Computer Architecture

Cache Design I

Farshad Khun Jush

Some slides come from Mirjana Stojilovic

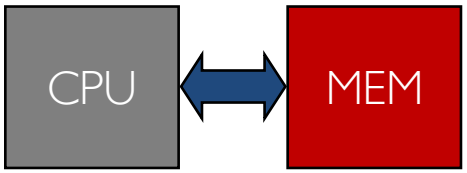
Outline

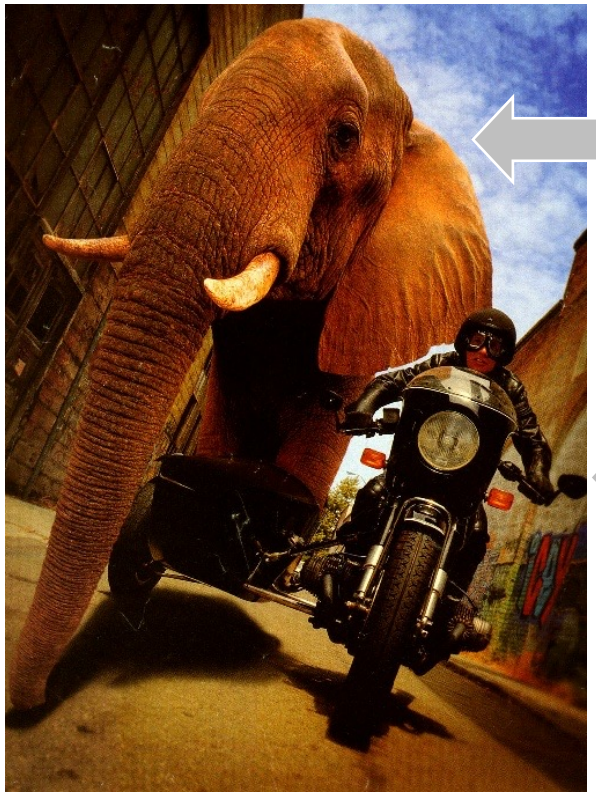
- *Memories, Reminder*
- Basic Cache Design
- Spatial and Temporal Locality
- Cache Hits and Misses
  - Strategies on Read
  - Strategies on Write
- Eviction Policies
- Full associative Caches
- Direct-Mapped Caches
- Summary

Reminder

- **SRAM**: **S**tatic **R**andom **A**ccess **M**emory
  - Static: content will last "forever"(as long as there's power)
  - Basic cell: similar to a flip flop, at circuit level
  - Low density, high power, **expensive**, **fast**

- **DRAM**: **D**ynamic **R**andom **A**ccess **M**emory
  - Dynamic: **needs to be "refreshed" regularly**
  - Basic cell: A capacitor and a switch (one transistor)
  - High density, low power, **cheap**, **slow**





This is your CPU's
main memory subsystem
You want it big.

This is your CPU.
You want it fast.

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger → Takes longer to determine the location

The Problem

- Ideal memory's requirements oppose each other

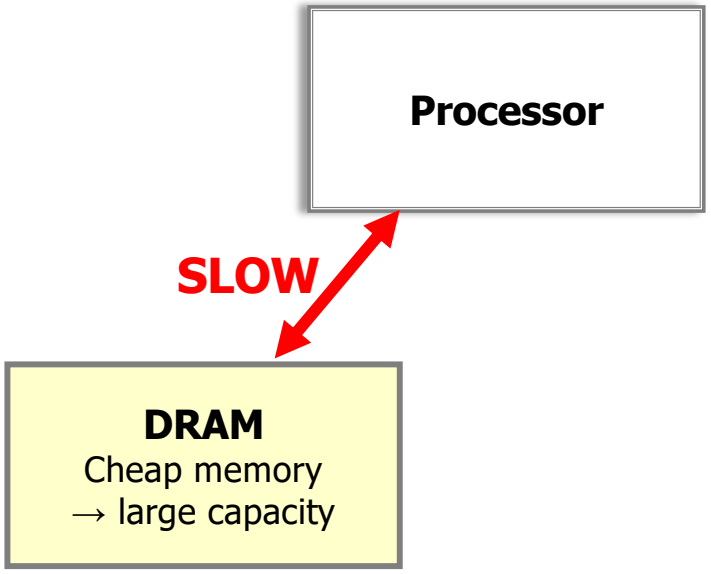
- Bigger is slower
  - Bigger → Takes longer to determine the location

- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. SSD vs. Disk vs. Tape

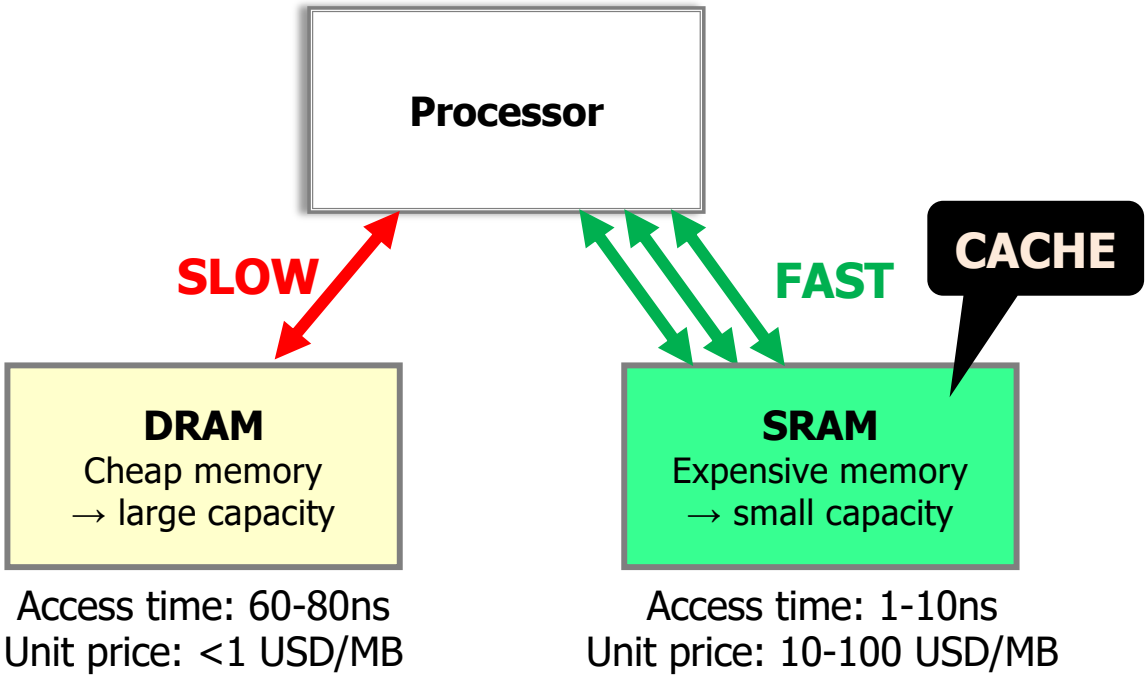
The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. SSD vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more banks, more ports, more channels, higher frequency or faster technology

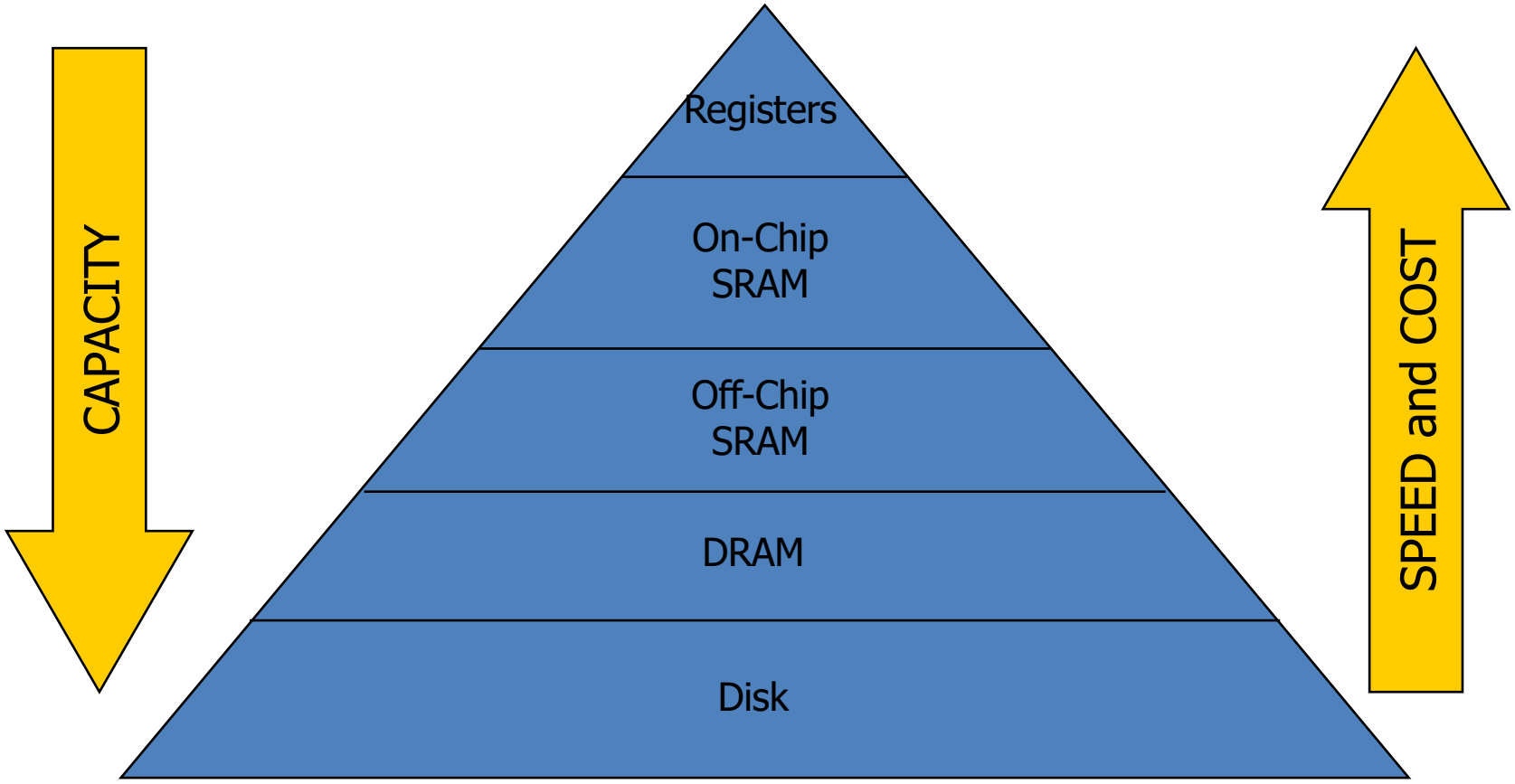
Can We Have Both Big and Fast Memory System??



Can We Have Both Big and Fast Memory System??



Memory Hierarchy

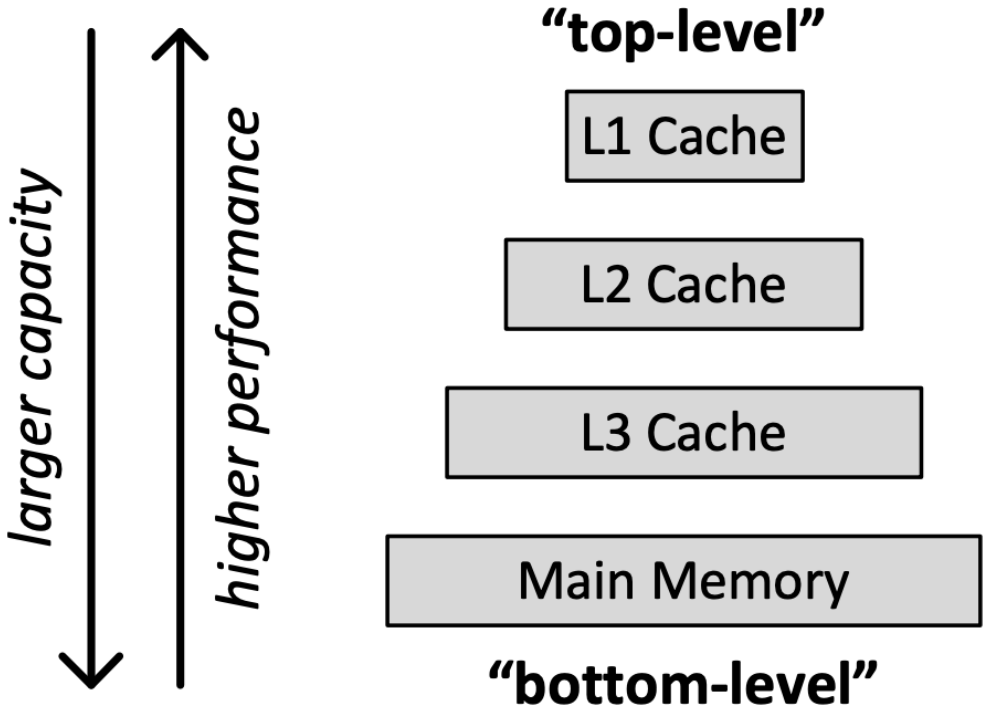


Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state

- All together: provide appearance of large, fast memory with cost of cheap, slow memory

Memory Hierarchy Example



capacity	latency
10's of KB	≈ 1ns
100's of KB	< 5ns
several MB	≈ 10ns
several GB	≈ 100ns

Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

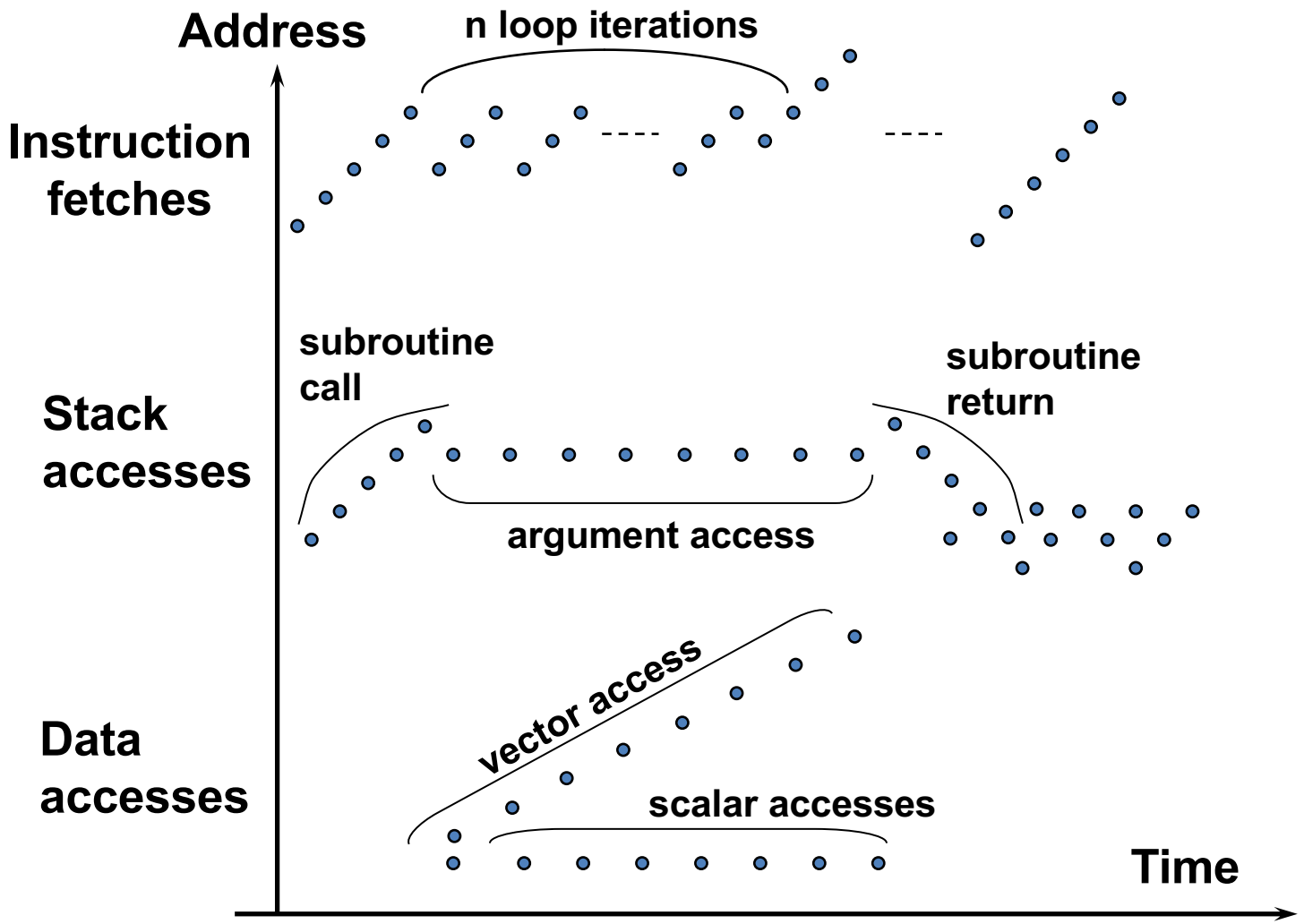
Why Does a Hierarchy Work?

- *Principle of Locality*: Programs access small portion of address space at any instant of time
- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies > 90% of references to multi-MB data set

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time
- What program structures lead to temporal and spatial locality in code?
- In data?

Typical Memory Reference Patterns



Spatial and Temporal Locality

Two important criteria to decide on the placement of instructions and data:

- **Temporal Locality**
  - Data that have been used **recently** have high likelihood of being used again
    - Code (instructions): loops, functions,…
    - Local variables and data structures

- **Spatial Locality**
  - Data that follow in the memory of the data that are currently being used are likely to be accessed in the future
    - Code: instructions are usually read **sequentially**
    - Arrays: array elements are usually accessed **sequentially**

How to Choose Which Data to Store (Allocate) in Which Memory?

```
1:  i = 0;
2:  sum = 0;
3:  while (i < 1024) {
4:    sum = sum + a[i];
5:    i = i + 1;
6:  }
```

How to Choose Which Data to Store (Allocate) in Which Memory?

```
1:  i = 0;
2:  sum = 0;
3:  while (i < 1024) {
4:    sum = sum + a[i];
5:    i = i + 1;
6:  }
```

- **Instructions** corresponding to lines 3–5 are **read over and over again:**
  - they should be stored in **fast memory**

How to Choose Which Data to Store (Allocate) in Which Memory?

```
1:  i = 0;
2:  sum = 0;
3:  while (i < 1024) {
4:    sum = sum + a[i];
5:    i = i + 1;
6:  }
```

- **Instructions** corresponding to lines 3–5 are **read over and over again:**
  - they should be stored in **fast memory**
- If **variables** `i` and `sum` are stored in memory and they are also **used often**
  - they should be stored in **fast memory**
- Additionally, one would like to anticipate the future and load **the subsequent** instructions and array elements to have them ready

Fundamental Constraints on the Allocation Policy

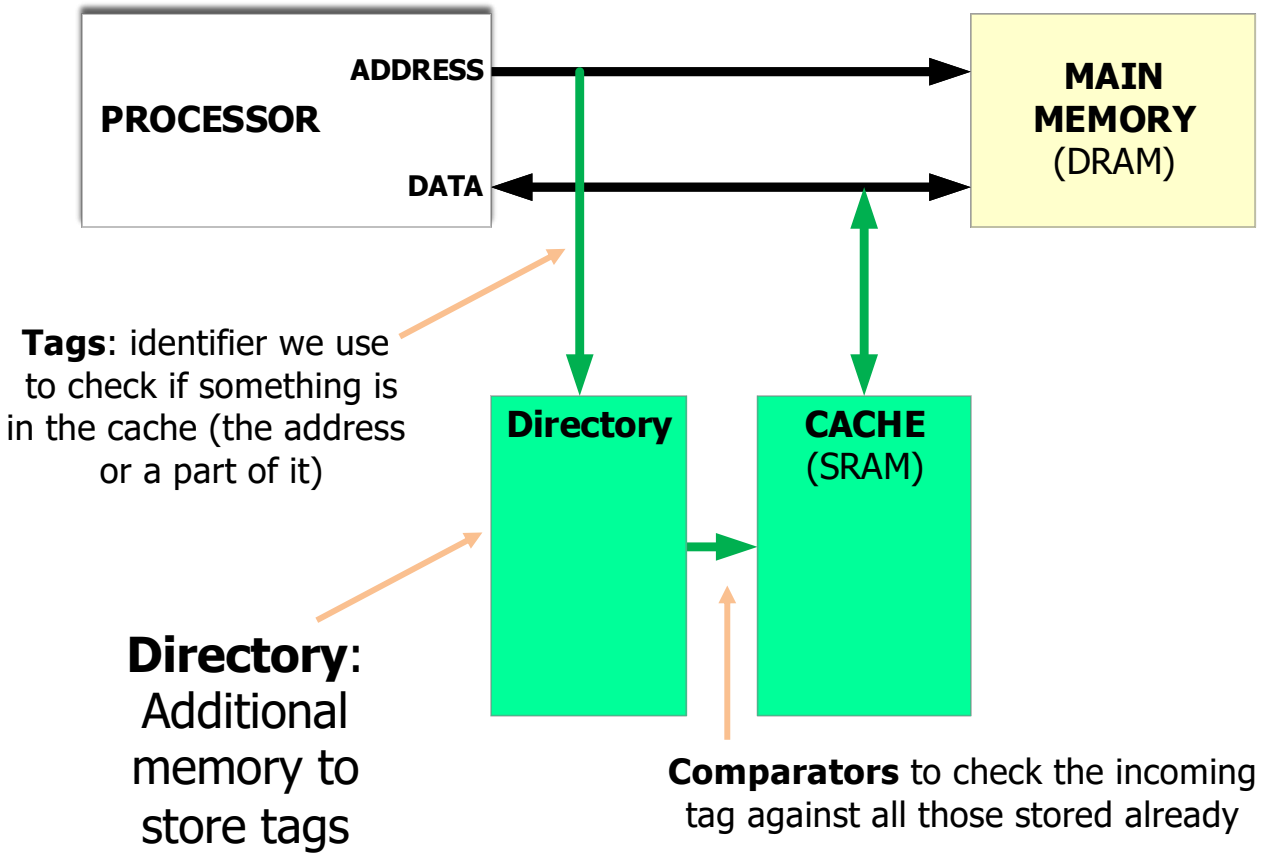
- Must be **invisible to the programmer**
  - One could analyse data structures and program semantics to detect frequently used variables/arrays and thus decide their placement, but this is not for us
- Must be **extremely simple and fast**
  - Goal is to access data in the fast memory in the order of a ns or below: there is obviously not much time to make complex decisions…

How is the Hierarchy Managed?

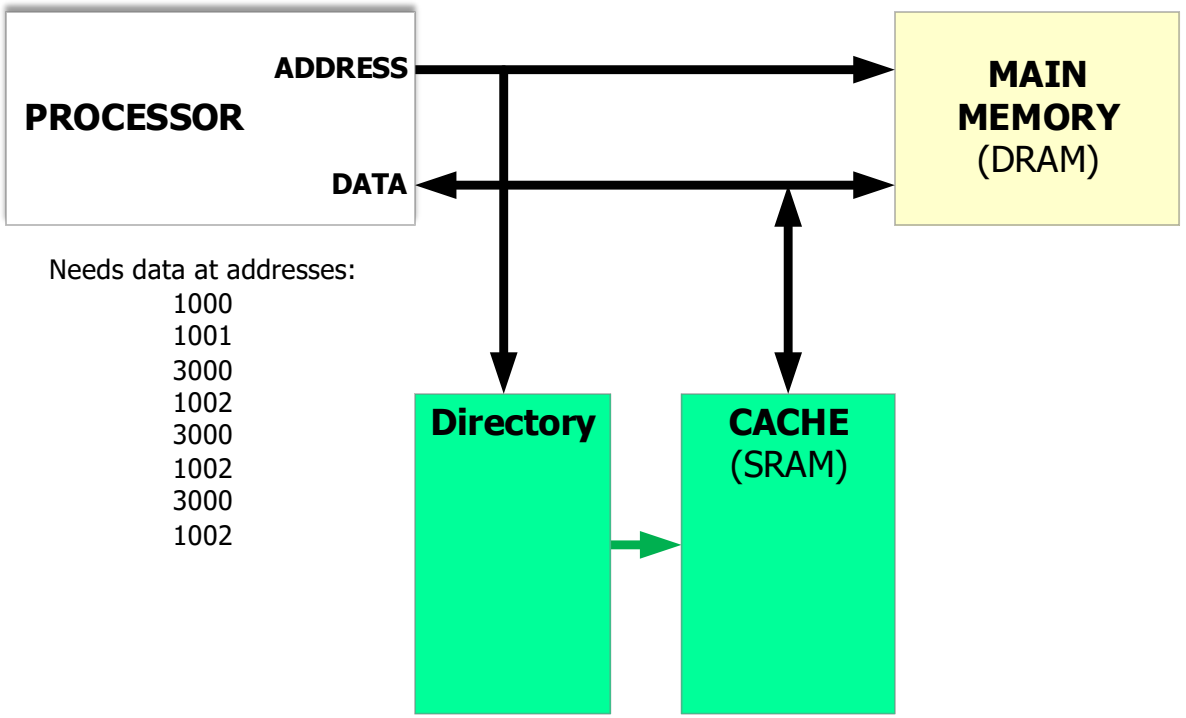
- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware
- main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware (TLB)
  - By the programmer (files)

Basic Cache Design

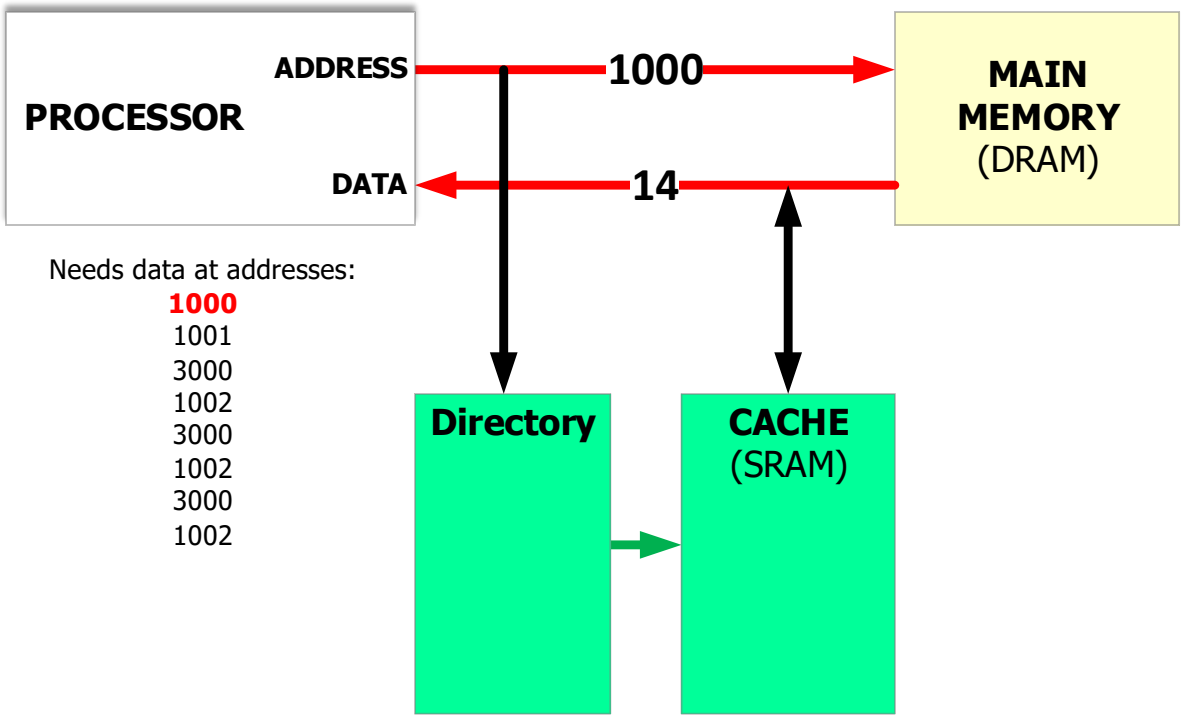
Cache Example



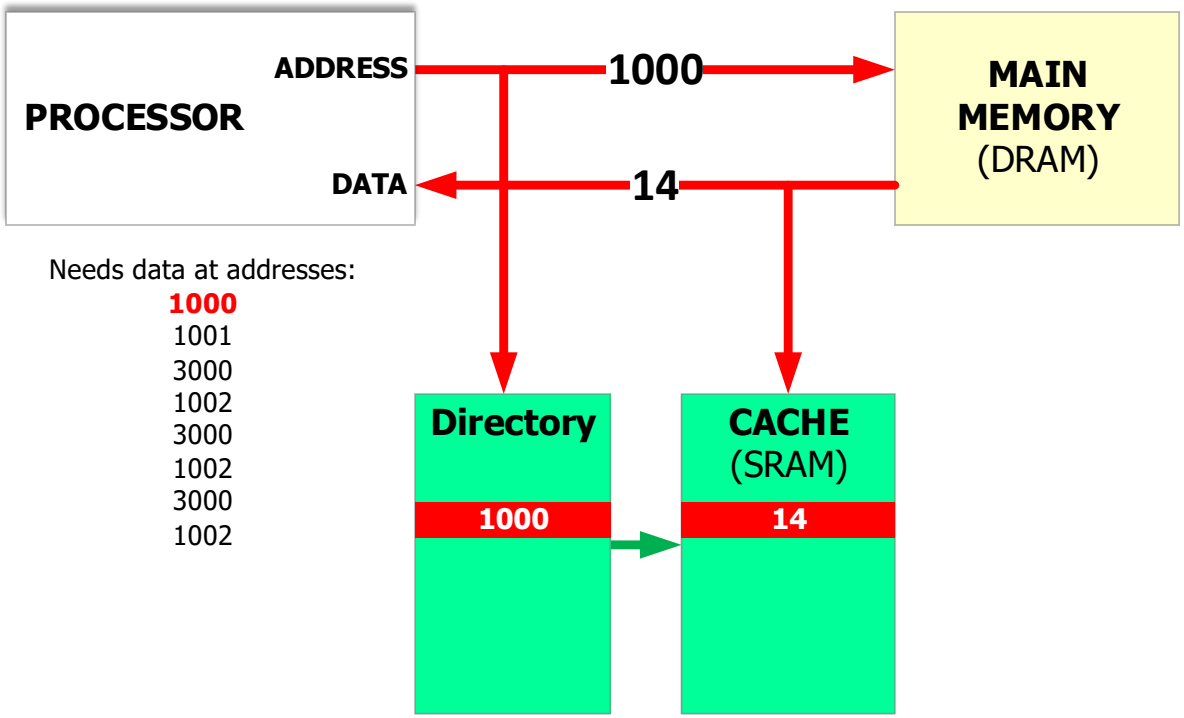
Cache, Example:



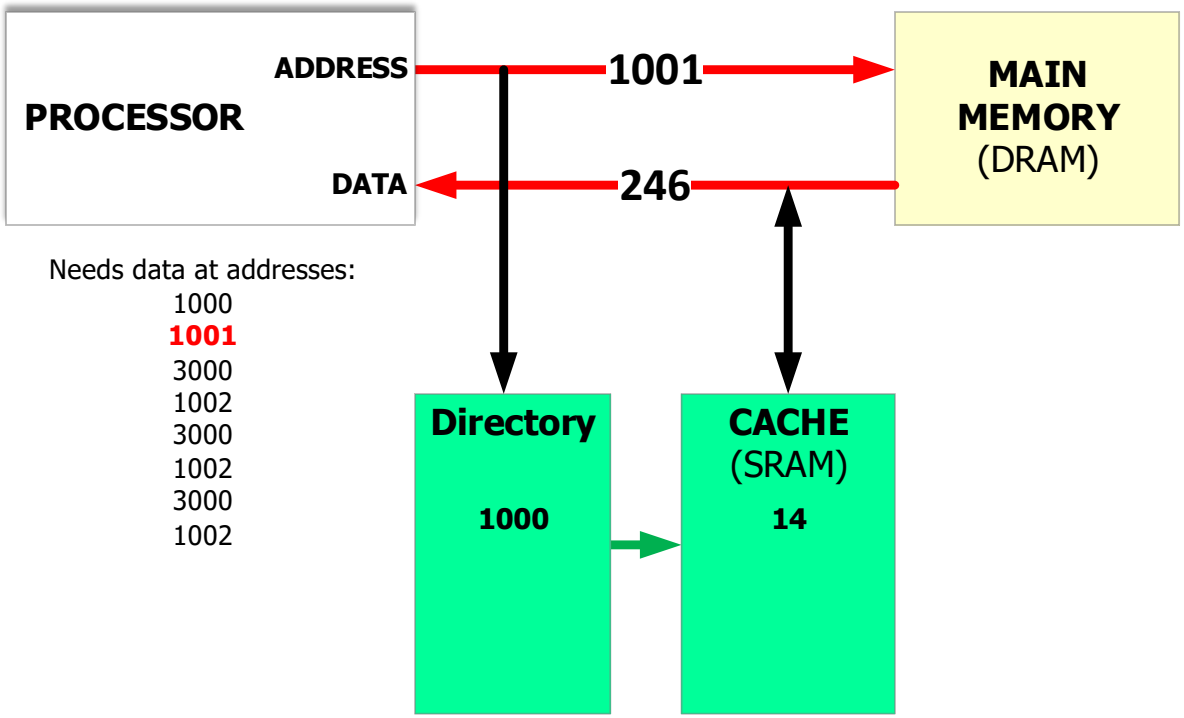
Cache, Example: (Cont'd)



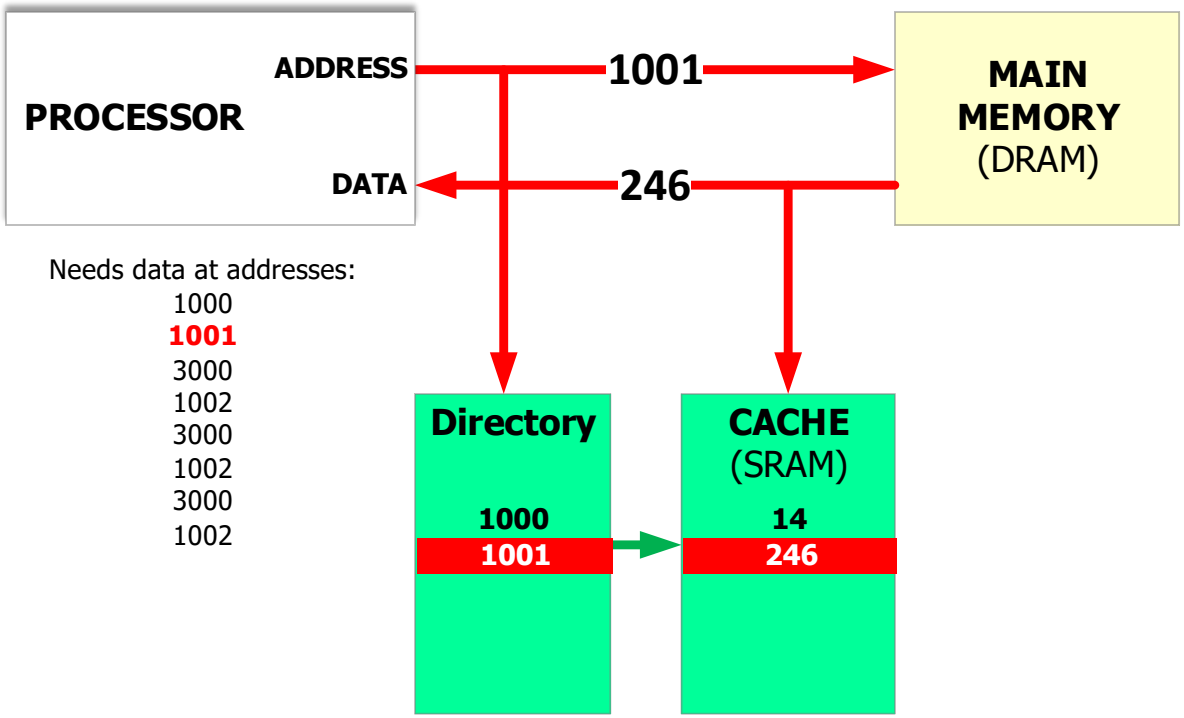
Cache, Example: (Cont'd)



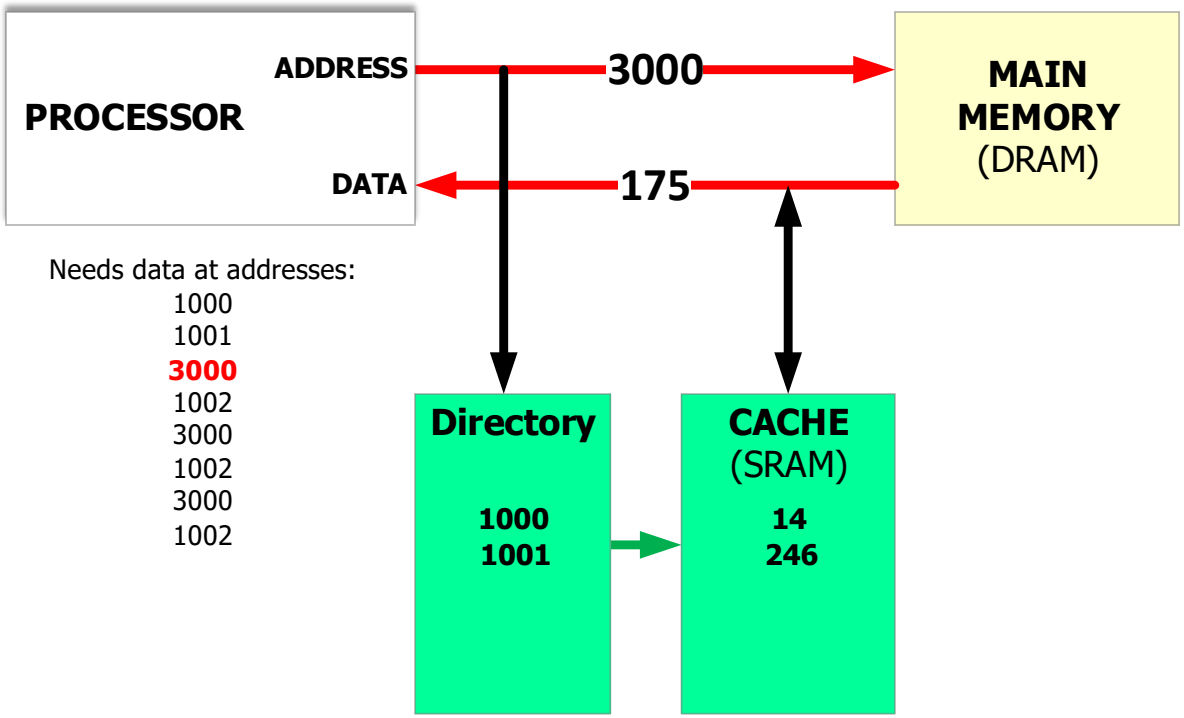
Cache, Example: (Cont'd)



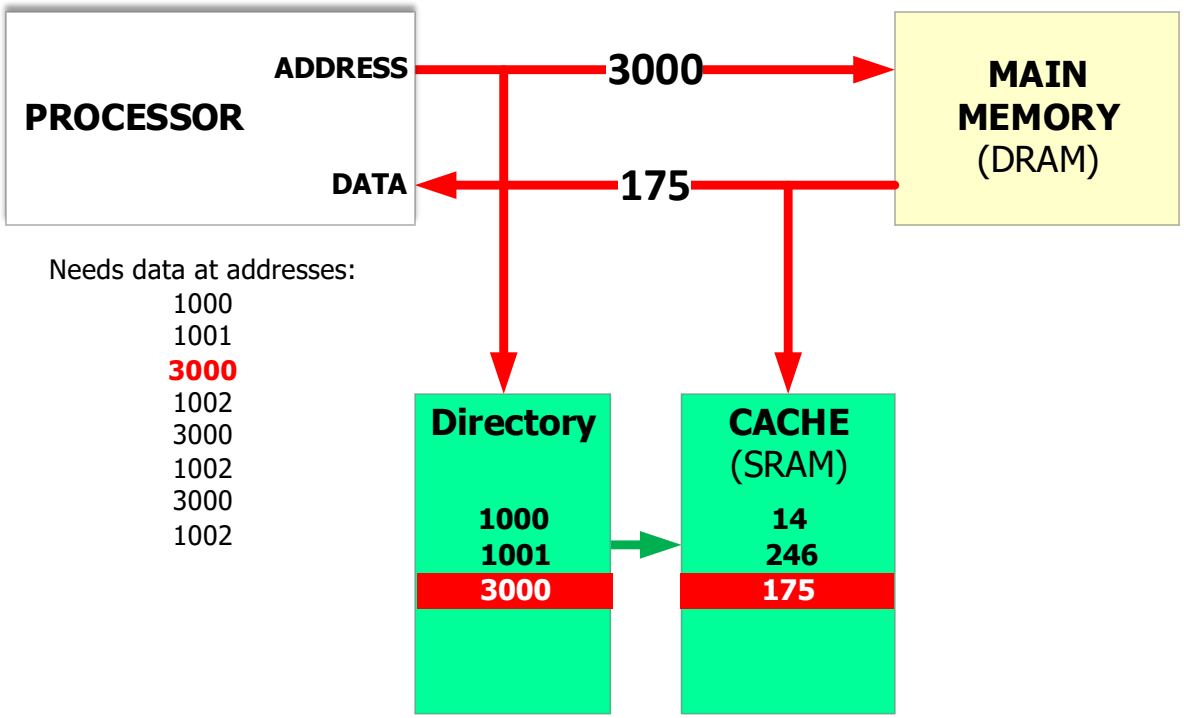
Cache, Example: (Cont'd)



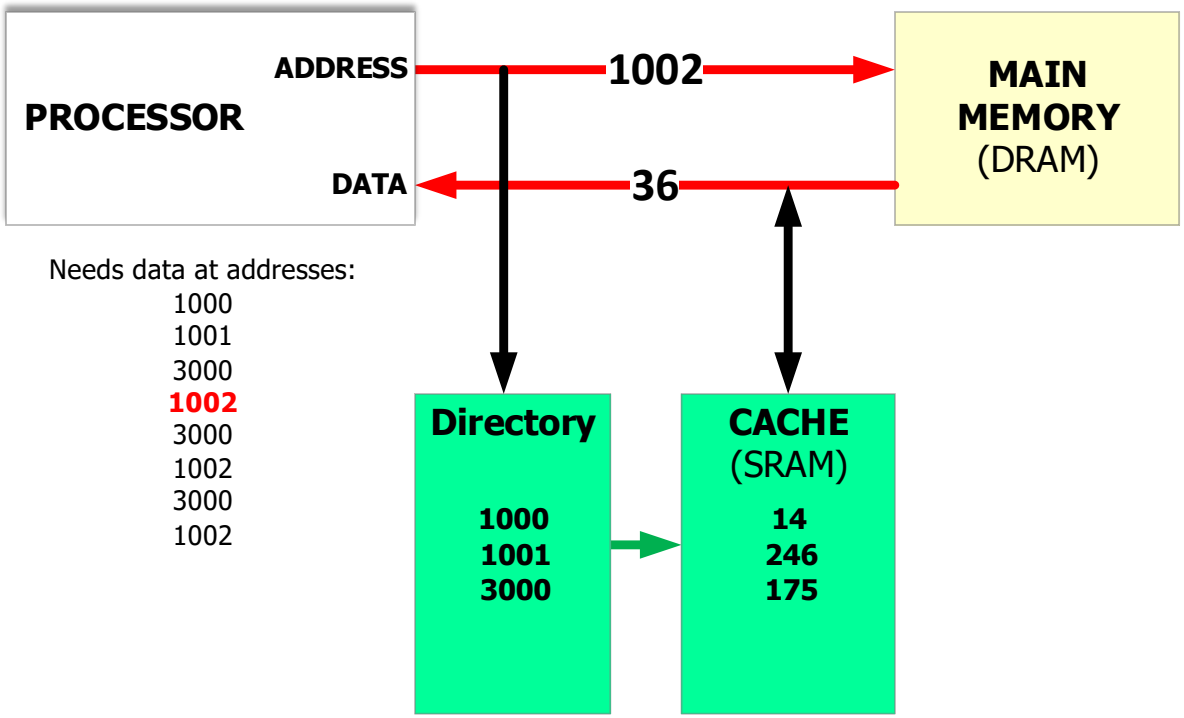
Cache, Example: (Cont'd)



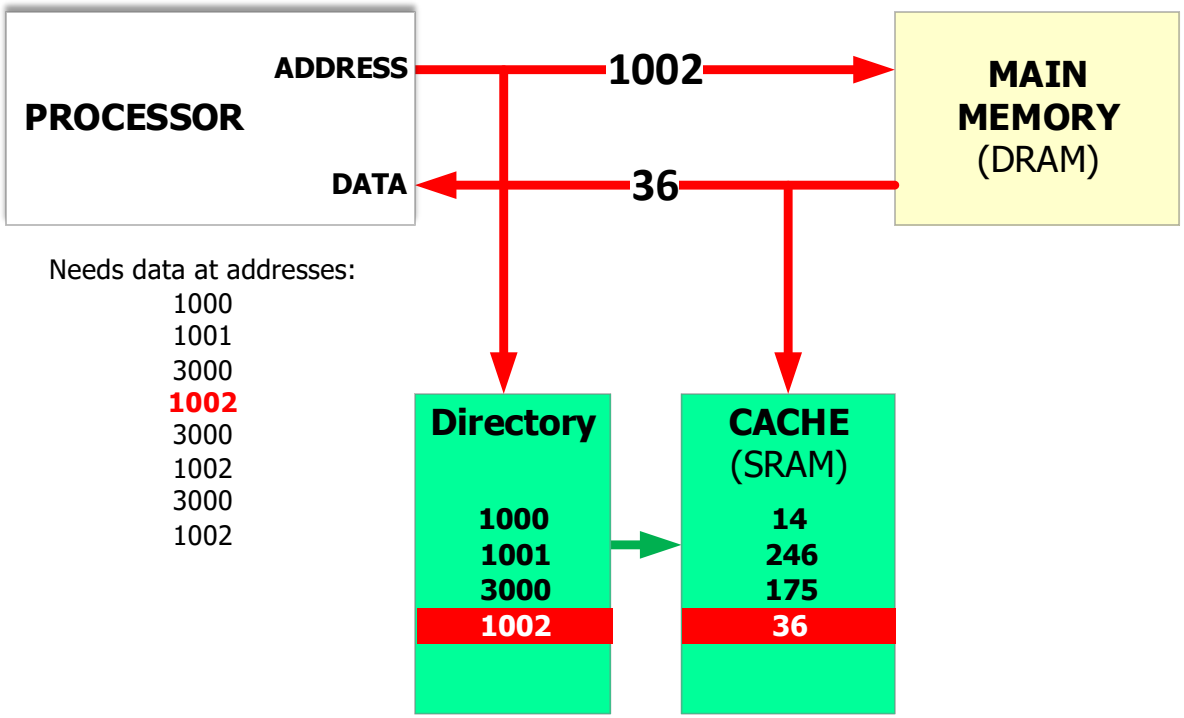
Cache, Example: (Cont'd)



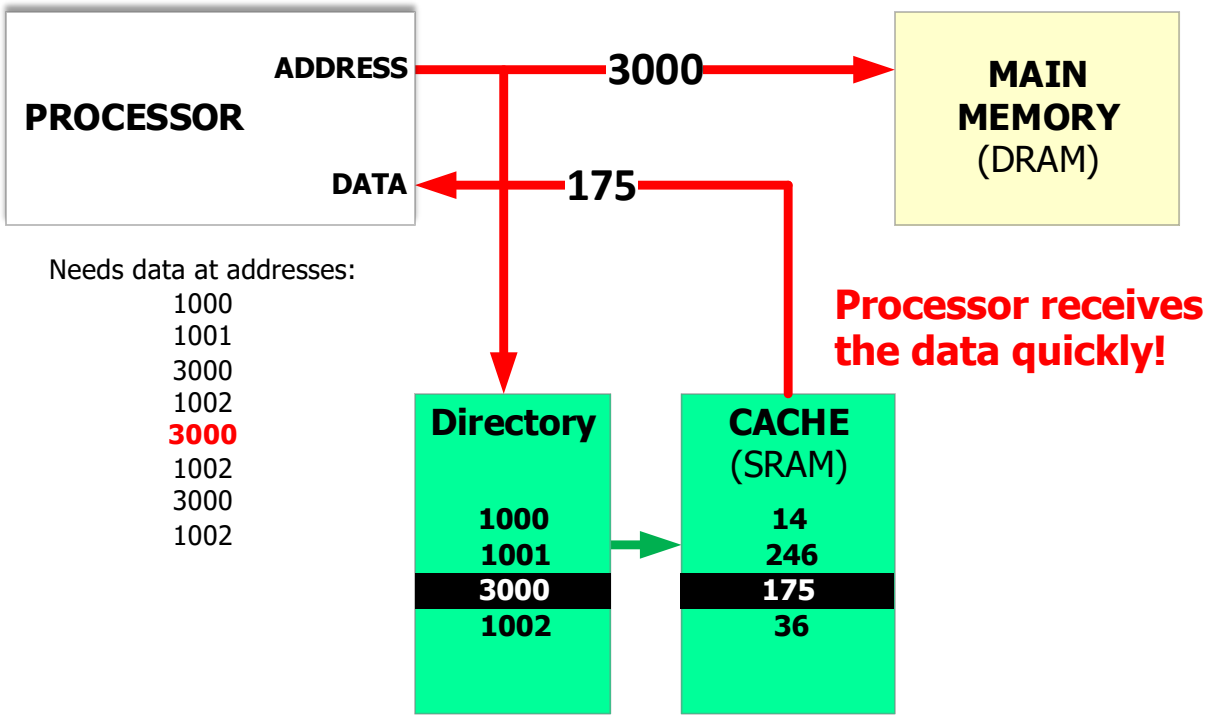
Cache, Example: (Cont'd)



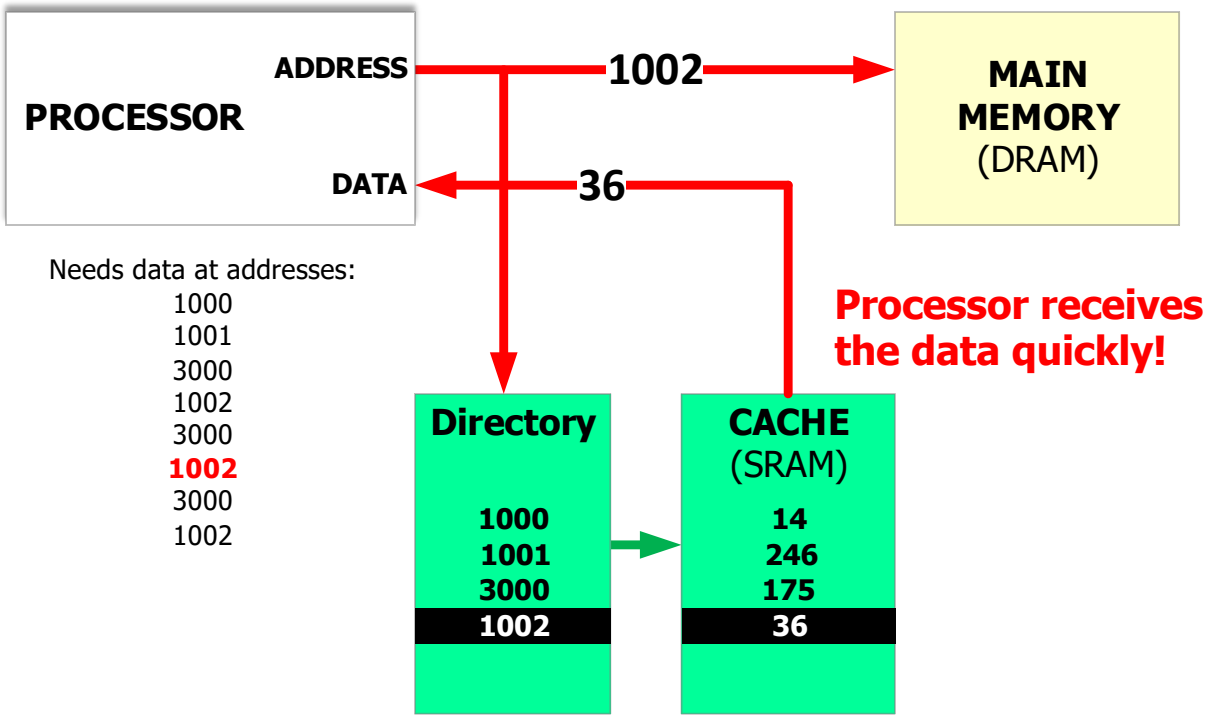
Cache, Example: (Cont'd)



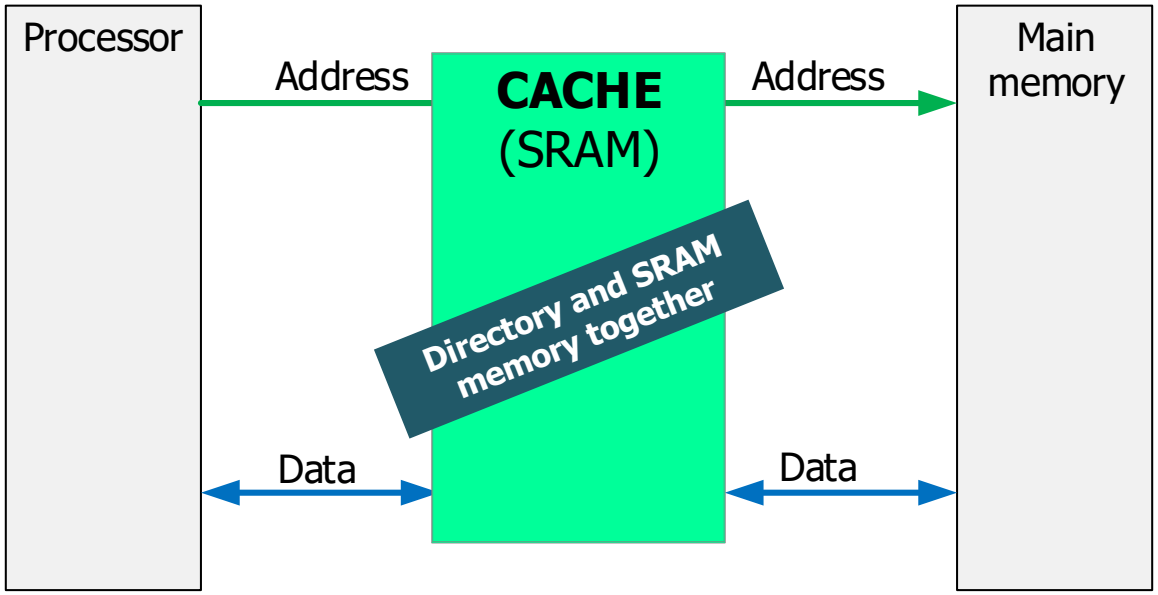
Cache, Example: (Cont'd)



Cache, Example: (Cont'd)



Cache is on the Data Path between Processor and Main Memory

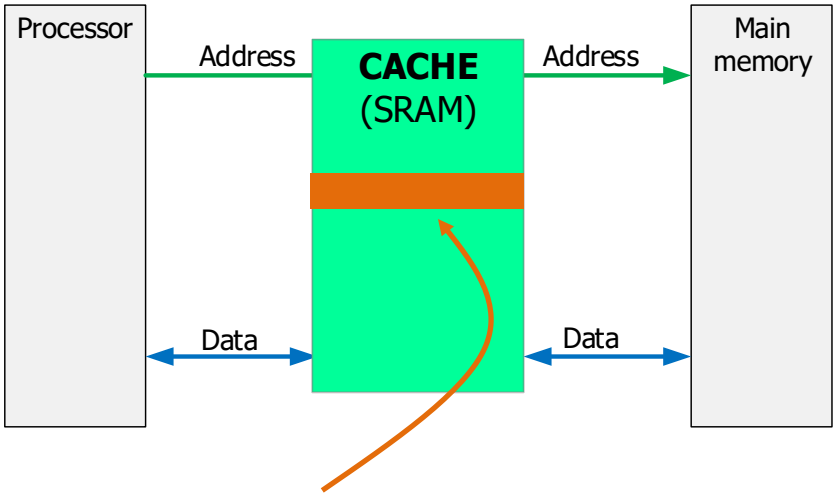


Basic Cache Design

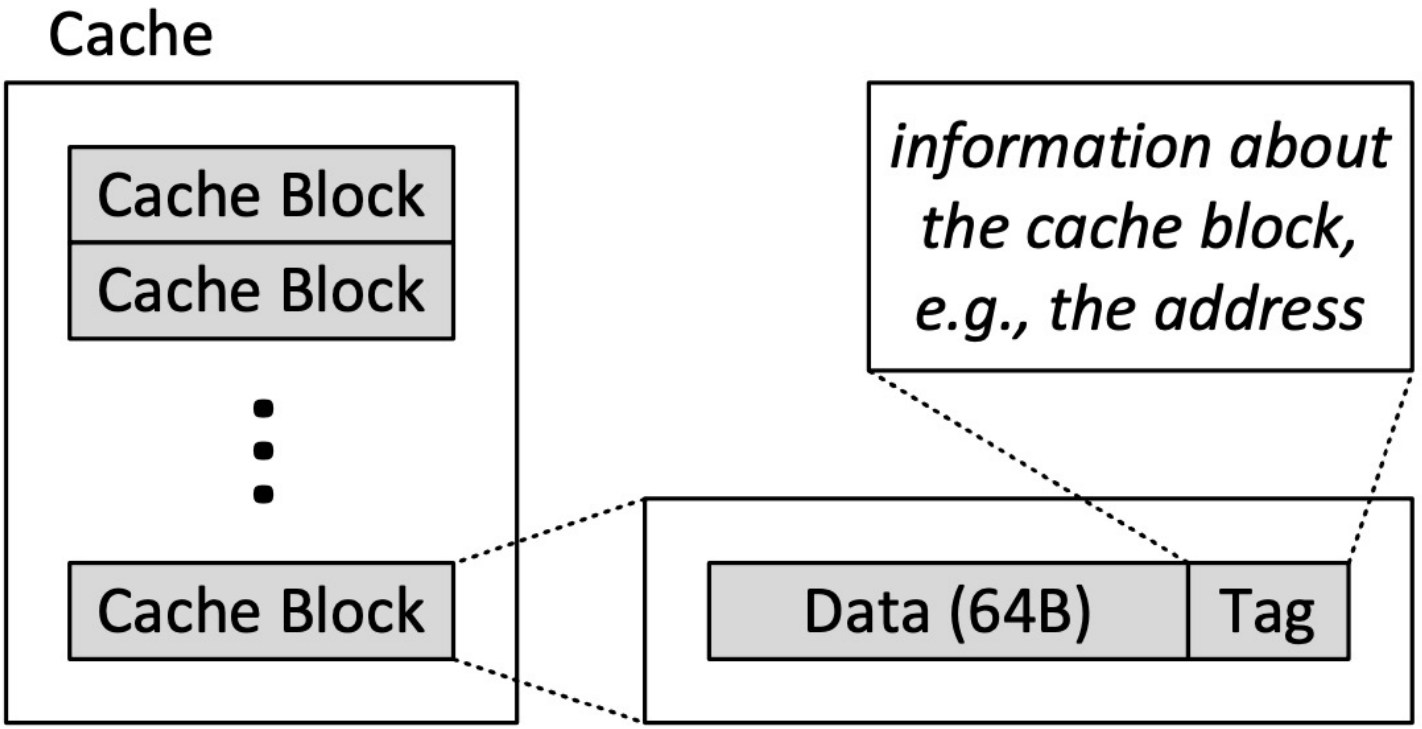
- Cache only holds a part of a program. Which part?
  - Cache holds **most recently accessed data/instructions**
- Cache is divided into **units** called **cache blocks** (or **cache "lines"**)
- How does the CPU know which part of the program the cache is holding?
  - Each cache block has **extra bits**, called **cache tag**, which hold the main memory address (or only a part of it) of the data that is currently in the cache block

Basic Cache Design (Cont'd)

- Cache is a fast (SRAM) memory of limited capacity, between the processor and the memory (DRAM)
- One cache **line** is a pair of one directory entry (**tag**) and data
- Finding a piece of data in cache requires finding whether the corresponding tag is already present in the cache



Basic Cache Design (Cont'd)



From Kim & Mutlu, "Memory Systems," Computing Handbook, 2014
https://people.inf.ethz.ch/omutlu/pub/memory-systems-introduction_computing-handbook14.pdf

Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into blocks that map to potential locations in the cache
- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - May have to evict some other block
- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Miss rate = (# misses) / (# accesses)
- Average memory access time (AMAT)

  = (hit-rate * hit-latency) + (miss-rate * miss-latency)

Memory Performance

- **Hit:** is found in that level of memory hierarchy
- **Miss:** is not found (must go to next level)

Hit Rate = # hits / # memory accesses
= 1 – Miss Rate

Miss Rate = # misses / # memory accesses
= 1 – Hit Rate

- **Average memory access time (AMAT):** average time it takes for processor to access data

AMAT = $t_{\text{cache}} + MR_{\text{cache}}[t_{MM} + MR_{MM}(t_{VM})]$

Memory Performance Example 1

- A program has 2,000 load and store instructions
- 1,250 of these data values found in cache
- The rest are supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Memory Performance Example 1

- A program has 2,000 load and store instructions
- 1,250 of these data values found in cache
- The rest are supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Hit Rate = 1250/2000 = **0.625**

Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$ cycle, $t_{MM} = 100$ cycles
- **What is the AMAT of the program from Example 1?**

Memory Performance Example 2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$ cycle, $t_{MM} = 100$ cycles
- **What is the AMAT of the program from Example 1?**

AMAT $= t_{\text{cache}} + MR_{\text{cache}}(t_{MM})$

$= [1 + 0.375(100)]$ cycles

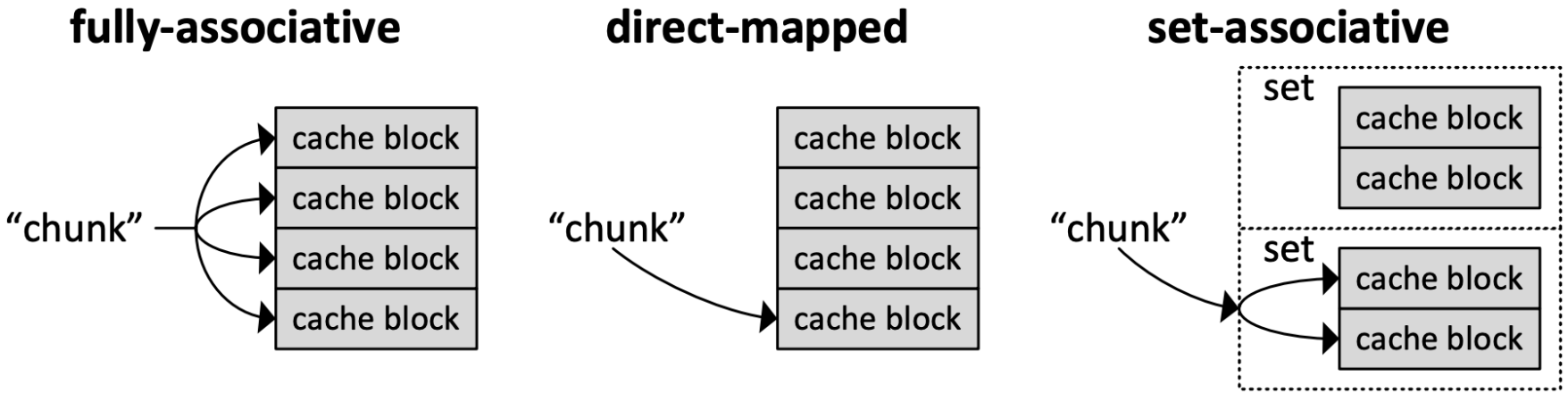
$=$ **38.5 cycles**

Caching Basics (Cont'd)

- Some important cache design decision
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?

Logical Organization of a Cache

- A key question: How to map chunks of the main memory address space to blocks in the cache?
  - Which location in cache can a given "main memory chunk" be placed in?

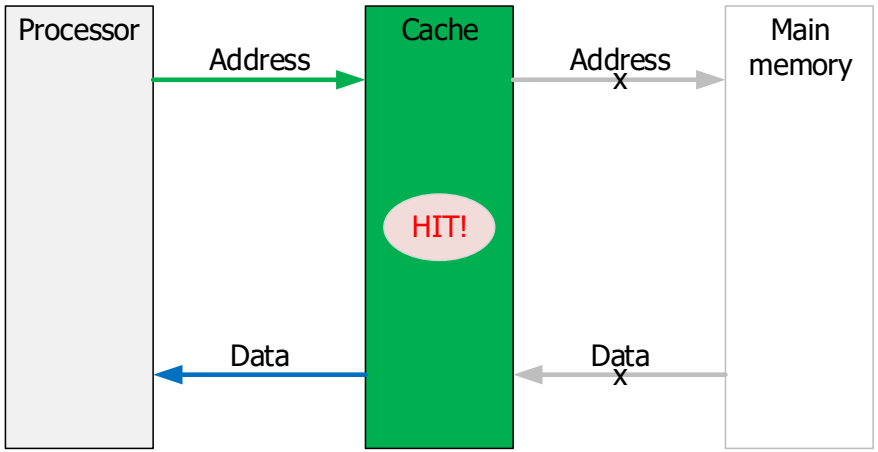


From Kim & Mutlu, "Memory Systems," Computing Handbook, 2014

Strategies on READ hit and miss

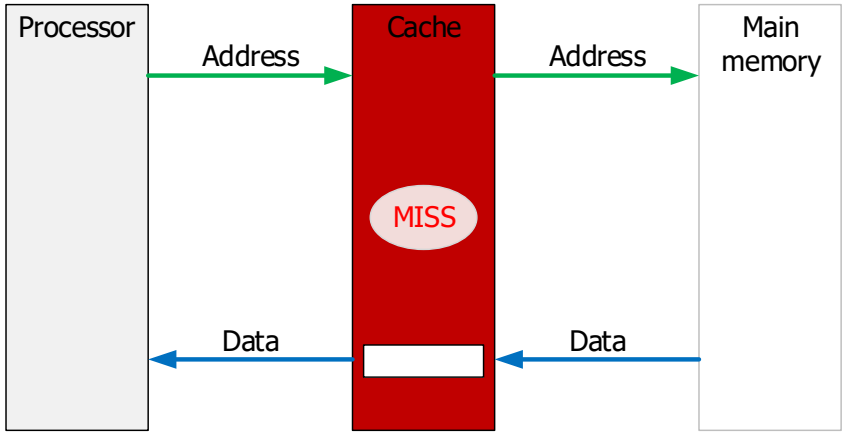
Strategy on a Read HIT

- When processor requests a memory read and cache detects that the requested data is already in the cache (**hit**):
  - The main memory is not accessed (no need for it)
  - The cache replies to the processor by sending the requested memory content
  - **Processor receives the data quickly!**



Strategy on a Read MISS

- When processor requests a memory read and cache detects that the requested data is **NOT** in the cache (**miss**):
  - The cache accesses main memory to get the data
  - The data is stored (allocated) in the cache **and** then passed to the processor
  - Processor gets the data, but after a longer waiting time



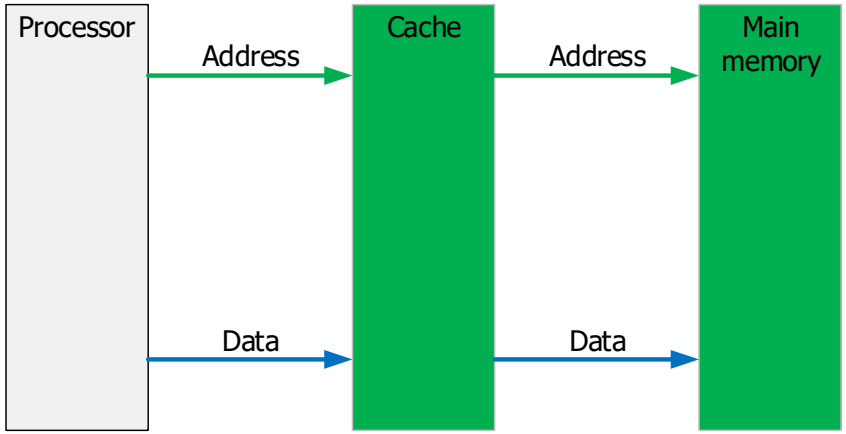
What if there is no space in the cache for the incoming data?

Strategies on WRITE hit and miss:

- **Write-through**
- **Write-back**
- **Write allocate**
- **Write no-allocate**

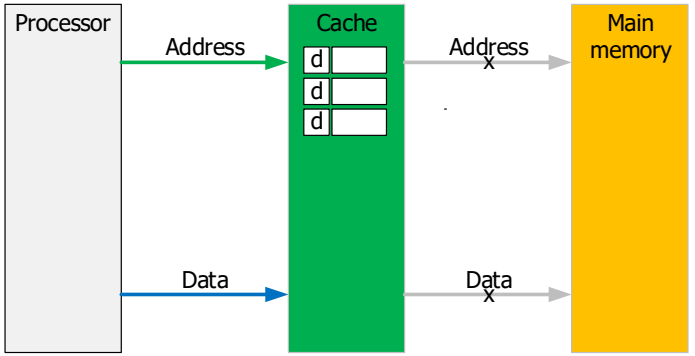
Strategies on a Write HIT: Write-Through Cache

- On a write hit, data is written both into the cache and into main memory:
  - Straightforward solution
  - May keep the memory/buses busy for nothing
  - Good for applications that write and then re-read data frequently as data is stored in the cache and results in low read latency



Strategies on a Write HIT: Write-Back Cache

- On a write, data is updated only in the cache (main memory data will eventually become wrong/obsolete)
  - Good sides: good for write-intensive applications
  - We need to remember which blocks are obsolete in memory. For this purpose, caches have an additional bit: **dirty bit**. It is set if the block is obsolete in memory.
  - Before overwriting a "dirty" block from the cache, its content must be stored to memory.



Strategies on a Write HIT: Write-Back Cache (Cont'd)

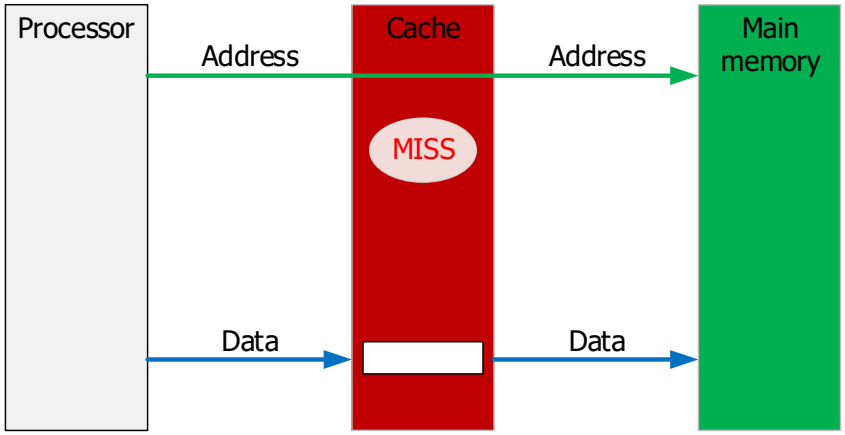
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Strategies on a Write MISS

- Should we cache a block if it is a write-miss?
  - Maybe it will not be read again soon
  - Maybe it will replace a block that would be read again soon
  - Two possible strategies:
    - Write allocate
    - Write no-allocate

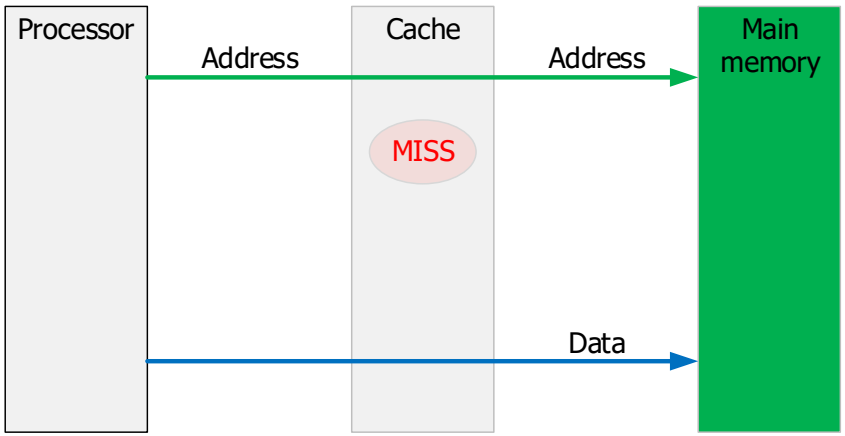
Strategies on a Write MISS: Write Allocate

- On a write miss, allocate the data block in the cache
- Then, signal a write hit
- Write-back caches generally use write allocate



Strategies on a Write MISS: Write No-Allocate

- On a write miss, write the data to memory, do not allocate in the cache
- Wait the next read miss to load the data and allocate in the cache
- Write-through caches generally use write no-allocate



Eviction Policies

Eviction Policies

- When there is no space for a new piece of data, we must **overwrite one of the cache lines**. This is called **eviction** or **replacement**.

- Policies (strategies) on how to decide what to evict:
  - **Least Recently Used** (LRU)
    - We overwrite the cache line which has not been accessed for the longest period of time
  - **Random**
    - The name says it all…
  - Others…more later.

Causes for Cache Misses: 3C

- *Compulsory:*

  first-reference to a block *a.k.a.* cold start misses

  - misses that would occur even with infinite cache

- *Capacity:*

  cache is too small to hold all the data the program needs

  - misses that would occur even under perfect placement & replacement policy

- *Conflict:*

  misses from collisions due to block-placement strategy

  - misses that would not occur with full associativity

Cache Types

- Two extreme cases:
  - **Fully-associative cache**
  - **Direct-mapped cache**
- Something in between
  - **Set-associative cache**

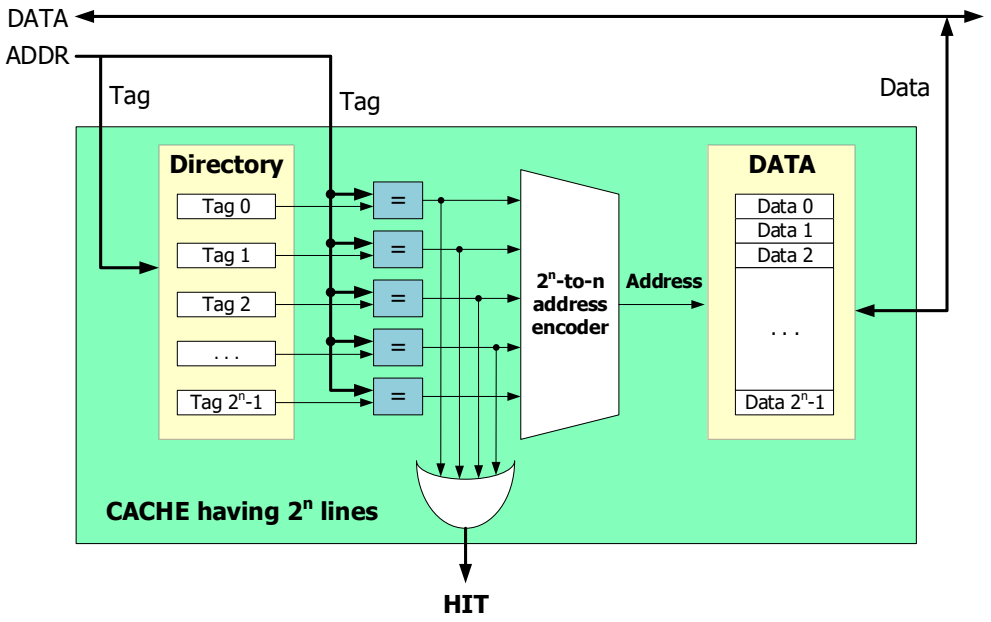
Fully-associative cache

Associativity of Caches

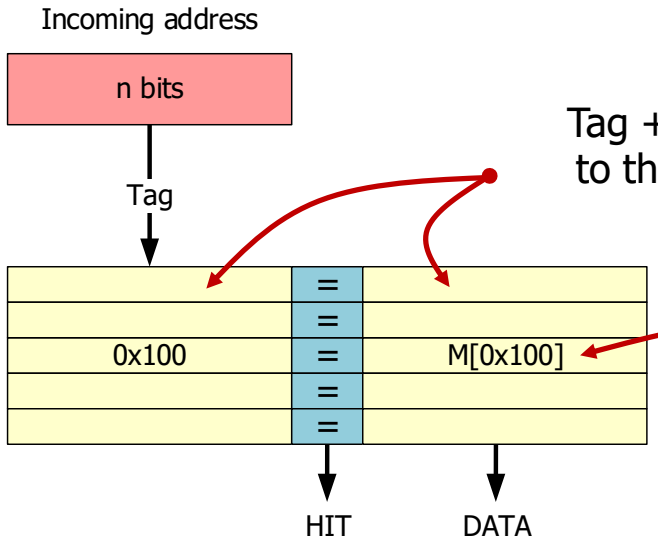
- **Associativity** indicates the number of cache lines where one word of data can be placed:
  - **Fully-associative**: a word can go in whichever line of the cache, hence the "**full**" associativity
    - If we wish to be precise, the associativity of a fully associative cache with **L** lines is exactly **L**

Fully-Associative Cache

- Data/instructions can be cached in **any available** cache line
- When checking whether the data is already in the cache (hit/miss), the incoming tag needs to be compared **(in parallel!)** with all existing tags in the directory
  - Causing a huge logic overhead for all the comparators needed!



Fully Associative Cache: Compact Representation



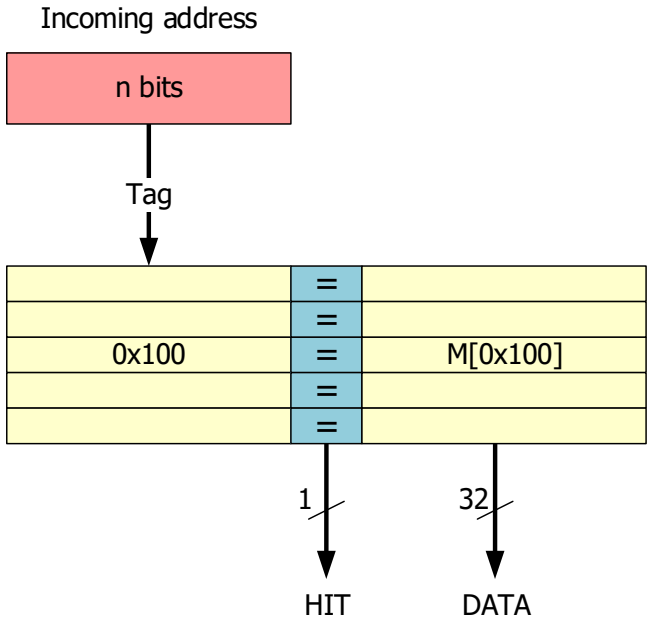
Tag + data/instruction corresponding to that tag make one **line of cache**

M[0x100] = memory content (data/instruction) at memory address 0x100

Temporal vs. Spatial Locality

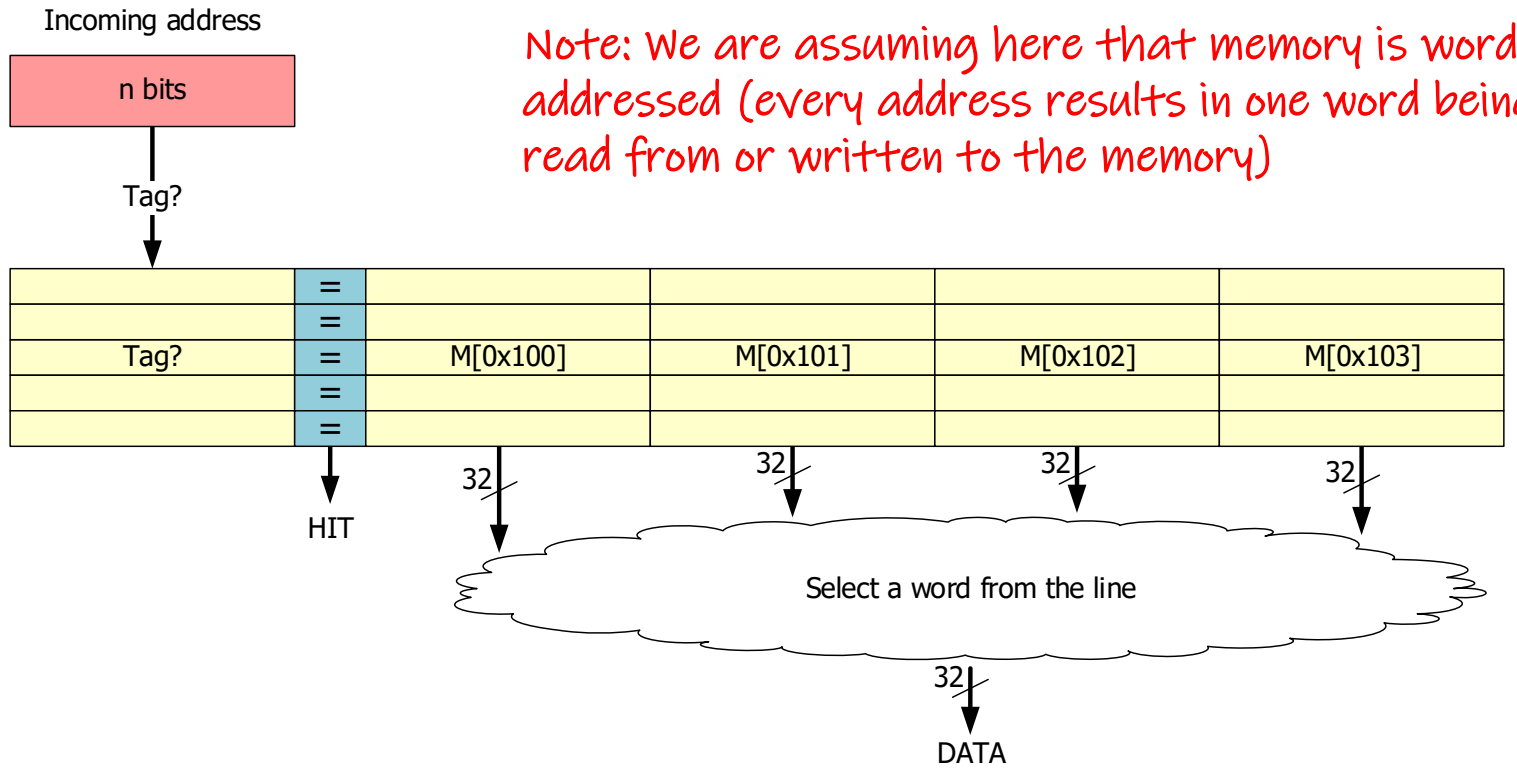
One Word Per Line Allows Exploiting Temporal Locality Only

- Temporal locality:
  - We load from main memory to the cache **exclusively the data required and when required**, hoping to use them again and benefit from having stored them in the cache
  - We are **not** bringing in any data in advance



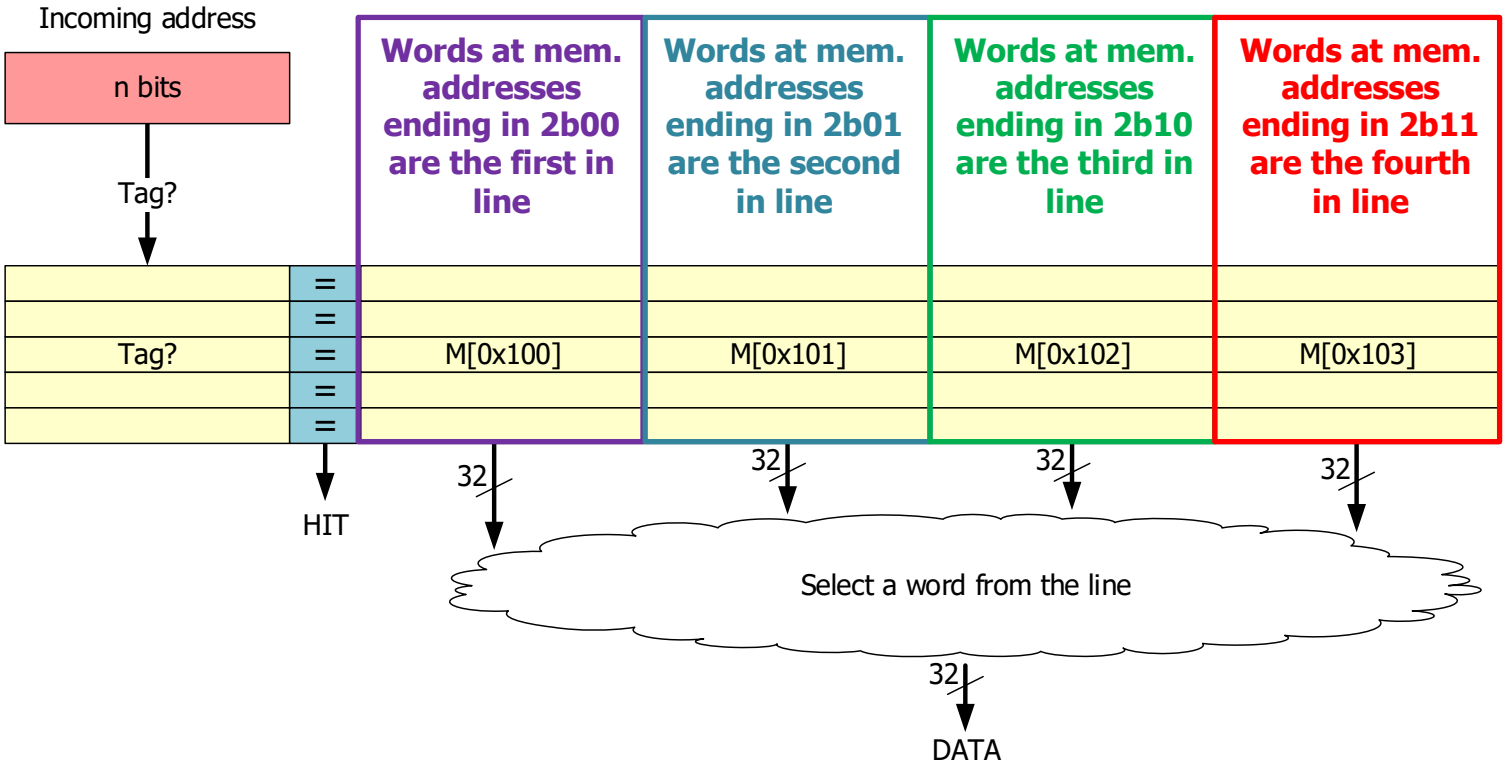
Exploiting also Spatial Locality

- Load from the memory and store in the cache more than a single word of data at a time (not only the required word but also the words that are near it)

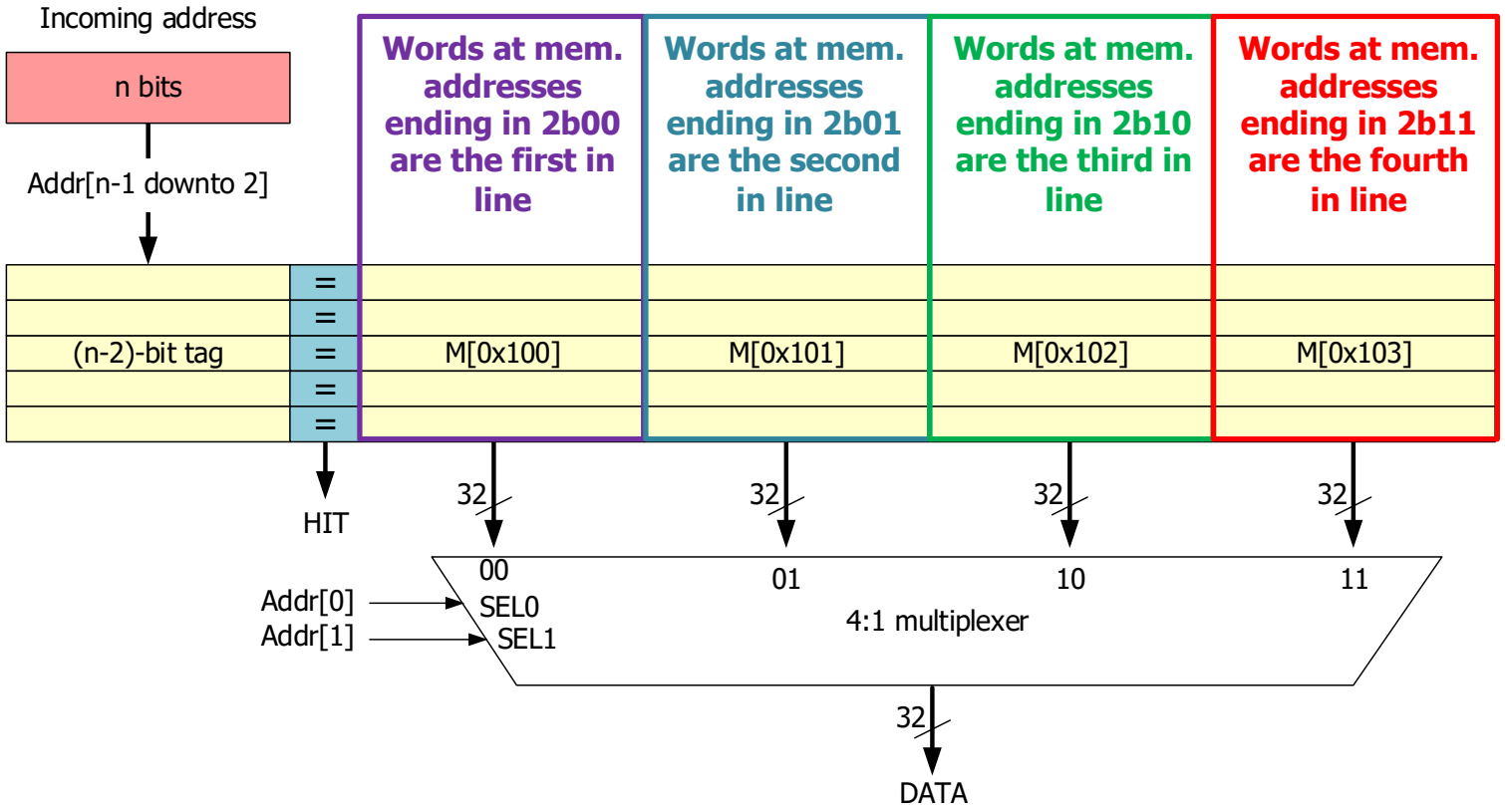


Note: We are assuming here that memory is word addressed (every address results in one word being read from or written to the memory)

Note: We are assuming here that memory is word addressed (every address results in one word being read from or written to the memory)



Note: We are assuming here that memory is word addressed (every address results in one word being read from or written to the memory)



Note: We are assuming here that memory is word addressed (every address results in one word being read from or written to the memory)

- In a more general case, if **2^m** words are loaded at a time and the memory address is **n**-bits wide, then
  - Last **m** bits of the address are used to **select a word** from the cache line
  - First **(n – m)** bits of the address form the **tag** of that cache line

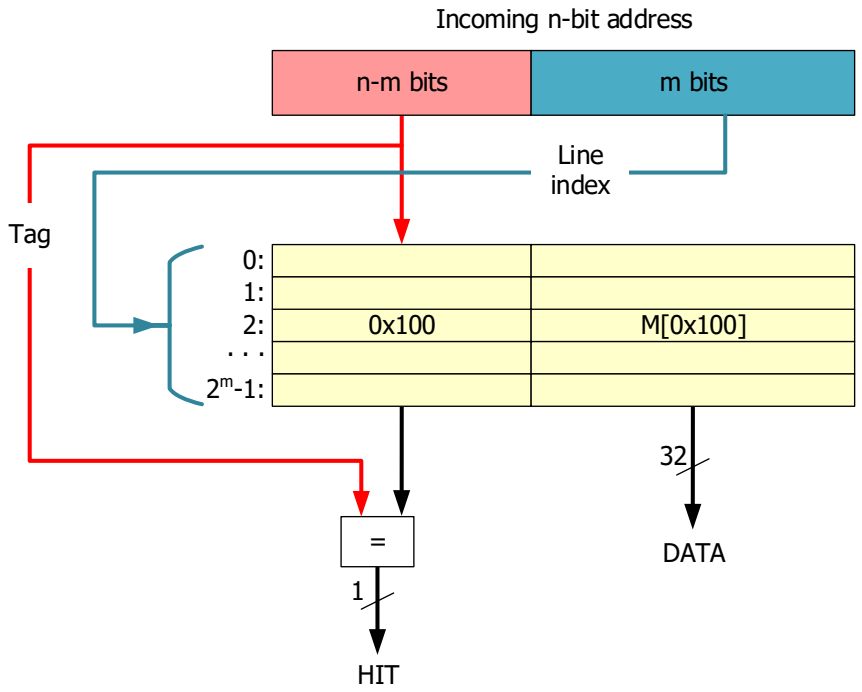
Can we do better?

- The cost of fully-associative caches is dominated by the comparators, making these cache memories feasible only if their capacity is very small

Direct-Mapped Cache

For a cache with capacity of 2^m words:

- **m** address bits are used to **index** the cache lines
- **(n – m)** address bits make a tag
- Each address **directly** maps to one (and only one) cache line
- Incoming tag needs to be **compared with only one** cache tag
- The whole cache is now behaving like standard memory



Example:
Fully-Associative vs Direct-Mapped Cache

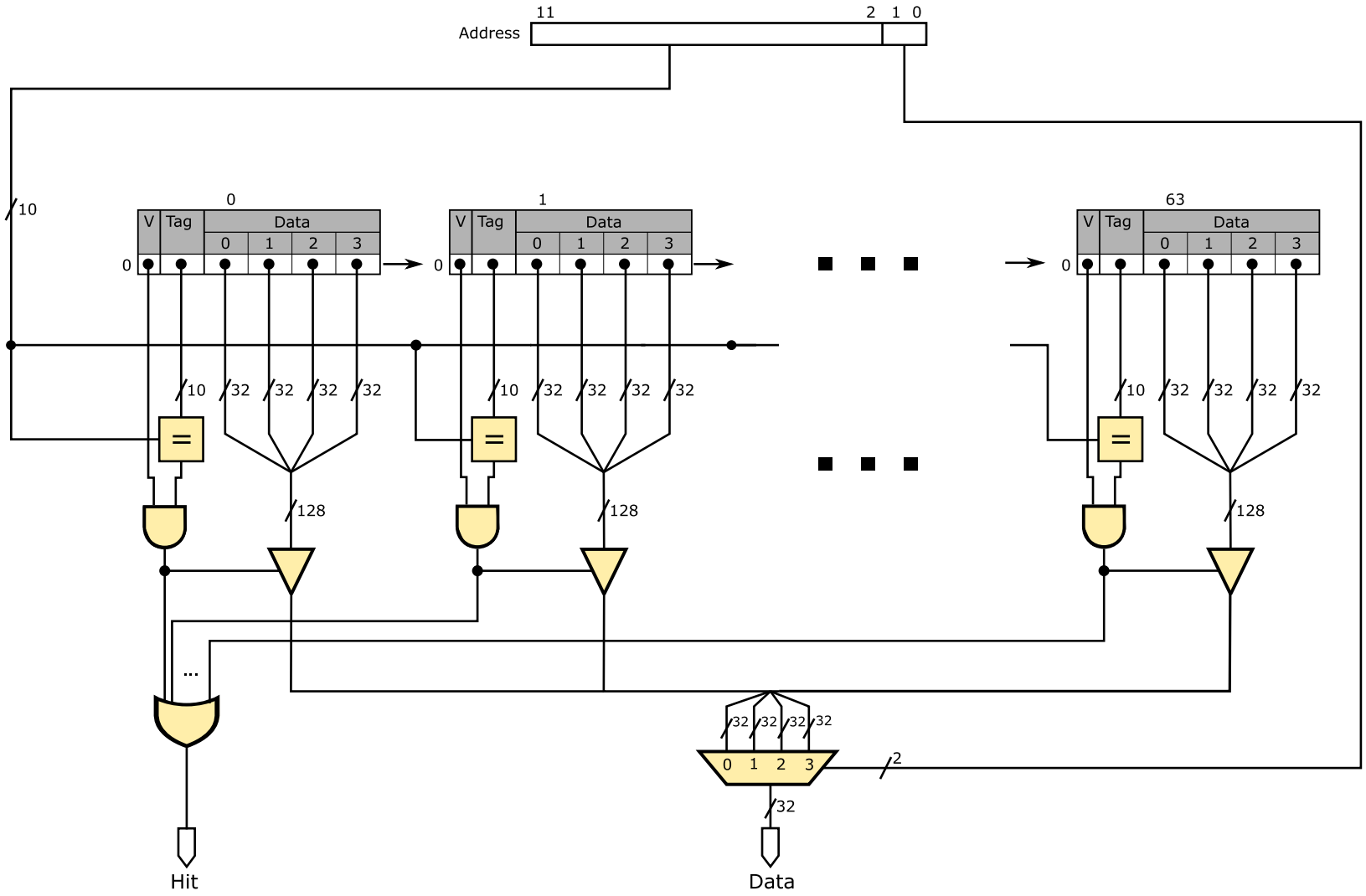
Example: Fully-Associative vs Direct-Mapped Cache

- Consider a fully-associative and direct-mapped cache, both with
  - 64 cache lines and
  - four **words** per line (= 256 **words** per cache, in total), where a word is 32-bit long
- Suppose memory accesses at 0x100, 0x101, 0x200, 0x102, 0x300, 0x103, 0x201, 0x102, 0x301, 0x103,…in that order.
- Memory is word addressed, addresses are 12-bit long
- Initially, both cache memories are empty. Which accesses result in **hits**? Which in **misses**?

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103
Fully Ass.	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?
Direct Mapp.	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?	Miss or Hit?

- **Fully-associative:**
  - The number of cache lines: $N_L = 64$
  - The number of words per line: $N_W = 4$
  - The number of address bits to select words inside a line: $\log_2(N_W) = 2$
  - Cache tag length: all the remaining address bits

Fully-Associative



Fully-Associative (1)

INITIAL STATE

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	0						
1	0						
2	0						
...	...						
63	0						

Compulsory miss

1. Access to M[0x100]

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	MISS
1	0						
2	0						
...	...						
63	0						

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Fully-Associative (2)

2. Access to M[0x101]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	0						
2	0						
...	...						
63	0						

Compulsory miss

3. Access to M[0x200]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	MISS
2	0						
...	...						
63	0						

Remember that in fully-associative cache words can be stored in ANY line, and not necessarily in the next available line.

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Fully-Associative (3)

4. Access to M[0x102]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	0						
...	...						
63	0						

Compulsory miss

5. Access to M[0x300]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	MISS
...	...						
63	0						

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Fully-Associative (4)

6. Access to M[0x103]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	
...	...						
63	0						

7. Access to M[0x201]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	HIT
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	
...	...						
63	0						

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Fully-Associative (5)

8. Access to M[0x102]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	
...	...						
63	0						

9. Access to M[0x301]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	HIT
...	...						
63	0						

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Fully-Associative (6)

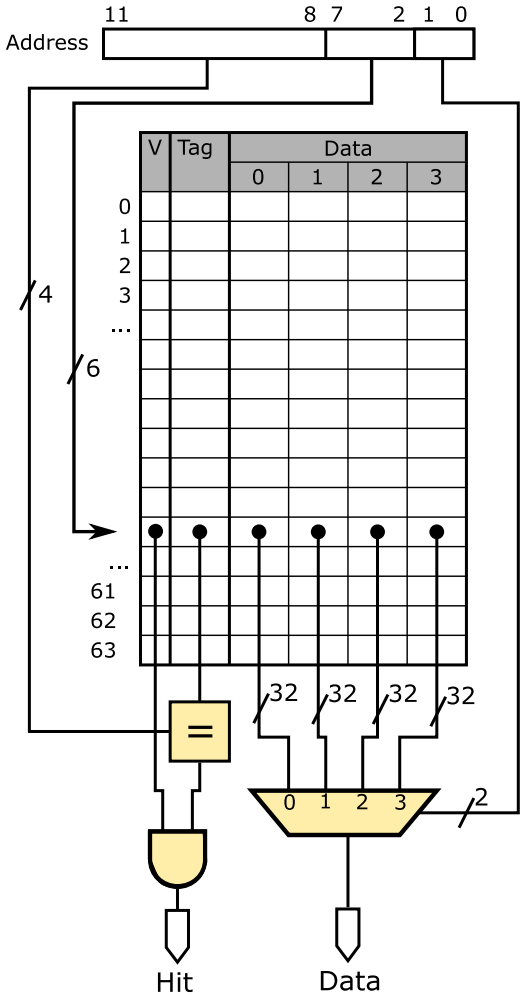
10. Access to M[0x103]

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	1	0x40	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	1	0x80	M[0x200]	M[0x201]	M[0x202]	M[0x203]	
2	1	0xC0	M[0x300]	M[0x301]	M[0x302]	M[0x303]	
...	...						
63	0						

	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103
Fully Ass.	M	H	M	H	M	H	H	H	H	H

Direct-Mapped

- **Direct-mapped:**
  - The number of cache lines: $N_L = 64$
  - The number of words per line: $N_w = 4$
  - The number of address bits to select words inside a line: $\log_2(N_W) = 2$
  - The number of address bits to index cache lines: $\log_2(N_L) = 6$
  - Cache tag length: all the remaining address bits = the upper 4 bits



Direct-Mapped (1)

INITIAL STATE

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	0						
1	0						
2	0						

Compulsory miss

1. Access to M[0x100]

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	1	0x1	M[0x100]	M[0x101]	M[0x102]	M[0x103]	MISS
1	0						
2	0						

2. Access to M[0x101]

Line number	Valid bit	Tag	Word 0 $A1A0 = 00_2$	Word 1 $A1A0 = 01_2$	Word 2 $A1A0 = 10_2$	Word 3 $A1A0 = 11_2$	HIT/MISS
0	1	0x1	M[0x100]	M[0x101]	M[0x102]	M[0x103]	HIT
1	0						
2	0						

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Direct-Mapped (2)

3. Access to M[0x200]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x2	M[0x200]	M[0x201]	M[0x202]	M[0x203]	MISS
1	0						
2	0						

Compulsory miss

4. Access to M[0x102]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x1	M[0x100]	M[0x101]	M[0x102]	M[0x103]	MISS
1	0						
2	0						

Conflict miss

5. Access to M[0x300]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x3	M[0x300]	M[0x301]	M[0x302]	M[0x303]	MISS
1	0						
2	0						

Compulsory miss

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Direct-Mapped (3)

6. Access to M[0x103]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x1	M[0x100]	M[0x101]	M[0x102]	M[0x103]	MISS
1	0						
2	0						

Conflict miss

7. Access to M[0x201]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x2	M[0x200]	M[0x201]	M[0x202]	M[0x203]	MISS
1	0						
2	0						

Conflict miss

8. Access to M[0x102]

Line number	Valid bit	Tag	Word 0 A1A0 = 00_2	Word 1 A1A0 = 01_2	Word 2 A1A0 = 10_2	Word 3 A1A0 = 11_2	HIT/MISS
0	1	0x1	M[0x100]	M[0x101]	M[0x102]	M[0x103]	MISS
1	0						
2	0						

Conflict miss

Addr.	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103

Conclusion

	0x100	0x101	0x200	0x102	0x300	0x103	0x201	0x102	0x301	0x103
Fully Ass.	M	H	M	H	M	H	H	H	H	H
Direct Mapp.	M	H	M	M	M	M	M	M	M	M

- Addresses 0x1…, 0x2…, and 0x3… use the same line of the direct-mapped cache (they are said to **alias**), resulting in frequent **cache pollution** or **conflict misses**
- Fully-associative cache is more flexible than direct-mapped, at a high cost

$$HitRate\,(\text{FullyAssociative}) = \frac{7}{10} = 70\%$$

$$HitRate\,(\text{DirectMapped}) = \frac{1}{10} = 10\%$$

Can we think of an intermediate solution?

Summary

- Memory Hierarchy
- Principal of Locality makes caches work
  - Spatial Locality
  - Temporal Locality
- Placement, Identification, Replacement, Write Policy
- Read Strategies
- Write Strategies
  - Writhe through Caches
  - Write back Caches
- Write miss cases
  - Write Allocate
  - Write no-Allocate
- Eviction Policies
  - LRU, Random, MRU, etc.
- Fully-Associative Caches
- Direct-Mapped Caches