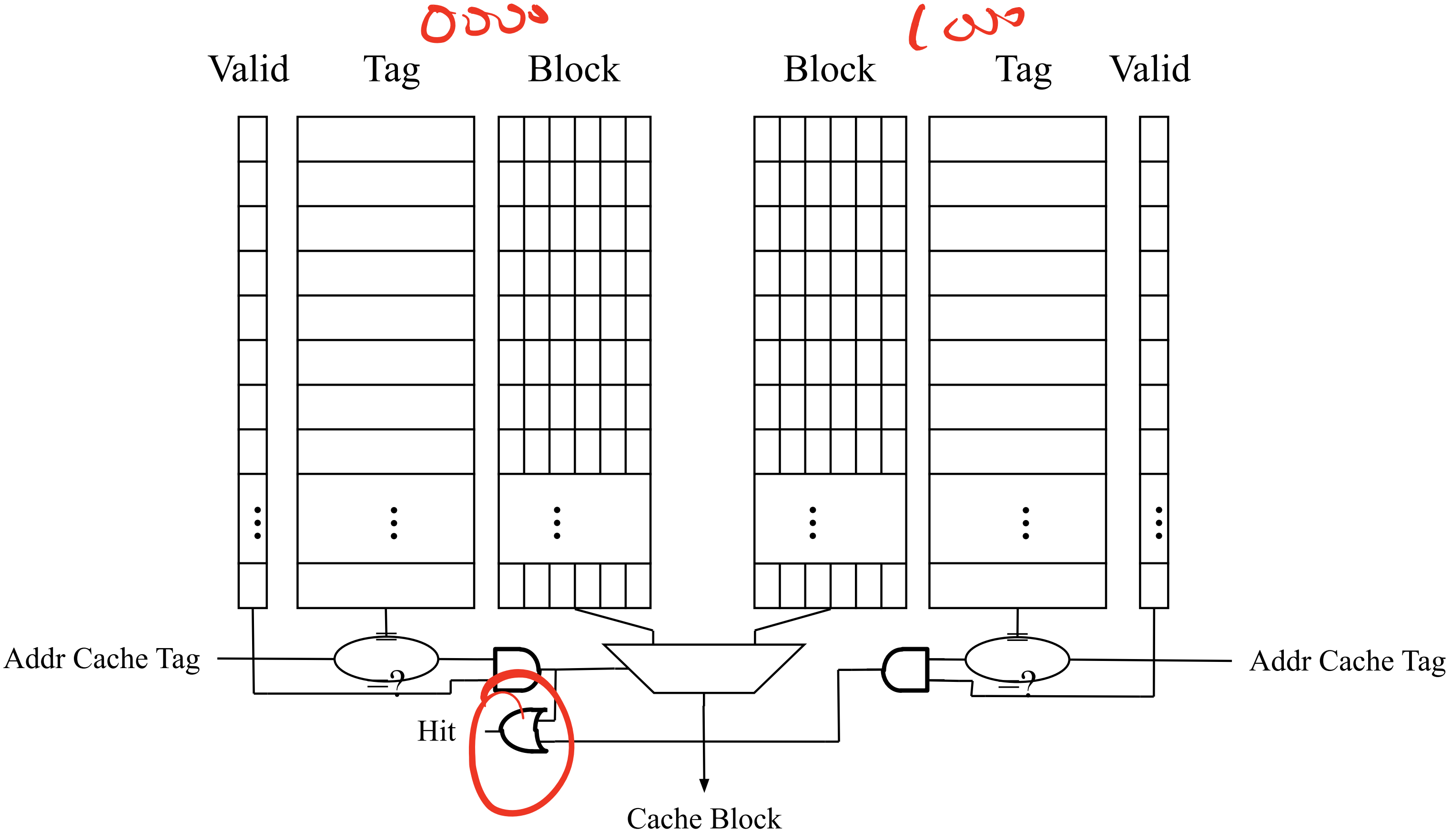
Fewer blocks: compromises temporal locality



Increased Miss Penalty & Miss Rate

N-Way Set Associative Mapping

- Like N parallel direct map caches



Associative Mapping

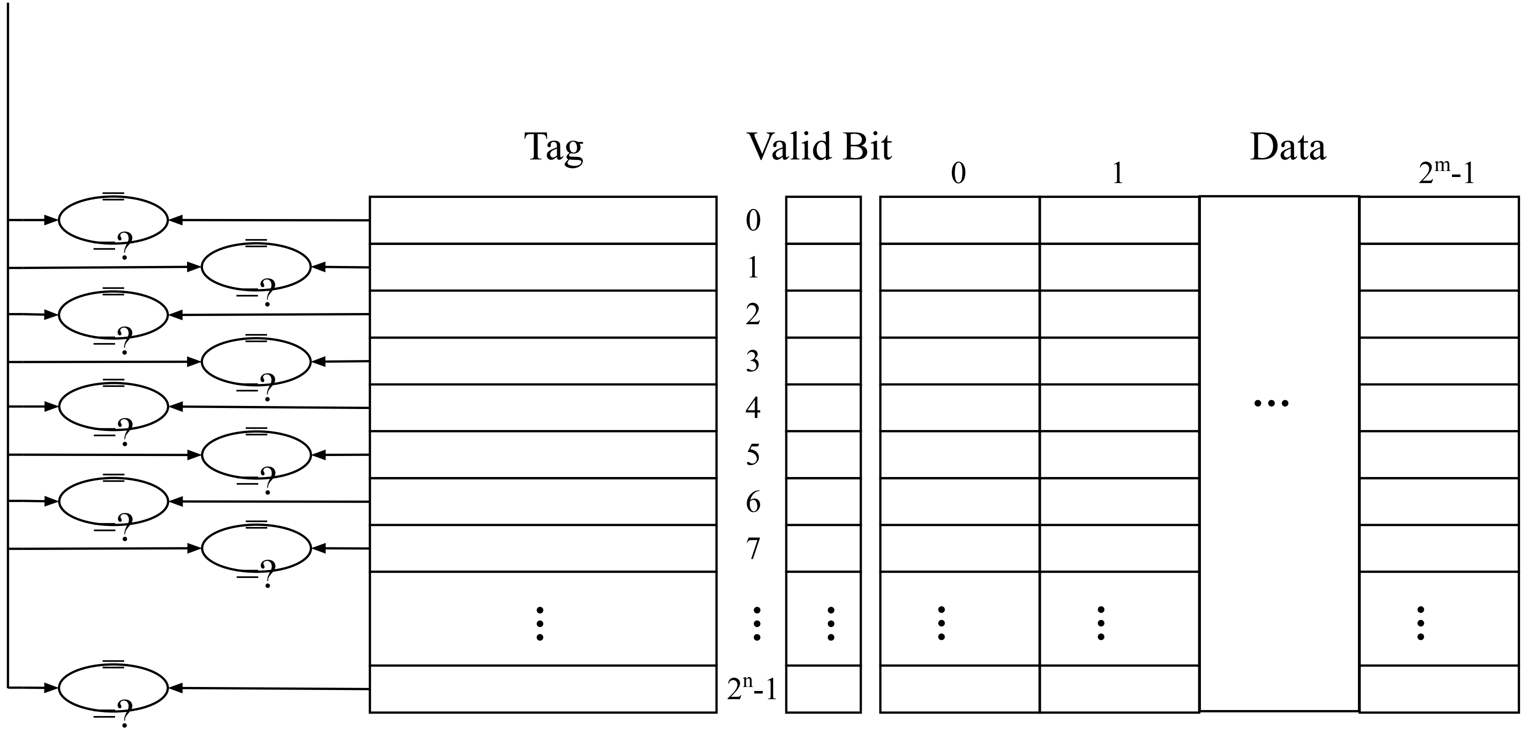
- Can handle up to N conflicts without eviction

- Better Temporal Locality
  - Assuming good Eviction policy

- More Complicated Control Logic
  - Slower to confirm hit or miss
  - Slower to find associated cache line

Fully Associative Cache

- Full Associative = 2^N Way Associative
  - Everything goes Anywhere!



Cache Arrangement

- **Direct Mapped** – Memory addresses map to particular location in that cache
  - 1-Way Associative Set

- **Fully Associative** – Data can be placed anywhere in the cache
  - 2^N-Way Associative Set

- **N-way Set Associative** – Data can be placed in a limited number of places in the cache depending upon the memory address

Replacement Methods

- If we need to load a new cache line, where does it go?
- Direct-mapped

  Only one possible location

- Set Associative

  N locations possible, optimize for temporal locality?

- Fully Associative

  All locations possible, optimize for temporal locality?

What gets evicted?

- Approach #1: Random
  - Just arbitrarily pick from possible locations
- Approach #2: Least Recently Used (LRU)
  - Use temporal locality
  - Must track somehow – extra bits to recent usage
- In practice, Random ~12% worse than LRU

3 C’s of Cache Misses

- Compulsory/Coldstart
  - First access to a block – basically unavoidable
  - For long-running programs this is a small fraction of misses
- Capacity
  - Had been loaded, evicted by too many other accesses
  - Can only be mitigated by cache size
- Conflict
  - Had been loaded, evicted by mapping conflict
  - Mitigated by associativity

Cache Miss Comparison

- Fill in the blanks: Zero, Low, Medium, High, Same for all

	Direct Mapped	N-Way Set Associative	Fully Associative
Cache Size: Small, Medium, Big?	Big (Few comparators)	Medium	Small (lots of comparators)
Compulsory Miss:			
Capacity Miss			
Conflict Miss			

Cache Miss Comparison

- Fill in the blanks: Zero, Low, Medium, High, Same for all

	Direct Mapped	N-Way Set Associative	Fully Associative
Cache Size: Small, Medium, Big?	Big (Few comparators)	Medium	Small (lots of comparators)
Compulsory Miss:	Same	Same	Same
Capacity Miss	Low	Medium	High
Conflict Miss	High	Medium	Zero

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

00001

00110

00001

11010

00110

	Valid Bit	Tag	Data
0			
1			
2			
3			

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

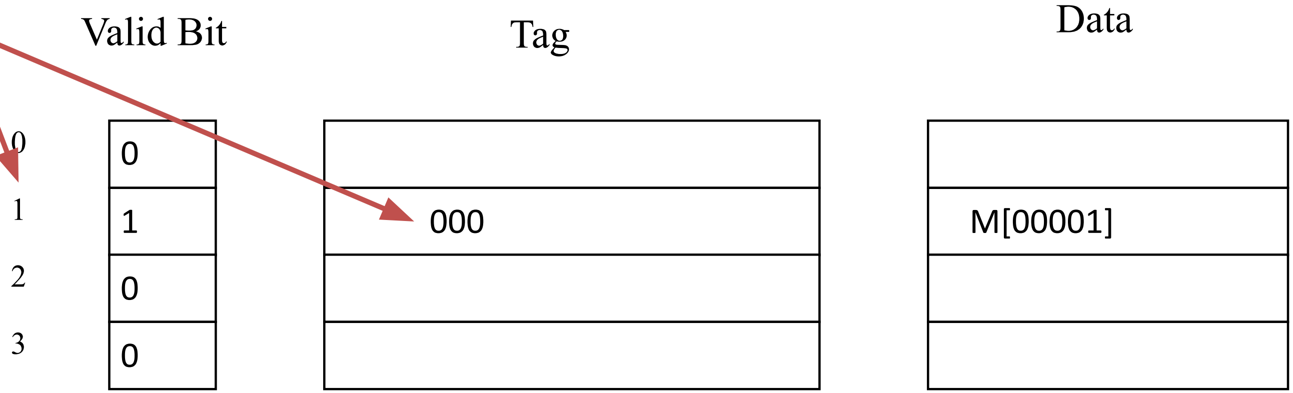
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	0		
3	0		

Low N bits of address determine cache Line, rest is Tag (M=0 so no block address)

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

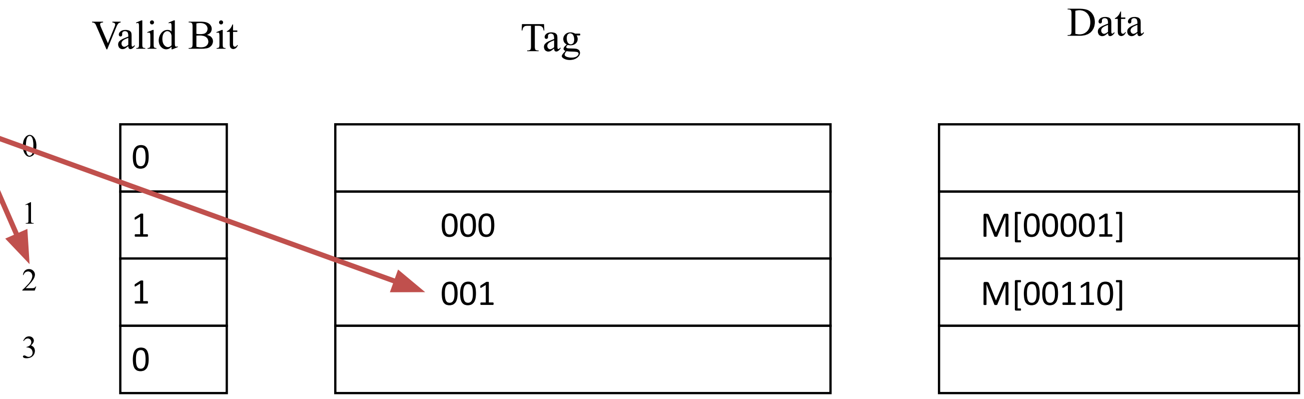
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	1	001	M[00110]
3	0		

Access misses (compulsory), fills cache line 2

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

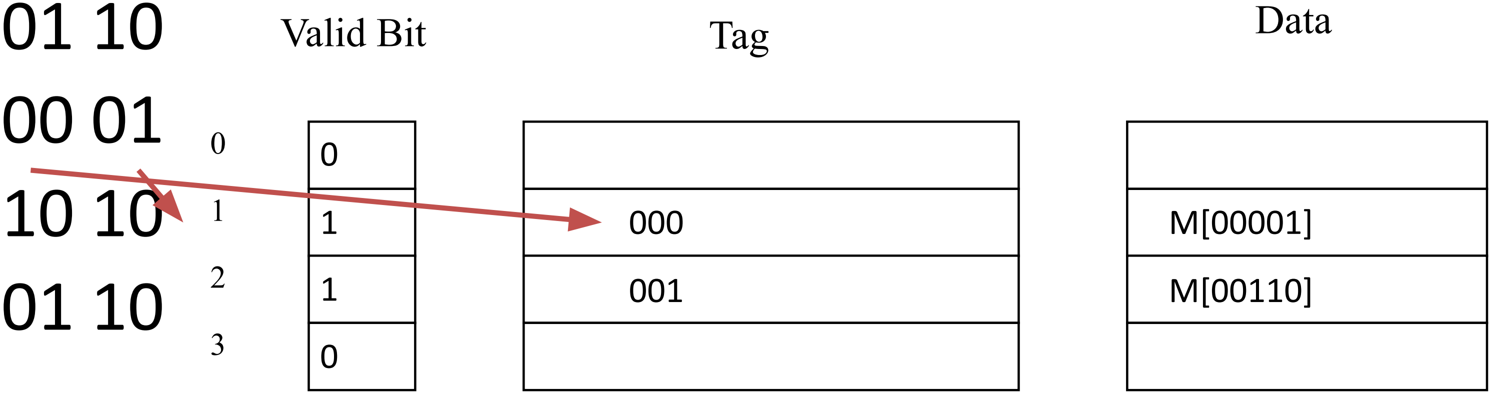
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	1	001	M[00110]
3	0		

Hits in cache

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

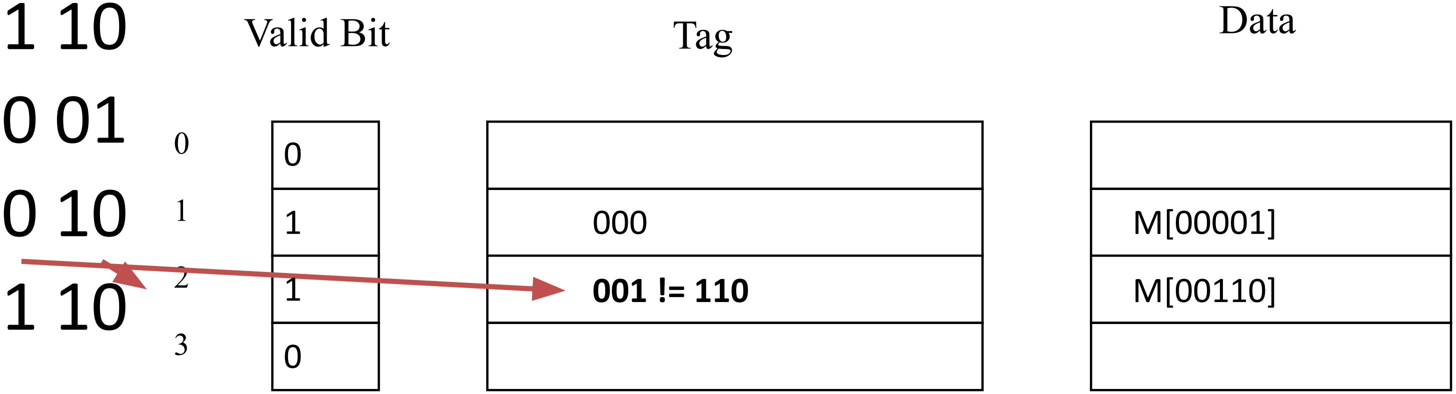
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	1	**001 != 110**	M[00110]
3	0		

Miss (compulsory)

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

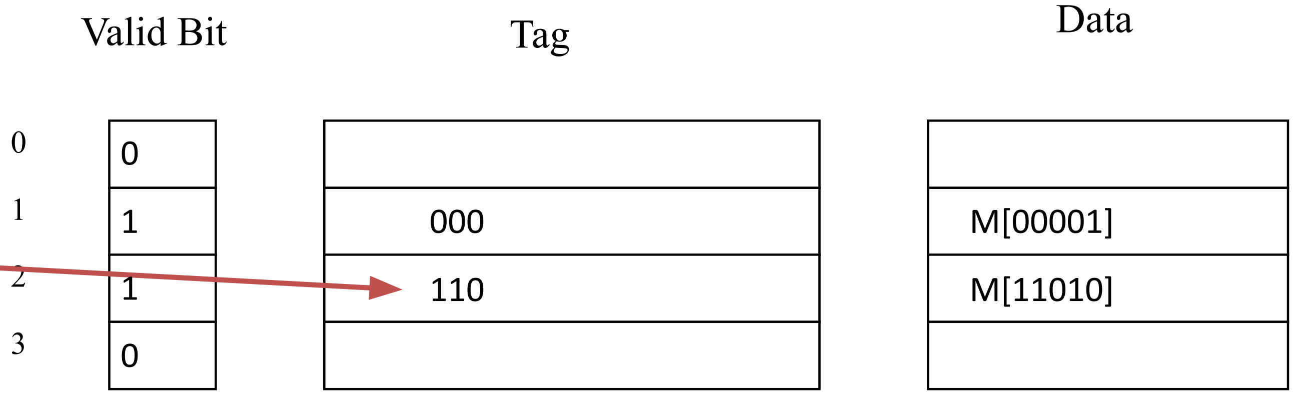
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	1	110	M[11010]
3	0		

Evicts previous data from cache

Cache Access Example

Assume 4 byte direct-mapped cache (N=2, M=0)

Access pattern:

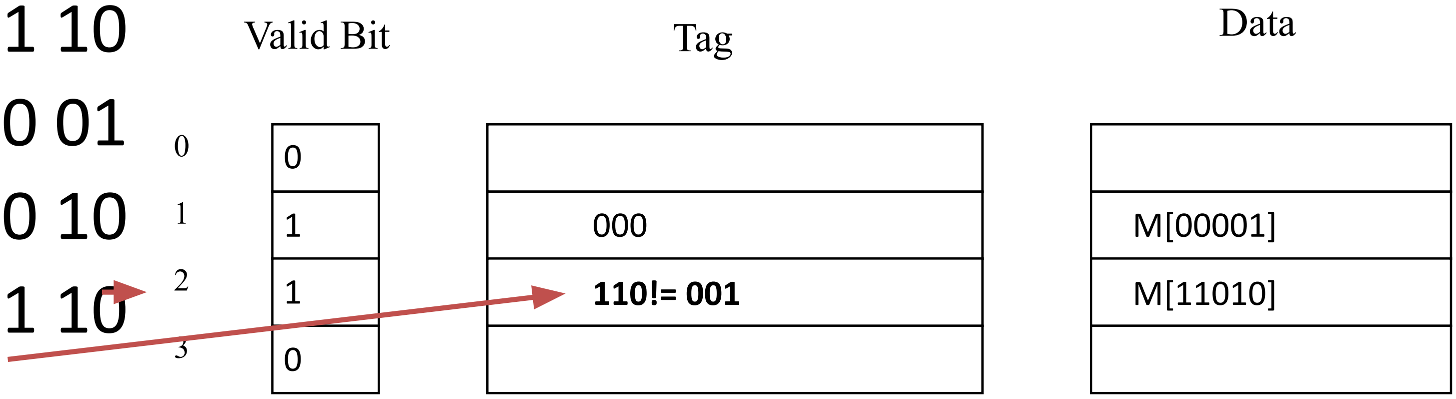
000 01

001 10

000 01

110 10

001 10



	Valid Bit	Tag	Data
0	0		
1	1	000	M[00001]
2	1	**110!= 001**	M[11010]
3	0		

Miss (conflict) – this used to be in cache but was evicted

Cache Miss Example

- 8-word cache, 8-byte blocks. Determine types of misses (CAP, COLD, CONF).

Byte Addr	Block Addr	Direct Mapped	2-Way Assoc	Fully Assoc
0	0	Cold	Cold	Cold
4	0	Hit (in 0's block)	Hit (in 0's block)	Hit (in 0's block)
8	1	Cold	Cold	Cold
24	3	Cold	Cold	Cold
56	7	Cold	Cold	Cold
8	1	Conf (w/56)	Conf (w/56)	Hit
24	3	Hit	Conf (w/8)	Hit
16	2	Cold	Cold	Cold
0	0	Hit	Hit	Cap
	Total Miss:	6	7	6

Cache Miss Example

- 8-word cache, 8-byte blocks. Determine types of misses (CAP, COLD, CONF).

Byte Addr	Block Addr	Direct Mapped	2-Way Assoc	Fully Assoc
0				
4				
8				
24				
56				
8				
24				
16				
0				
	Total Miss:			

Cache Miss Example

- 8-word cache, 8-byte blocks. Determine types of misses (CAP, COLD, CONF).

Byte Addr	Block Addr	Direct Mapped	2-Way Assoc	Fully Assoc
0	0	Cold		
4	0	Hit (in 0's block)		
8	1	Cold		
24	3	Cold		
56	7	Cold		
8	1	Conf (w/56)		
24	3	Hit		
16	2	Cold		
0	0	Hit		
	Total Miss:	6		

With Remaining Time

- Finish Cache Example (Answers Below)
- Find out what your computer’s caches are

- Write your own “Pop Quiz” on Caching

- We will exchange & take quizes
  - Not graded
  - Then discuss

Quiz Ideas

- Describe a specific Cache:
  - How big is the tag? Where does address X map?
- Describe an access pattern:
  - Hit Rate? Average Access Time?

Cache Summary

- Software Developers must respect the cache!
  - Usually a good place to start optimizing
    - Especially on laptops/desktops/servers
  - Cache Line size? Page size?

- Cache design implies a usage pattern
  - Very good for instruction & stack
  - Not as good for Heap
  - Modern languages are very heapy

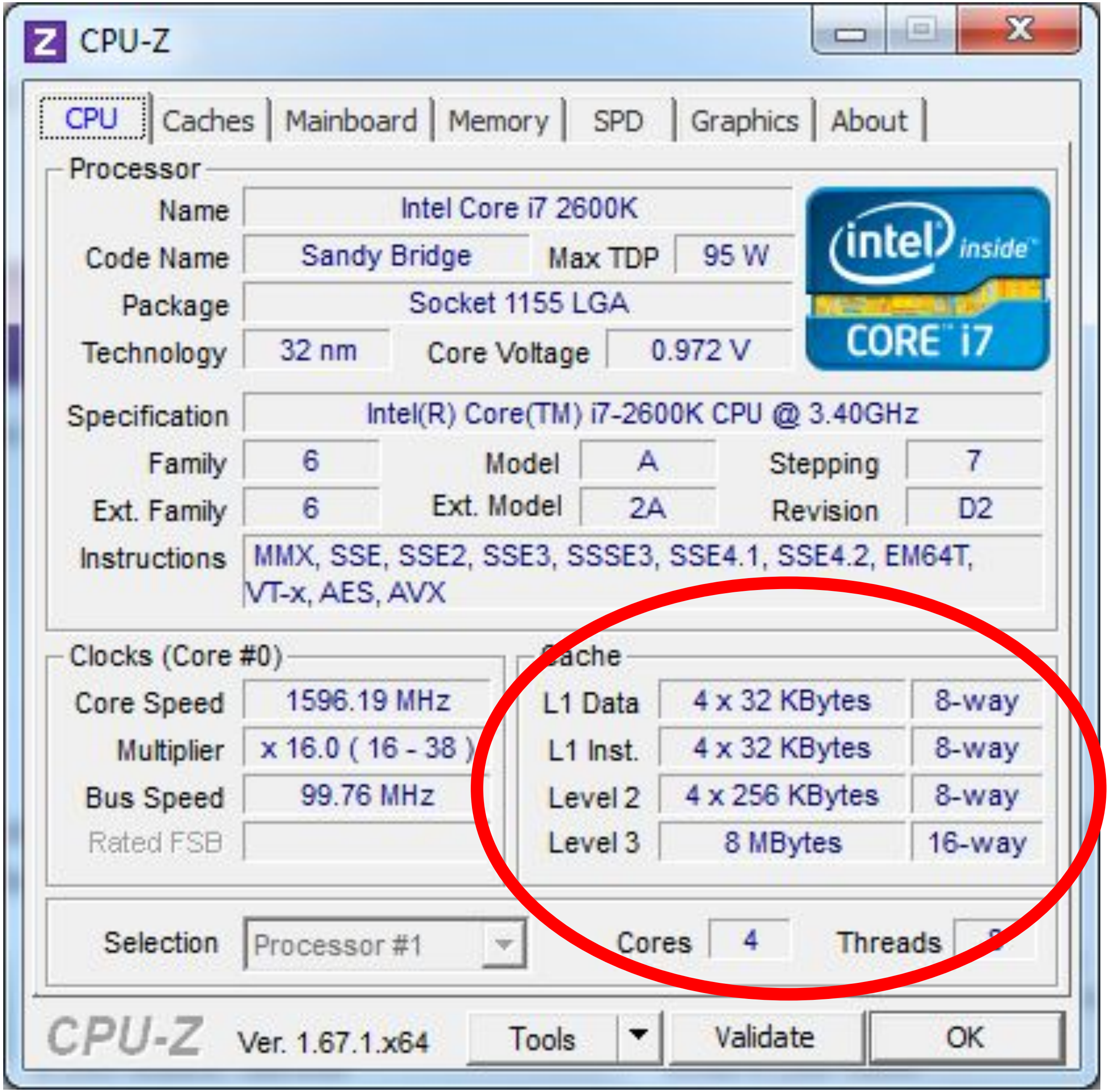
Matrix Multiplication

```
for (k = 0; k < n; k++){ for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
      c[k][i] = c[k][i] + a[k][j]*b[j][i];
}}}

Vs

for (k = 0; k < n; k++){ for (i = 0; i < n; i++){
   for (j = 0; j < n; j++){
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
}}}
```

- Which has better locality?



Processor

Name: Intel Core i7 2600K

Code Name: Sandy Bridge | Max TDP: 95 W

Package: Socket 1155 LGA

Technology: 32 nm | Core Voltage: 0.972 V

Specification: Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz

Family: 6 | Model: A | Stepping: 7

Ext. Family: 6 | Ext. Model: 2A | Revision: D2

Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, VT-x, AES, AVX

Clocks (Core #0)

Core Speed: 1596.19 MHz

Multiplier: x 16.0 (16 - 38)

Bus Speed: 99.76 MHz

Rated FSB:

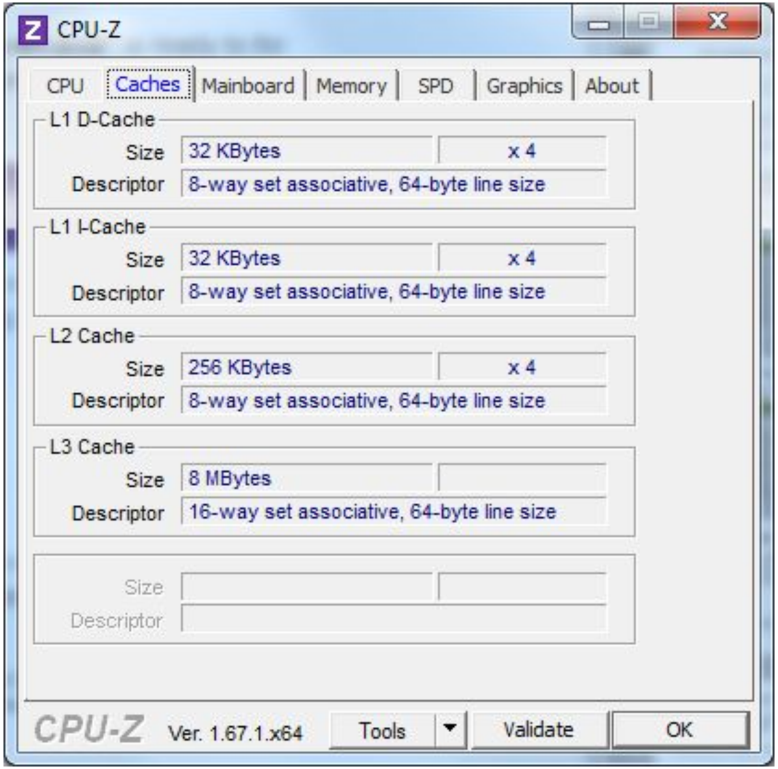
Cache

L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	8-way
Level 3	8 MBytes	16-way

Selection: Processor #1 | Cores: 4 | Threads: 8

Ver. 1.67.1.x64

My L1 Data Cache



L1 D-Cache

Size: 32 KBytes x 4

Descriptor: 8-way set associative, 64-byte line size

L1 I-Cache

Size: 32 KBytes x 4

Descriptor: 8-way set associative, 64-byte line size

L2 Cache

Size: 256 KBytes x 4

Descriptor: 8-way set associative, 64-byte line size

L3 Cache

Size: 8 MBytes

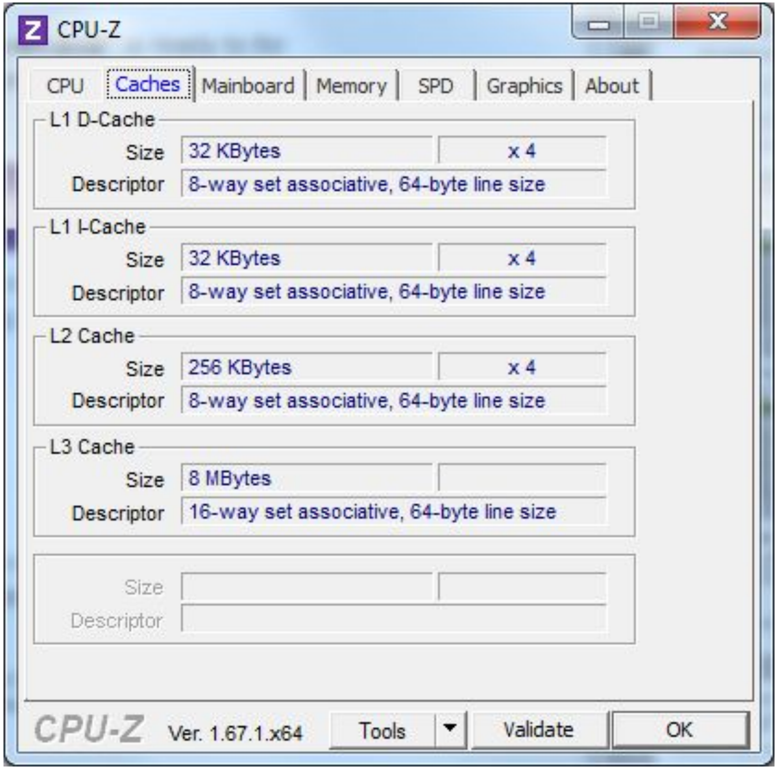
Descriptor: 16-way set associative, 64-byte line size

Size:

Descriptor:

- M = ?
- N = ?
- Tag Size = ?

My L1 Data Cache



L1 D-Cache

Size	32 KBytes	x 4
Descriptor	8-way set associative, 64-byte line size	

L1 I-Cache

Size	32 KBytes	x 4
Descriptor	8-way set associative, 64-byte line size	

L2 Cache

Size	256 KBytes	x 4
Descriptor	8-way set associative, 64-byte line size	

L3 Cache

Size	8 MBytes	
Descriptor	16-way set associative, 64-byte line size	

Size		
Descriptor		

- M = 6 bits
  - 64-byte line size
  - 64 byte blocks
  - 2^6 = 64
- N = 6 bits
  - 32kByte -> 15 bits
  - 8 way -> 3 fewer bits
  - M=6 -> 6 fewer bits
- Tag Size = 52 bits
  - 64 bit address space
  - Whatever is left over

Memory System Summary

Goal: provide the illusion of large, fast memory

Use hierarchy of memory types with complementary properties

- Small/fast register file <-> huge/slow disk

Cache performance enabled by spatial and temporal locality in programs