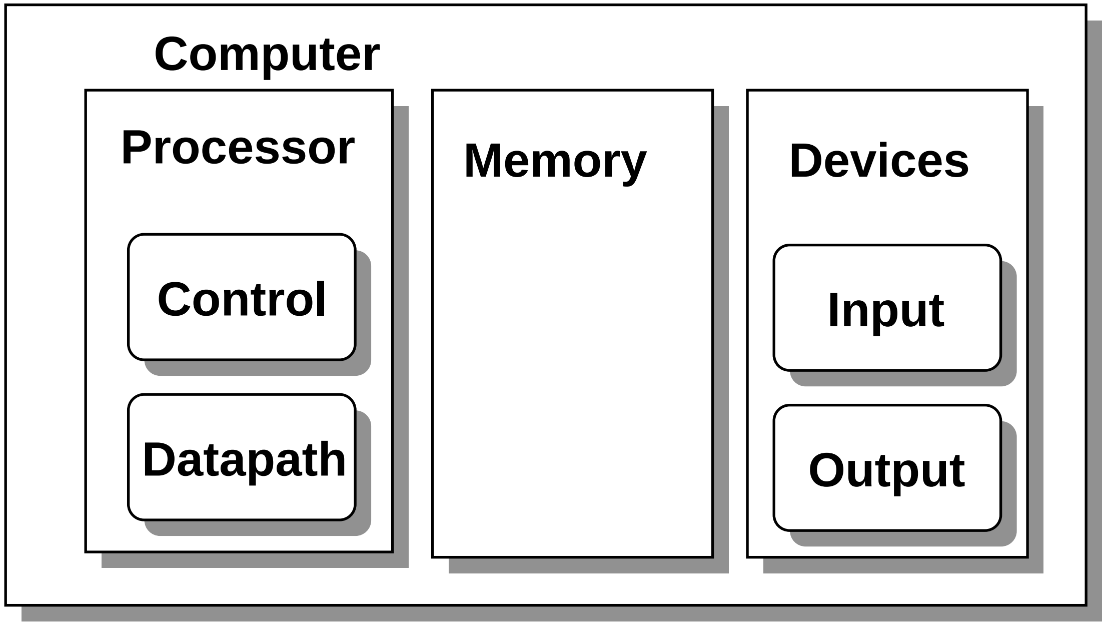
Memory Hierarchy: Caches, Virtual Memory

Big memories are slow

Fast memories are small



Need to get fast, big memories

Random Access Memory

Dynamic Random Access Memory (DRAM)

- High density, low power, cheap, but slow
- Dynamic since data must be “refreshed” regularly
- Random Access since arbitrary memory locations can be read

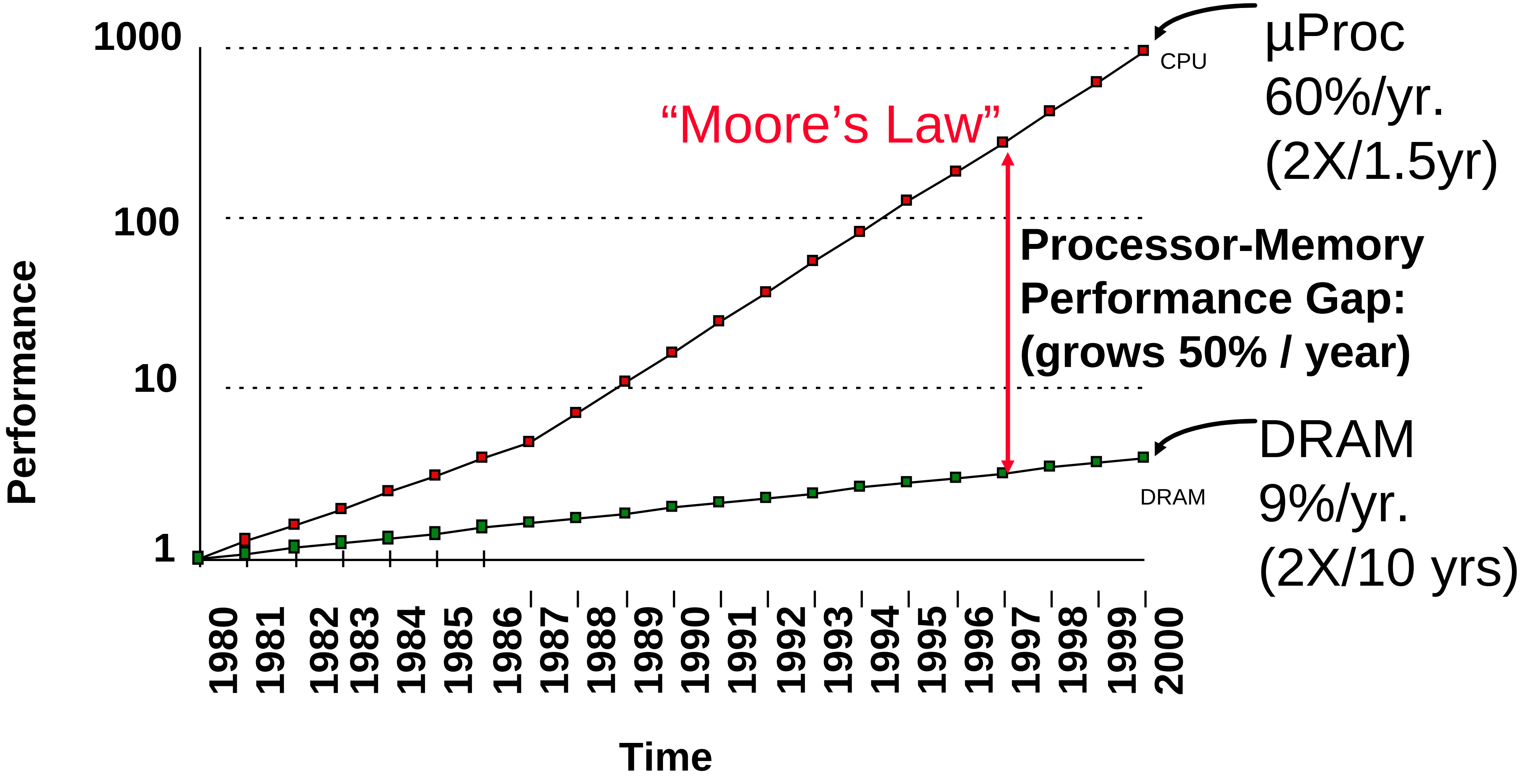
Static Random Access Memory

- Low density, high power, expensive
- Static since data held as long as power is on
- Fast access time, often 2 to 10 times faster than DRAM

Technology	Access Time	$/MB in 1997
SRAM	5-25ns	$100-$200
DRAM	60-120ns	$5-$10
Disk	(10-20)x10^{6}ns	$0.10-$0.20

Technology Trends

Processor-DRAM Memory Gap (latency)



"Moore's Law"

µProc
60%/yr.
(2X/1.5yr)

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

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

The Problem

The Von Neumann Bottleneck

- Logic gets faster
- Memory capacity gets larger
- Memory speed is not keeping up with logic

Cost vs. Performance

- Fast memory is expensive
- Slow memory can significantly affect performance

Design Philosophy

- Use a hybrid approach that uses aspects of both
- Keep frequently used things in a small amount of fast/expensive memory
    - “Cache”
- Place everything else in slower/inexpensive memory (even disk)
- Make the common case fast

Locality

Programs access a relatively small portion of the address space at a time

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

Types of 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

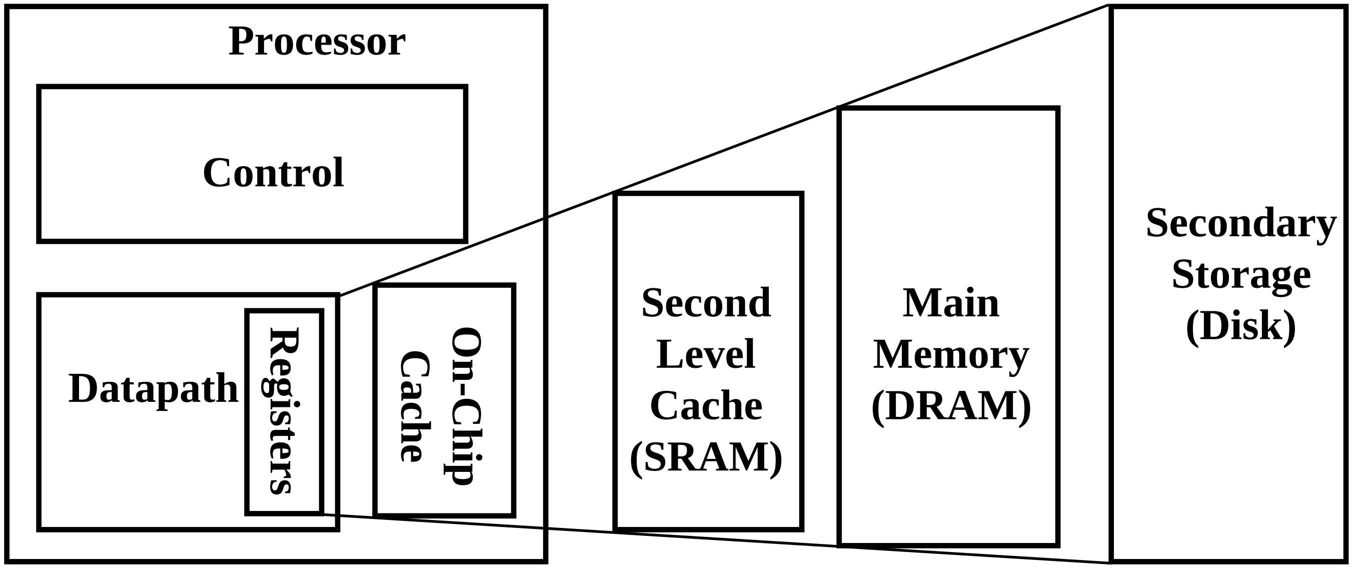
Locality guides caching

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.



Name	Register	Cache	Main Memory	Disk Memory
Speed	<1ns	<10ns	60ns	10 *ms*
Size	100 Bs	KBs	MBs	GBs

Cache Terminology

Block – Minimum unit of information transfer between levels of the hierarchy

Block addressing varies by technology at each level

Blocks are moved one level at a time

Upper vs. **lower** level – “upper” is closer to CPU, “lower” is futher away

Hit – Data appears in a block 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 high hit rate & low hit time, since miss penalty is large

Average Memory Access Time (AMAT)

Apply average access time to entire hierarchy.

Cache Access Time Example

Level	Hit Time	Hit Rate	Access Time
L1	1 cycle	95%	1 + 0.05 × 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)

Handling A Cache Miss

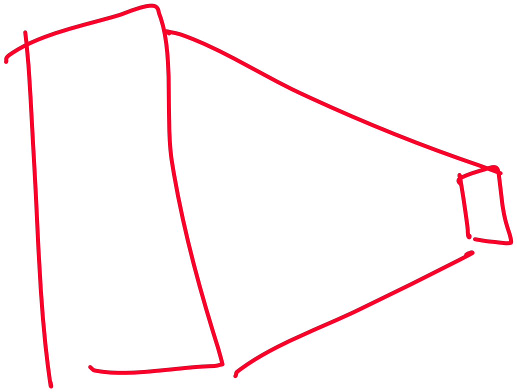
Processor expects a cache hit (1 cycle), so no effect on hit.

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

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



Exploiting Locality

Spatial locality

- Move blocks consisting of multiple contiguous words to upper level

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

Cache Arrangement

How should the data in the cache be organized?

Caches are smaller than the full memory, so multiple addresses must map to the same cache “line”

Direct Mapped – Memory addresses map to particular location in that cache

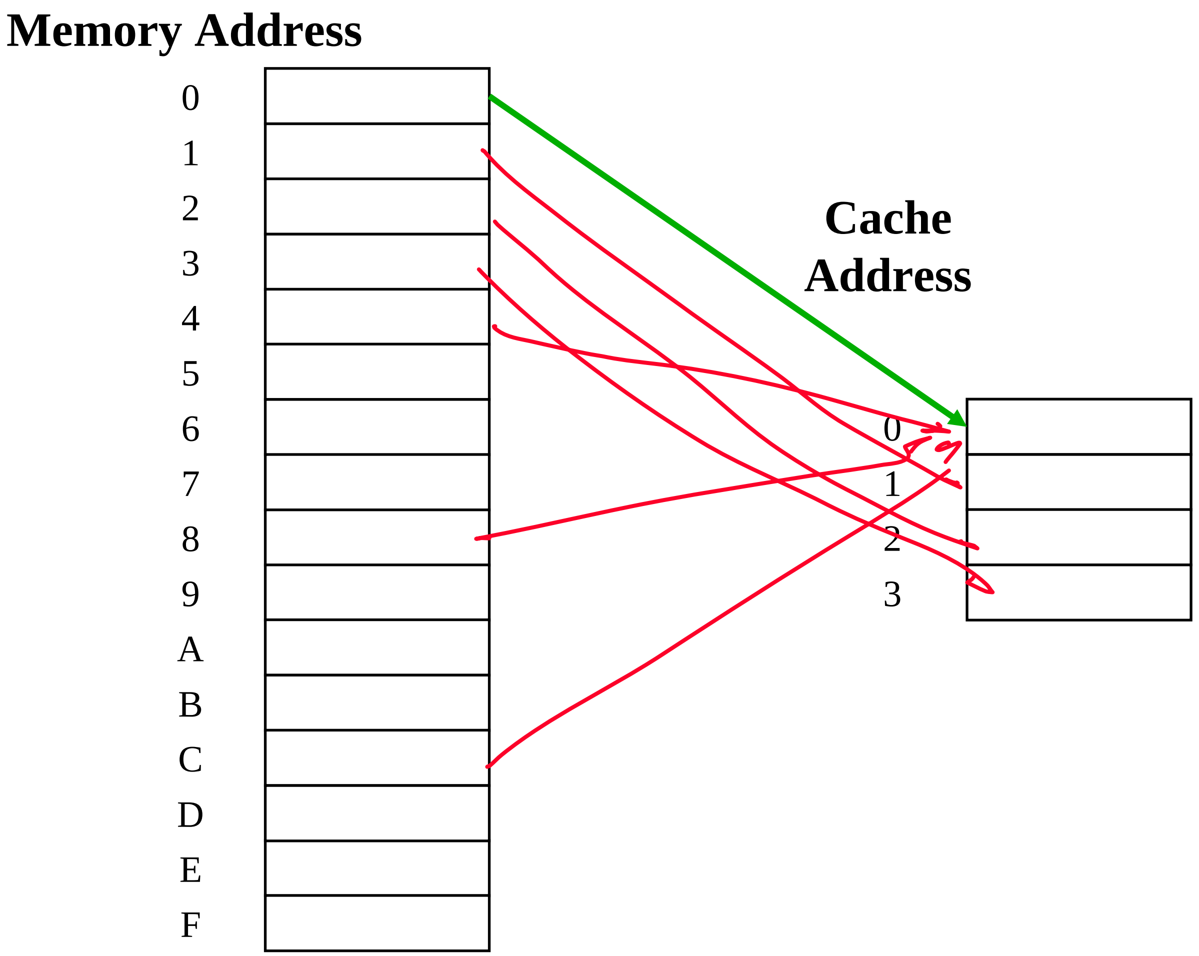
Fully Associative – Data can be placed anywhere in the cache

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

Direct Mapped Cache

4 byte direct mapped cache with 1 byte blocks

- Optimize for spatial locality (close blocks likely to be accessed soon)



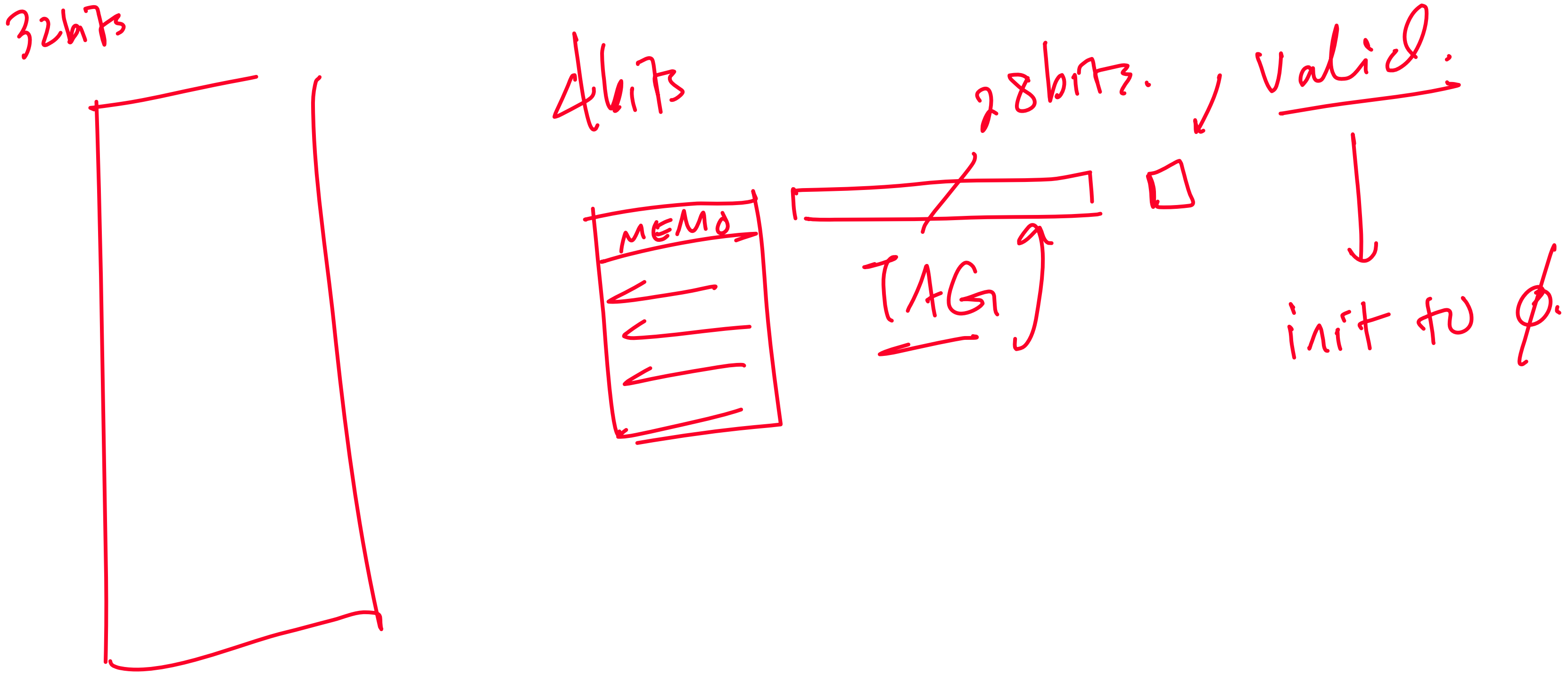
Finding A Block

Each location in the cache can contain a number of different memory locations

Cache 0 could hold 0, 4, 8, 12, …

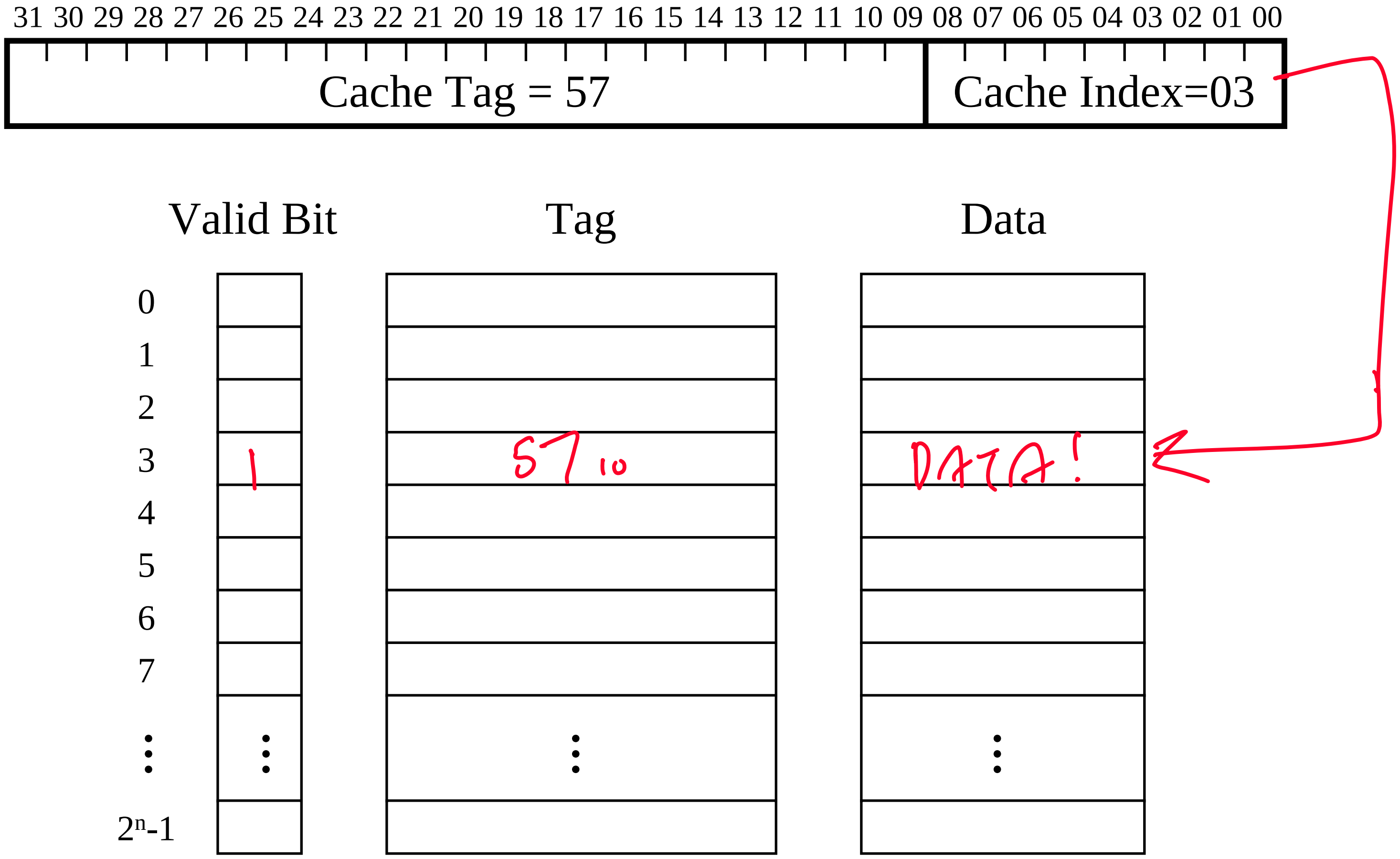
We add a **tag** to each cache entry to identify which address it currently contains

What must we store?



Cache Tag & Index

Assume 2^n byte direct mapped cache with 1 byte blocks



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

Cache Tag = 57 | Cache Index=03

	Valid Bit	Tag	Data
0			
1			
2			
3	1	57_{10}	DATA!
4			
5			
6			
7			
⋮	⋮	⋮	⋮
2^n-1			

Cache Access Example

Assume 4 byte cache

Access pattern:

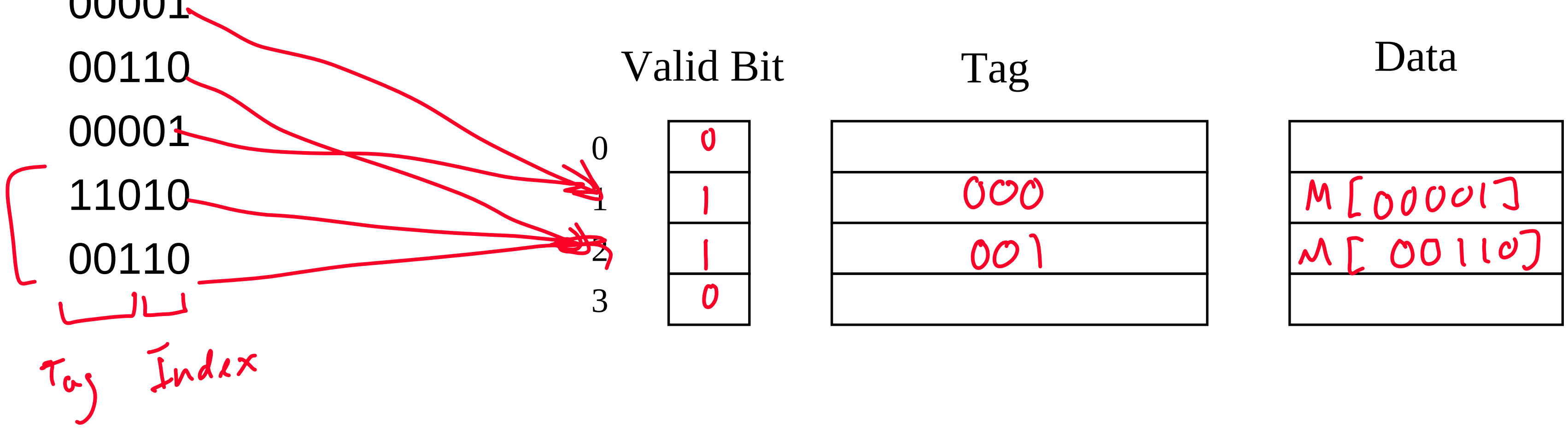
00001

00110

00001

11010

00110



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

Cache Access Example (cont.)

Assume 4 byte cache

Access pattern:

00001

00110

00001

11010

00110

	Valid Bit	Tag	Data
0			
1			
2			
3			

Cache Access Example (cont. 2)

Assume 4 byte cache

Access pattern:

00001

00110

00001

11010

00110

	Valid Bit	Tag	Data
0			
1			
2			
3			

Cache Size Example

How many total bits are required for a direct-mapped cache with 64 KB of data and 1-byte blocks, assuming a 32-bit address?

Index bits: 64KB = 2^{16} block = 16 bit index

Bits/block: 25 bits/block.

Data: 8 bits

Valid: 1 bit

Tag: 32 - 16 = 16

↑ index

Total size: → 25×2^{16} = 200 KB = ~3x size of data cached