Cache Basics

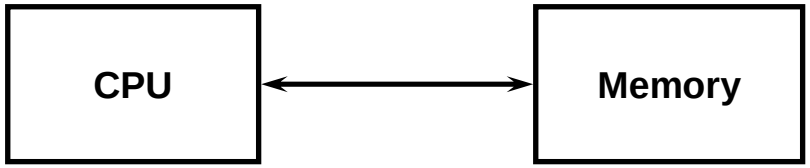
Trends

- **Memory consumption is growing**
  - **About 2X every 2 years**
  - **Typical size (1998): 32M-64M**
- **CPU speed grows faster than memory and buses**
  - **CPU:**
    - **486 25-66MHz**
    - **Pentium 66-233MHz**
    - **Pentium-II 200-450MH**
  - **Memory:**
    - **DRAM: 60-100ns (“10-16MHz”); Cost: <10$ per 1M**
    - **SRAM: <10ns (“100MHz”); Cost: ~30$ per 1M**
  - **Bus: 486 33MHz, Pentium 66MHZ, Pentium-II 66-100MHz**

Memory becomes the bottleneck

Slow ***or expensive***

Simple Cost Model

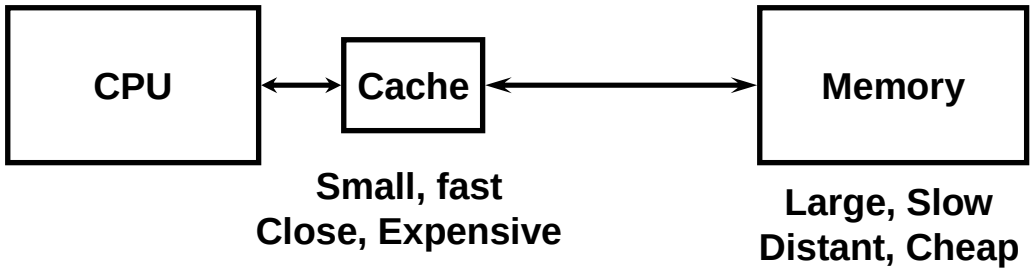


- Assume (for simplicity)
  - ~200 MHz CPU => 5ns cycle
  - DRAM: 60ns, 15 cycles (12 + 3 overhead)
  - SRAM: 10ns, 3 cycles (2 + 1 overhead)
- 1/3 of instructions access the memory, Out of 100
  - 67 take 1 clock
  - 33 take 3 or 15 depending on memory

Type	Cost of 1M	Cost of 32M	Memory Speed	Time for 100
SRAM	40$	40$*32M=1280$	3	67+33*3 = 166
DRAM	10$	10$*32M=320$	15	67+33*15=562

- Can get a significant performance improvement - but at cost!
- Ignored: instructions are also fetched from memory, bus bandwidth limitations

Solution



- Cache! A ***Small, Fast, Close*** memory
  - Serves as a buffer between CPU and main memory
- At any time it contains a copy of a portion of main memory
  - Small in size
  - Significant advantage
  - Dynamically changing
- Why does it work?
  Space and time locality!
  - Code is fetched sequentially (Space)
  - Code is re-executed (loops, procedures) (Time)
  - Access close or previous data (Space, Time)

  Definitely reminds of paging behavior!

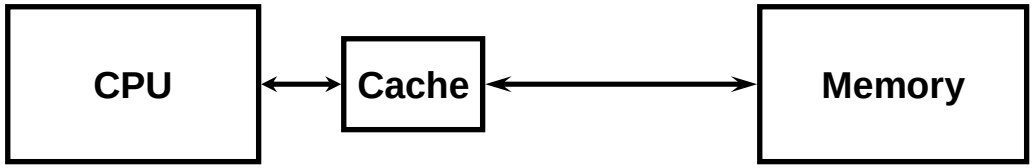
Analogy: refrigerator - keep your favorite drinks at home, no need to go to the supermarket every time

Terminology and Trends

- Hit Rate: percentage of accesses for which data is in the cache
- Miss Rate: 1-Hit_Rate
- Hit Rate depends on cache structure (size) and application
- Typical Hit rates
  - 90% for 16K Cache
  - 96% for 1/2M Cache
  - Subject to “law of diminishing return”...
- Performance - measured in *CPI*
  - Cycles Per Instruction = Core_Count*Core_Cycles + Mem_Count*Hit_rate*Hit_cycles + Mem_Count*Miss_Rate*Miss_cycles

    (e.g. 0.67*1+0.33*90%*3+0.33*10%*15 = 2.05)
- Application Trends
  - New applications are larger => reduce hit rate
  - Increased awareness of cache => Algorithms and code generators optimized for caches

Cache Cost Model

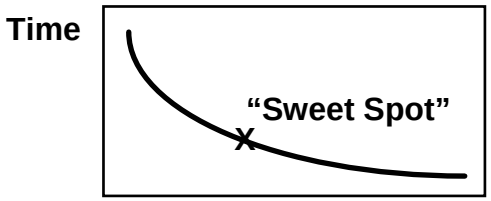


Small, fast
Close, Expensive

Large, Slow
Distant, Cheap

Type	Cost of 1M	Cost of 32M	Memory Speed	Time for 100
SRAM	40$	40$*32M=1280$	3	67+33*3 = ~166
DRAM	10$	10$*32M=320$	15	67+33*15=~562

Conf, Hit rate	**Cost**	**Memory Speed**	**Time for 100**
16K, 90%	**320$+ε =360$**	**3, 15**	**67+33*90%*3+33*10%*15=~206**
1/2M, 96%	**320$+20$+ε'=400$**	**3,15**	**67+33*96%*3+33*4%*15 = ~182**

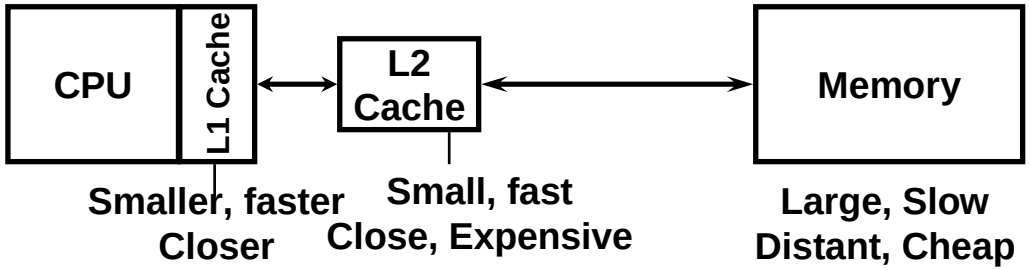


Can get most of the performance for a fraction of the cost!

In Practice...

- Cache is even more important since
  - It allows memory to cache accesses to be done in bursts
    - Eliminating DRAM page misses
- Faster caches are put on chip to allow super fast access
  - Allows latency and bandwidth which are impossible in any external memory
  - Split data and instruction cache to avoid trashing and increase bandwidth
  - Can be combined w/ external cache or w/o it
    - On-chip is called Level 1 (L1), off-chip is Level 2
- Typical configurations
  - 8KB-32KB on chip data and instruction cache
  - 256KB - 1MB off-chip cache
- Trend: Level-2 moves from external to on package to on die

On Chip / Off Chip Cache Cost Model



Type	Cost of 1M	Cost of 32M	Memory Speed	Time for 100
SRAM	40$	40$*32M=1280$	3	67+33*3 = ~166
DRAM	10$	10$*32M=320$	15	67+33*15=~562

Conf, Hit rate	**Cost**	**Memory Speed**	**Time for 100**
16K, 90%	320$+ε =360$	3, 15	67+33*90%*3+33*10%*15=~205
1/2M, 96%	320$+20$+ε'=400$	3,15	67+33*96%*3+33*4%*15 = ~182
16K, 90%	**on chip**	**1, 15**	**67+33*90%*1+33*10%*15=~146**
16K+1/2M	**+40$**	**1,3,15**	**67+33*90%*1+ 33*6%*3+33*4%*15=~122**

L1 hits = 33*90% = 30% of instructions; L2 accesses = 33-30 = 3
L2 hits = 3*60% = 1.8; L2 misses = 3-1.8 = 1.2 accesses to memory

- **Most of the benefit is from L1, but L2 still gives >15%**
  May be more in memory-hungry applications

Cache Challenges

- How do we know what is in the cache and what is not?
- When and how is cache content replaced?
- How to make sure everything is consistent?

Cache Structure

Main idea:

- **Lines contain a copy of consecutive bytes of memory (4-64 bytes)**
  # of lines = CacheSize/LineSize (e.g., Pentium: 16K/32B=512 lines)
- **Tags indicates the address which the line maps**
- **Coherency information: line status relative to memory**
- **Replacement hints**

- **Cache "organization"**
  Determines which memory portion can be mapped to each line
  - **A portion of memory can be mapped to only a subset of the lines in the cache. This subset is called "set"**
- **Options:**
  - **Direct Map**
  - **Set Associative**
  - **Fully Associative**
  - **Sectored**

Direct Map Cache

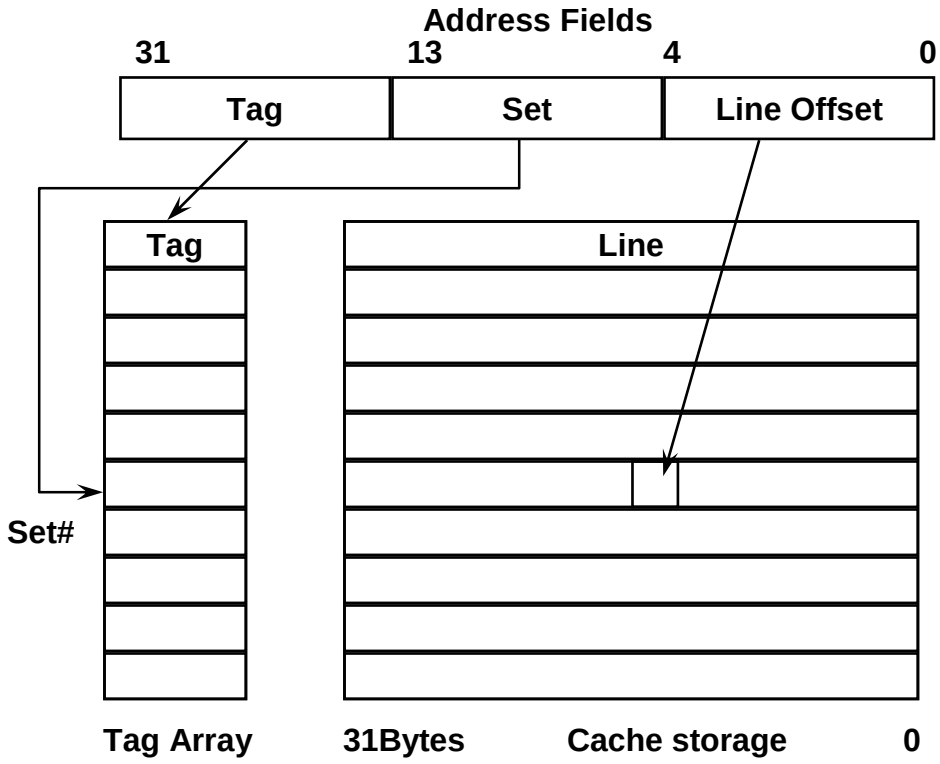
- Memory is “conceptually” divided into slices whose size is the cache size
- Offset from from slice start indicates position in cache (set)
- Addresses with the same offset map into the same line
- One tag per line is needed

Example:

Line Size:	32 bytes
Cache Size	16KB
# of lines	512 lines
(#sets	512)
Offset bits	5 bits
Set bits	9 bits
Tag bits	18 bits

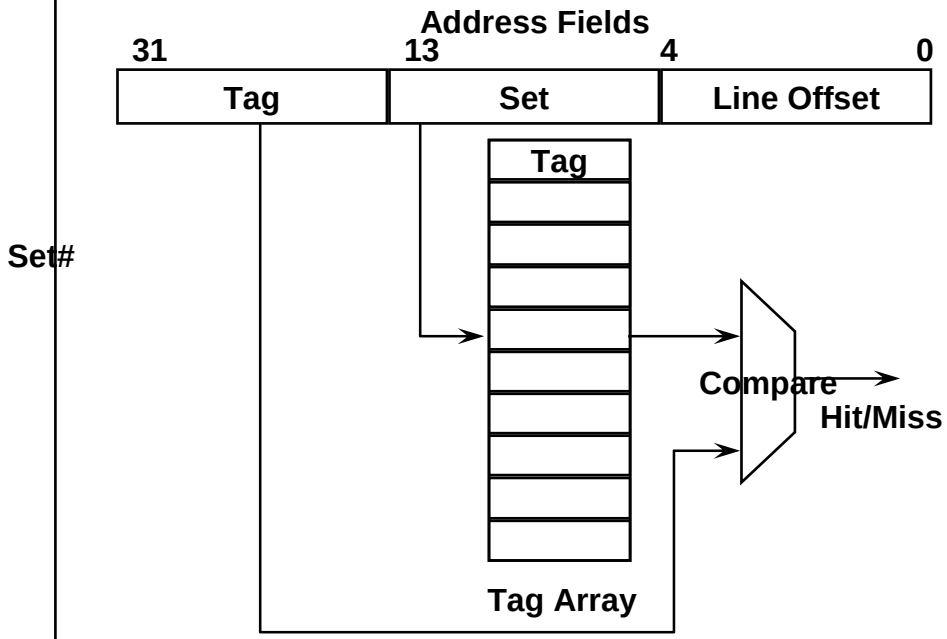
Address 0x12345678
0001 0010 0011 0100
0101 0110 0111 1000

Offset: 1 1000 = 0x18 = 24
Set: 0 1011 0011 = 0x0B3 = 179
Tag: 00 0100 1000 1101 0001 =
0x048B1

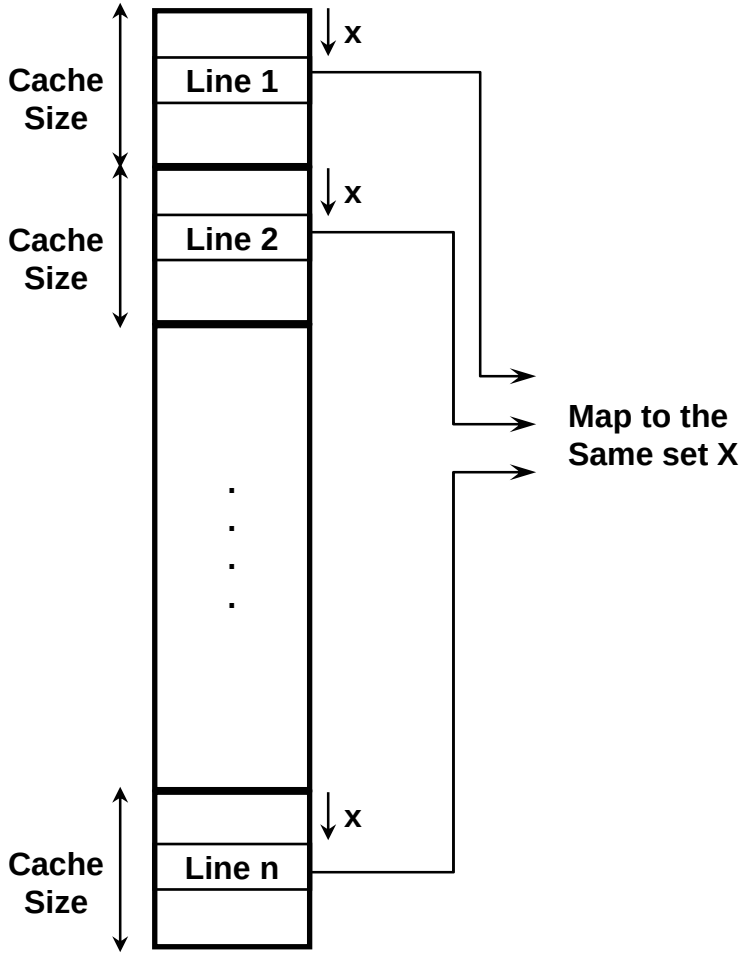


Direct Map Cache (Cont.)

- **Advantages**
  - Easy hit/miss resolution
  - Easy replacement algorithm
  - Lowest power
- **Disadvantage**
  - Excessive line replacement due to *conflict misses*



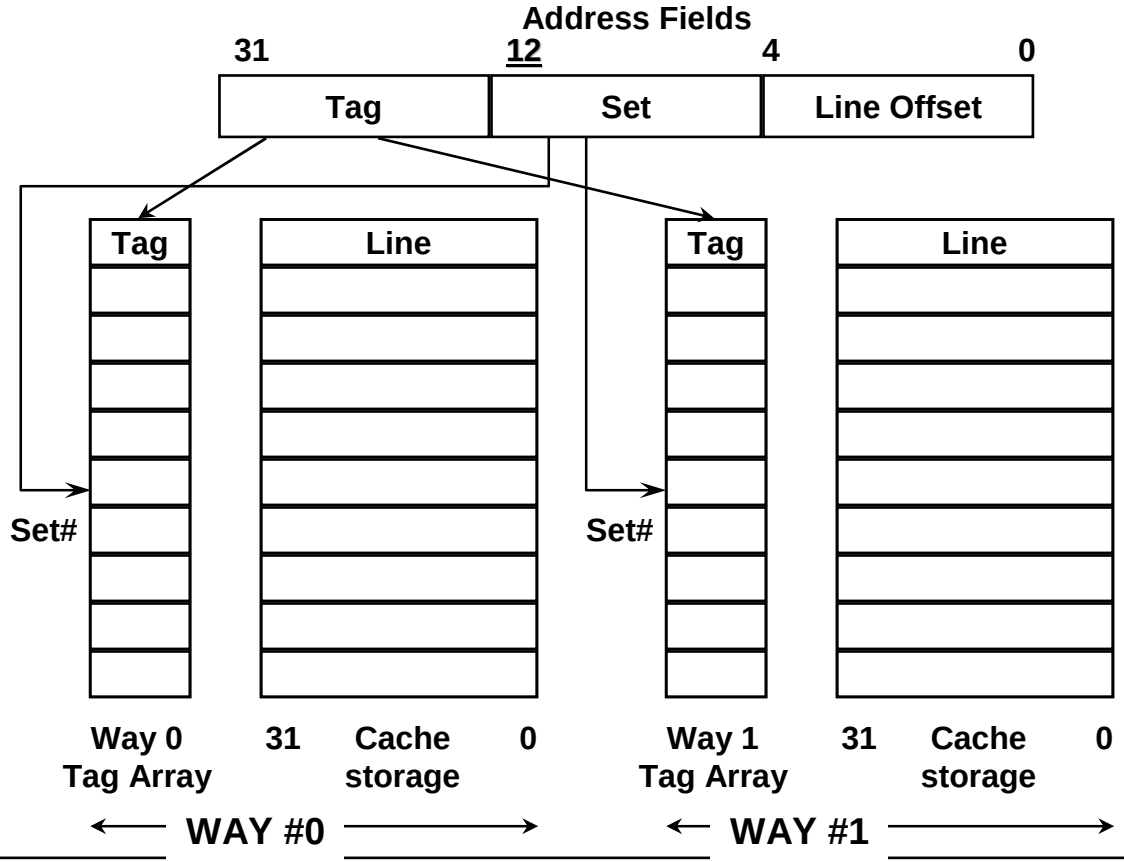
Direct Map Hit/Miss decision



Direct Map Line mapping

2-Way Set Associative Cache

- Memory is “conceptually” divided into slices whose size is 1/2 the cache size
- Offset from from slice start indicates set#
  But each set contains now two potential lines!
- Addresses w/ same offset map into the same set
- Two tags per set, one tag per line is needed



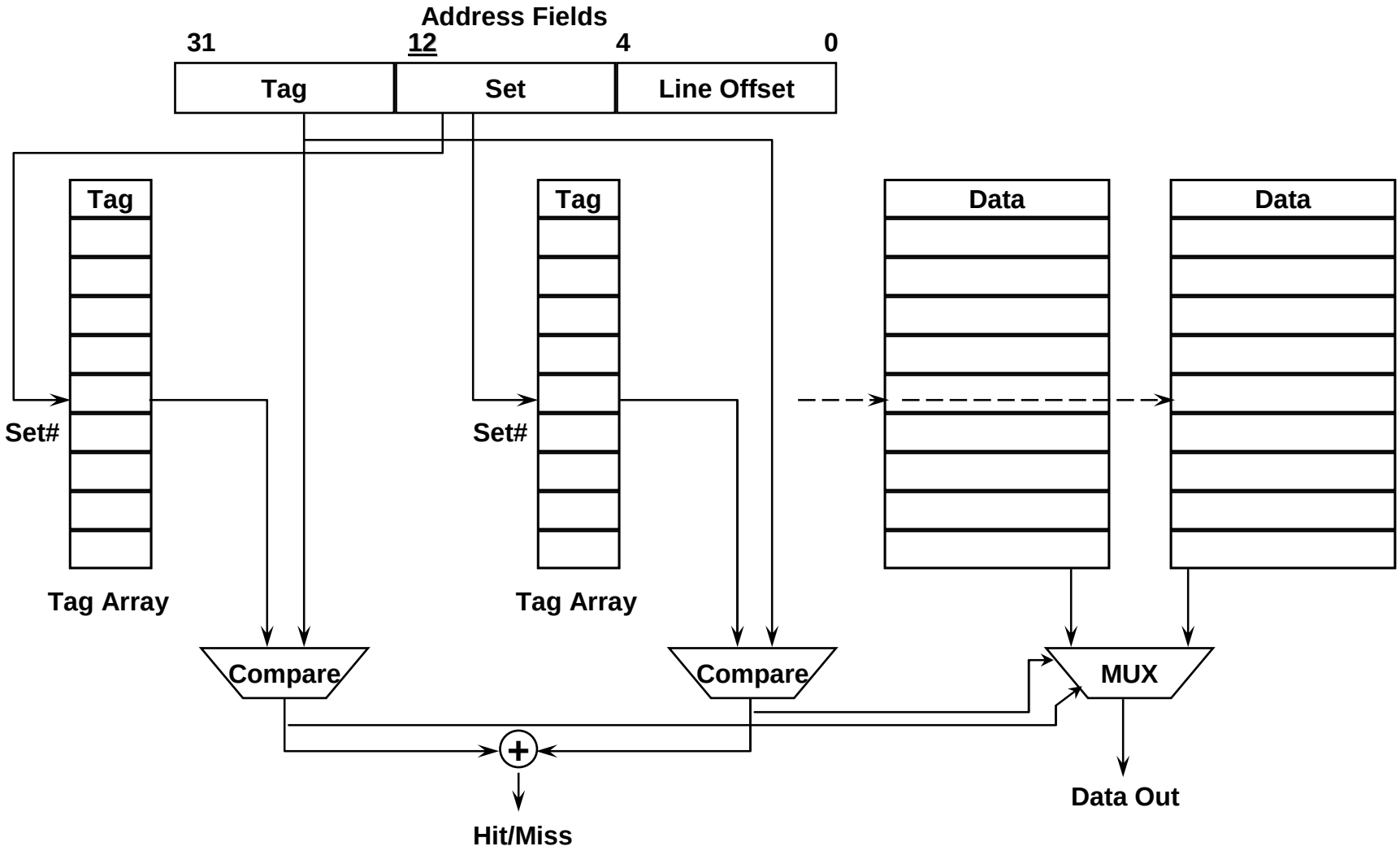
Example:

Line Size:	32 bytes
Cache Size	16KB
# of lines	512 lines
#sets	256
Offset bits	5 bits
Set bits	8 bits
Tag bits	19 bits

Address 0x12345678
0001 0010 0011 0100
0101 0110 0111 1000

Offset: 1 1000 = 0x18 = 24
Set: 1011 0011 = 0x0B3 = 179
Tag: 000 1001 0001 1010 0010 = 0x091A2

2-Way Cache - Hit Decision



Cache Structure

- 2-Way scheme scaleable to 4-way, 8-way and more...
  Trend: more ways for smaller caches, less for bigger
- Fully Associative: 1 set, # of ways = # of lines
  - Not feasible for a regular cache
  - Used for other small structures - e.g., TLBs
- Address fields for 16KB cache
  - 32 byte lines
  - 512 lines

Organization	Ways	Tag	Set	Offset
Direct	1	14-31 (18)	5-13 (9bits)	0-4 (5bits)
2 Way Set	2	13-31 (19)	5-12 (8bits)	“
4 Way Set	4	12-31 (20)	5-11 (7bits)	“
8 Way Set	8	11-31 (21)	5-10 (6bits)	“
Fully Assoc.	512	5-31 (27)	n/a	“

Replacement

- Each line contains *valid* indication
- Direct map: simple, line can be brought to only one place
  - Old line is evicted (re-written to memory, if needed)
- n-Ways: need to choose among ways in set
  Options: LRU, Random, Pseudo LRU.
  - 2 ways: requires 1 bit for a set to mark latest accessed
  - 4-ways: requires 2 bits for to approximate LRU.
  - Fully associative: approximate LRU (*pseudo-LRU*)
- How is the line size determined?
  - Long enough to gain from spatial (space) locality
  - Long enough to make effective use of bursts
  - Small to reduce line fill / write back time
  - Small to avoid internal fragmentation

  => Rule of thumb ~4X bus size

Elements of Cache performance

Already mentioned:

- **Size: bigger is better (as long as latency kept low)**
- **#of ways: more is better**
- **Replacement algorithms (LRU is better than random)**
- **Optimal line size**

More:

- **Memory update policies (see later)**
  - **Write-through: memory is always updated**
  - **Write-Back: smarter update, only when needed**
    Faster, better, but more complex to implement
- **Unified/Split Caches**
  - **Split instruction and data (common)**
  - **Split kernel/user (rare)**
- **Indexing: Virtual/Physical caches**
  - **Virtual is faster (no TLB access), but consistency is hard**

Write Hit Policy

- **WT – Write Through**
  - **All memory writes are written to main memory, even if the line is in cache**
  - **Memory is always updated**
  - **Do nothing on replacement**
  - **Valid / invalid**
  - **Simple, slow, lot of bus traffic (immediate writes, small chunks)**
  - **Write buffers are used to avoid write latency issues**
- **WB – Write Back**
  - **Hold a line in the cache, even if modified**
  - **Write back line to memory on replacement**
  - **Invalid, Non-modified, Modified (Dirty bit)**
  - **Smarter update, only when needed**
  - **Faster, better, but more complex to implement**

Write Miss Policy

- **Write Allocate**
  - **Write to cache**
  - **The line is loaded, followed by the write-hit actions above.**
  - **Similar to a read miss.**
- **No Write Allocate**
  - **Bypass cache**
  - **Write directly to main memory**

Cache Relevant Instructions

- **INVD:**
  Invalidate entire cache(15 cycles)

- **WBINVD:**
  Write Back and Invalidate entire cache (>2000 cycles)