Lecture 08
캐시와 가상 메모리

4190.414A
Multicore Computing
Fall 2017
© Jaejin Lee

캐시

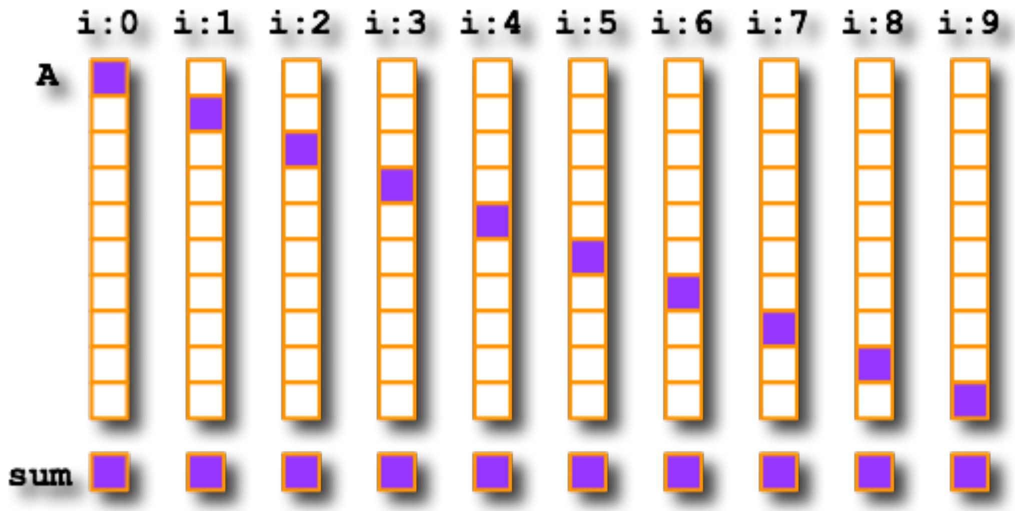
Principle of Locality

- The reuse of data or instructions that were recently used, or near those that have been used recently
  - Predictable behavior
- Temporal locality
  - Recently referenced items are likely to be referenced in the near future
  - Within relatively small time durations
- Spatial locality
  - Items in nearby locations tend to be referenced close together in time
  - Within relatively close locations and relatively small time durations

For Data

- Spatial locality
  - Reference array elements (A[i]) in succession (stride = 1)
- Temporal locality
  - Reference sum in each iteration

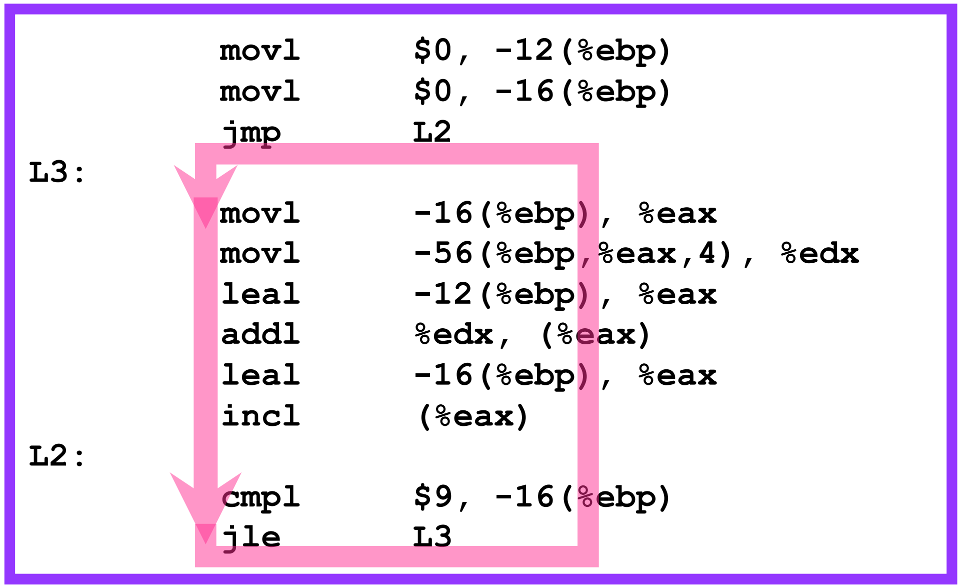
```
sum = 0;
for (i = 0; i < 10; i++)
   sum += A[i];
```



For Instructions

- Spatial locality
  - Reference instructions in sequence
- Temporal locality
  - Cycle through loop repeatedly

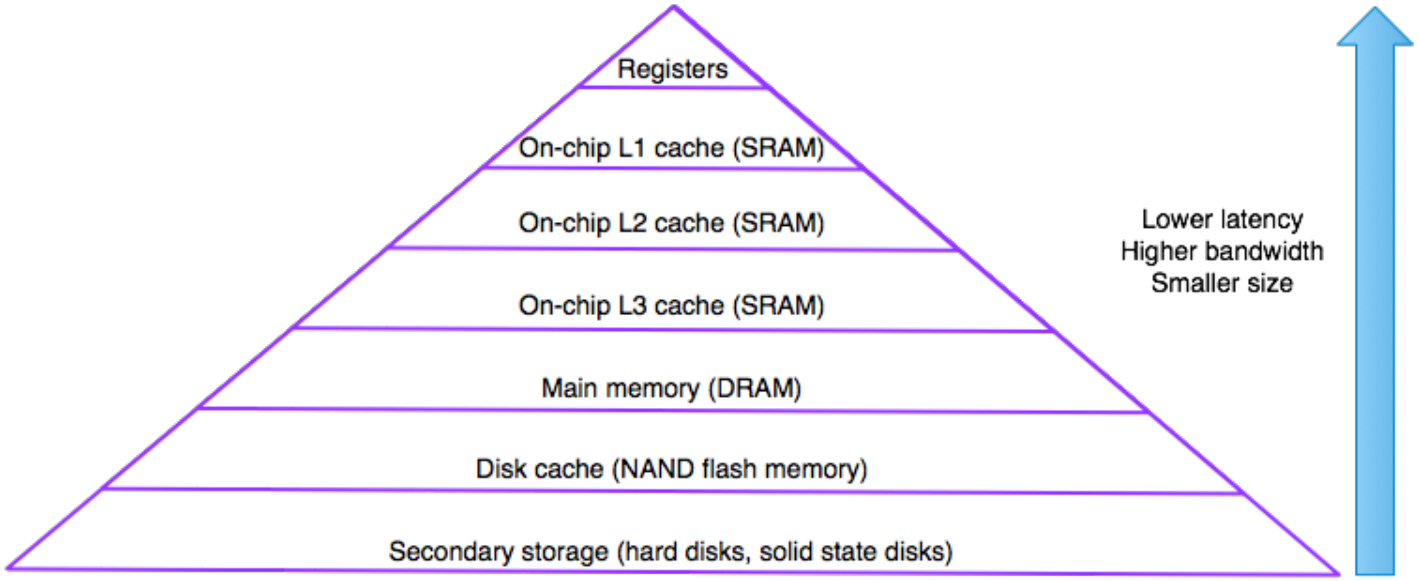
```
sum = 0;
for (i = 0; i < 10; i++)
  sum += A[i];
```



```
          movl      $0, -12(%ebp)
          movl      $0, -16(%ebp)
          jmp       L2
L3:
          movl      -16(%ebp), %eax
          movl      -56(%ebp,%eax,4), %edx
          leal      -12(%ebp), %eax
          addl      %edx, (%eax)
          leal      -16(%ebp), %eax
          incl      (%eax)
L2:
          cmpl      $9, -16(%ebp)
          jle       L3
```

Memory Hierarchies

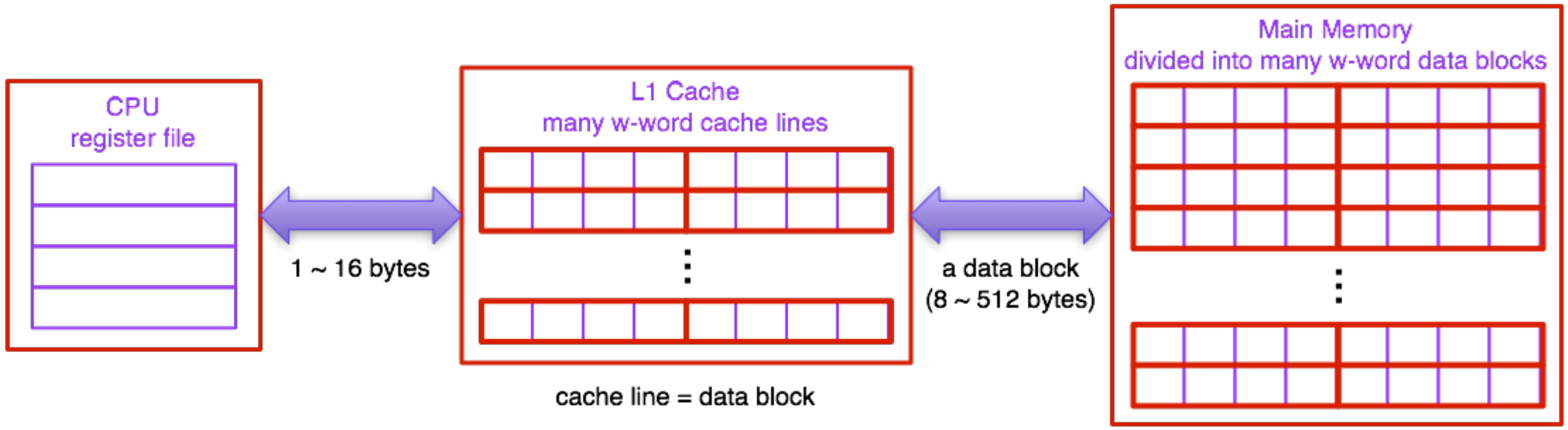
- Hierarchical arrangement of storage
  - To exploit locality of reference
- Fast storage technologies cost more per byte and have less capacity
- The gap between CPU and main memory speed is widening



Caching

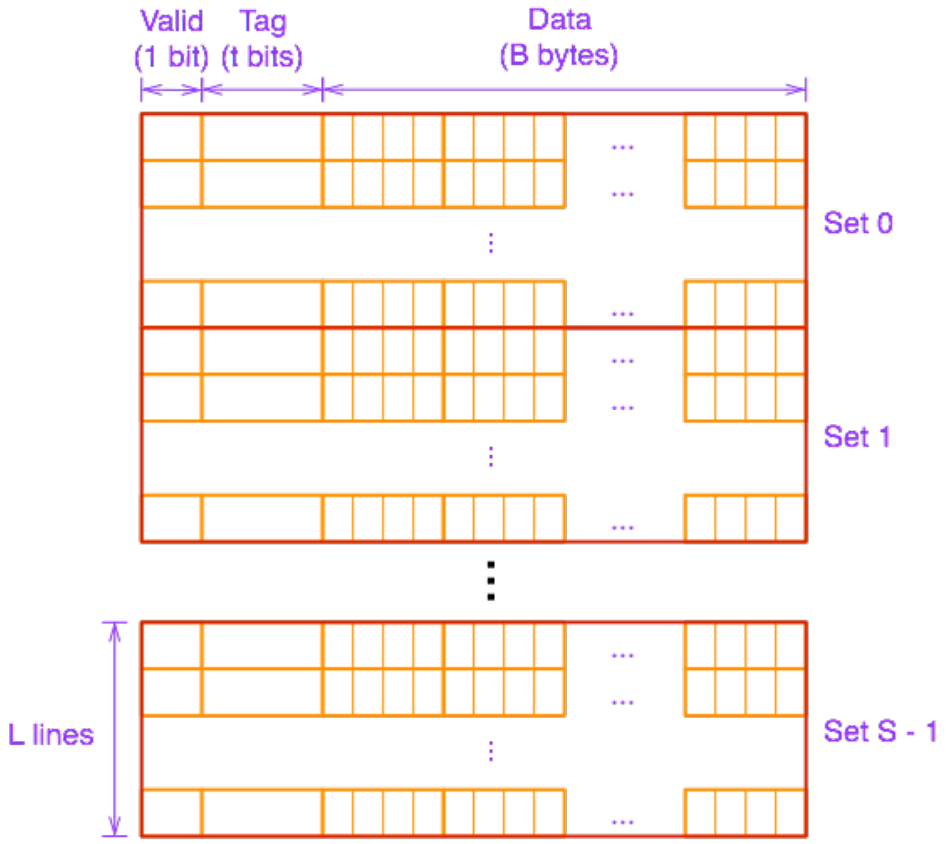
- Exploit temporal locality
  - Remember the contents of recently accessed locations
- Exploit spatial locality
  - Remember the blocks of recently accessed locations
- Cache block = cache line
  - The basic unit for cache storage
  - Multiple bytes or words
- Need an item d, which is stored in some block b
  - Cache hit
    - Find block b in the cache at level k
  - Cache miss
    - Block b is not in the cache at level k
    - The cache at level k must fetch b from level k+1
      - If the cache at level k is full, then some block in the cache must be replaced

L1 Cache between CPU and Main Memory



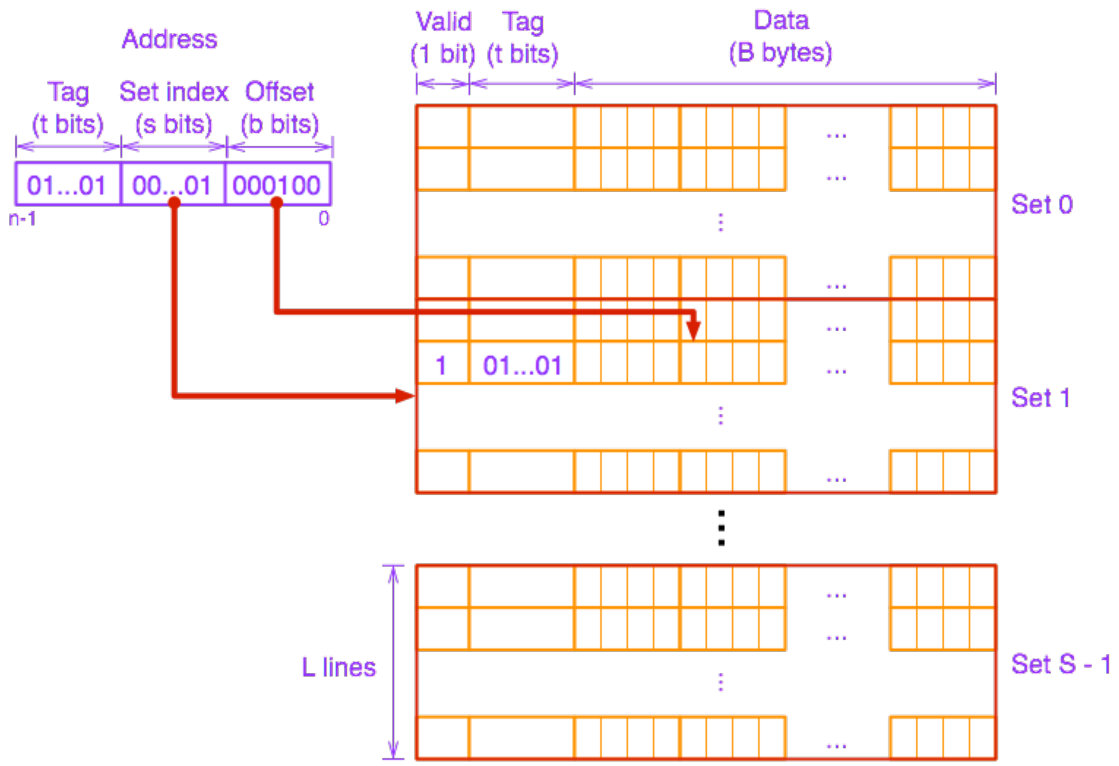
Cache Organizations in General

- Cache size = L×S×B bytes
- A set is a collection of cache locations in which a given block may be placed



Locating Data in the Cache

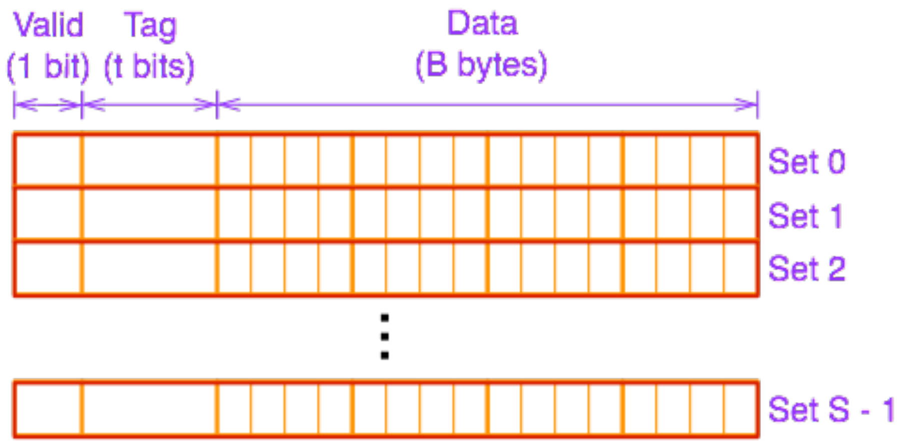
- The word at the requested address is in the cache if the tag bits in one of the valid lines in the specified set match the tag bits in the address
  - The set index is specified by the set index filed of the address
- The location of the word in the block is specified by the offset field in the address



$S = 2^s$
$B = 2^b$

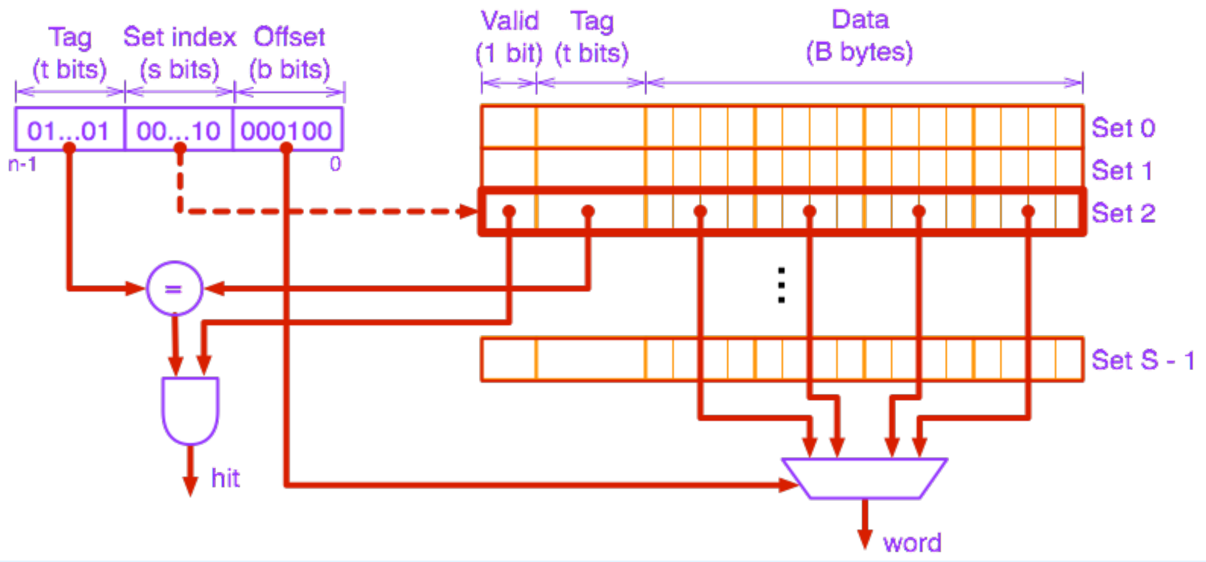
Direct-Mapped Caches

- One cache line per set
- Simplest
- Data block can be only in one place in the cache
  - Replacement is straightforward
  - Collisions between data blocks for the same cache line can occur



Addressing Direct-Mapped Caches

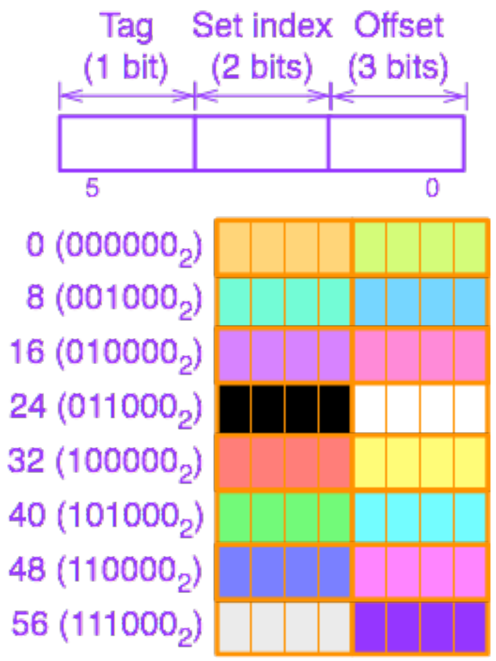
- Find a valid line in the selected set with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the selected set, and update the valid bit

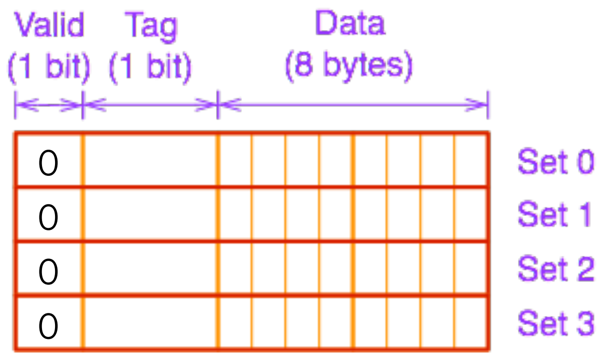


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000₂)

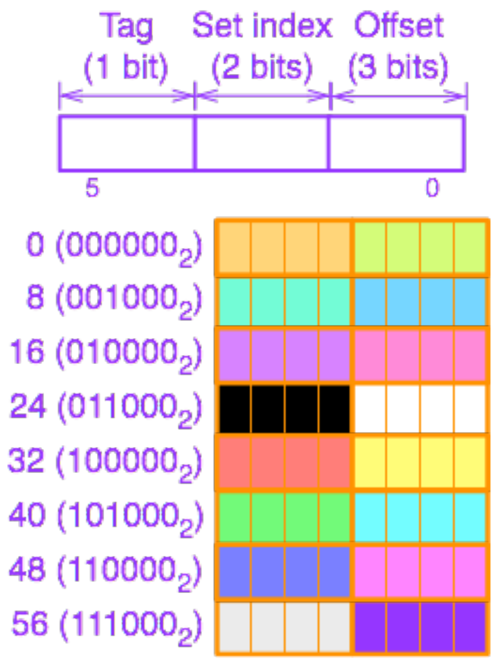


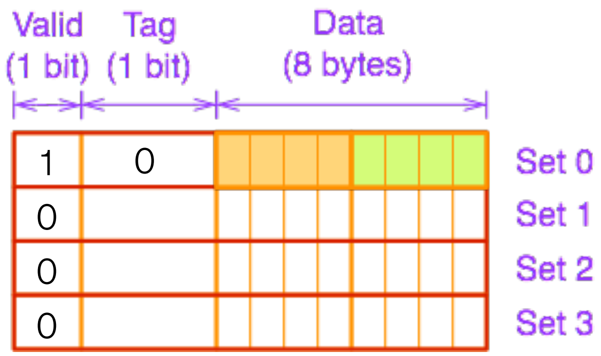


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000₂)

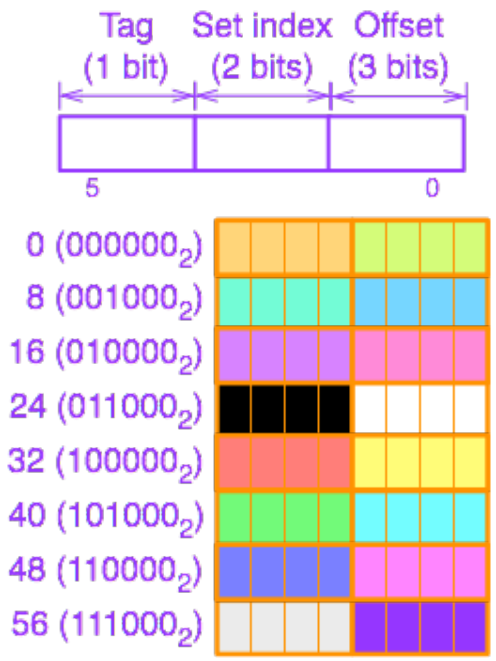


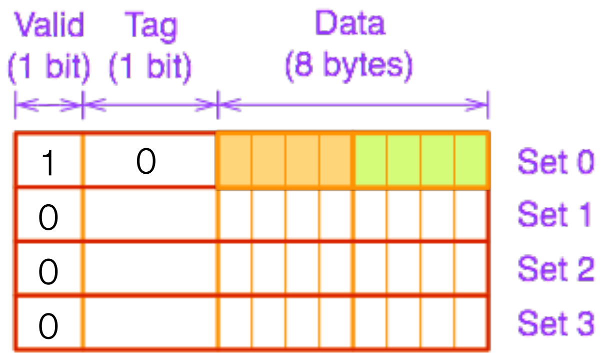


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2) 4 (000100_2)

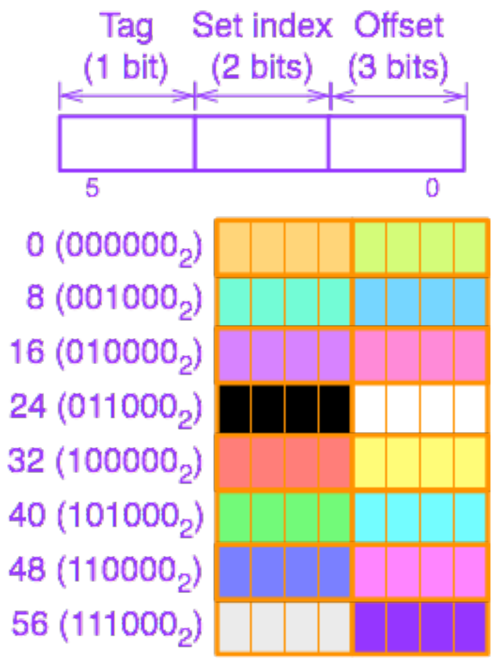


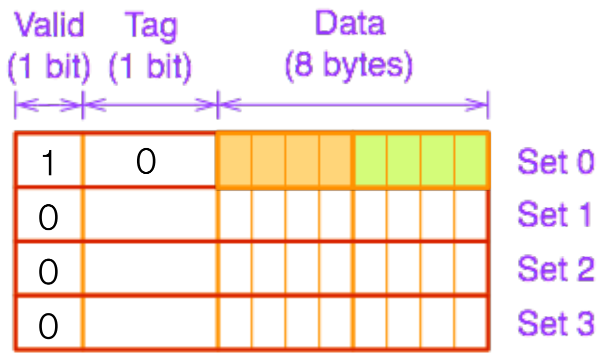


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2)   4 (000100_2)   20 (010100_2)

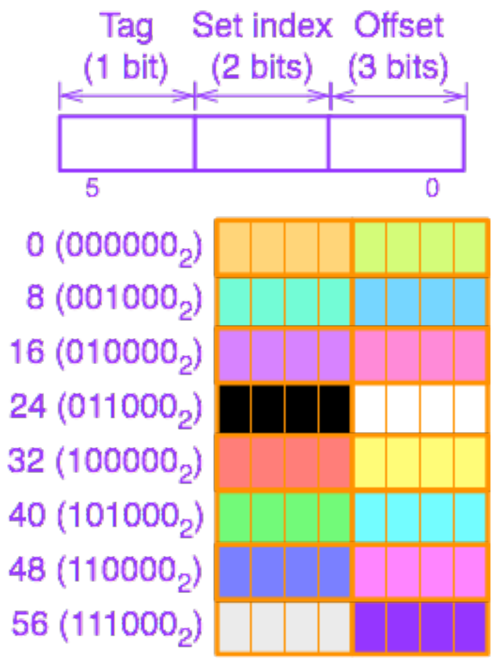


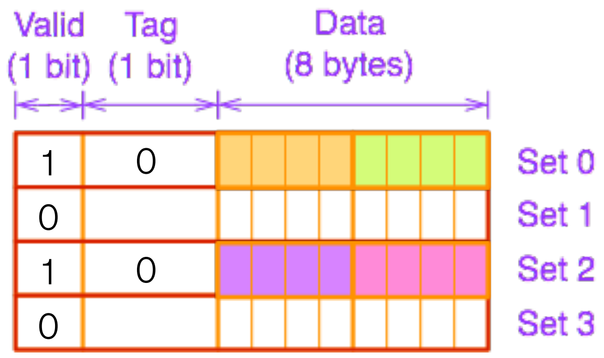


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2)   4 (000100_2)   20 (010100_2)

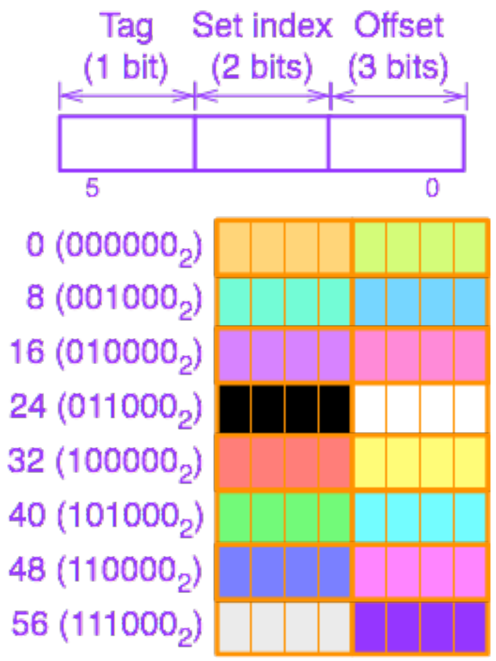


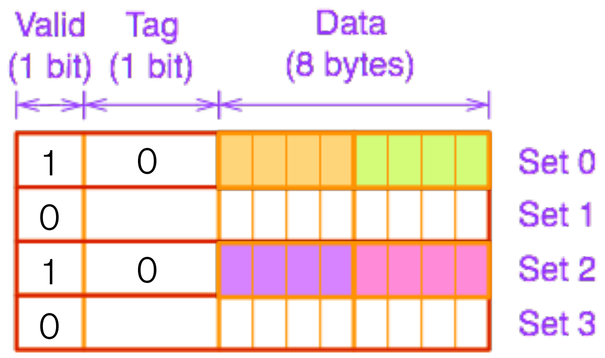


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2)   4 (000100_2)   20 (010100_2)   48 (110000_2)

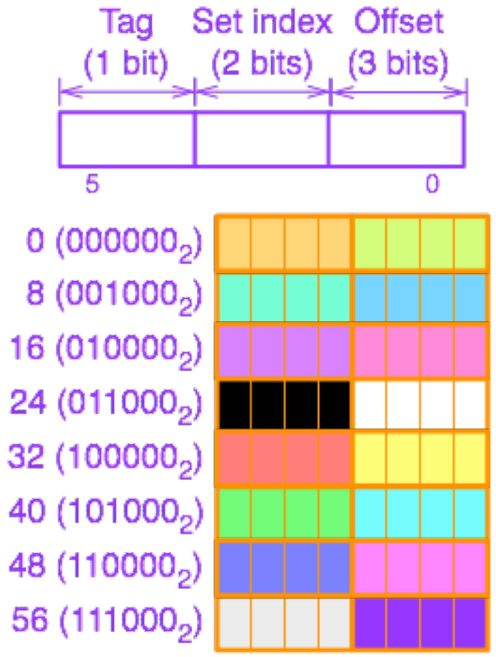


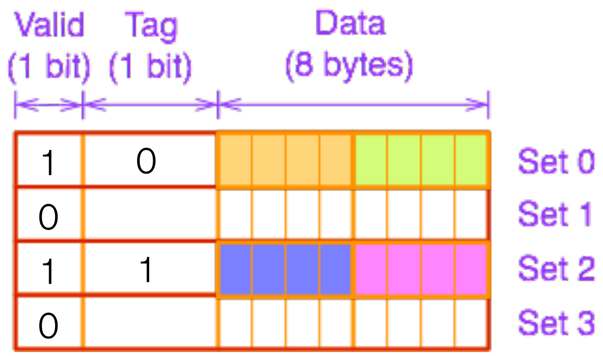


Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2) 4 (000100_2) 20 (010100_2) 48 (110000_2)



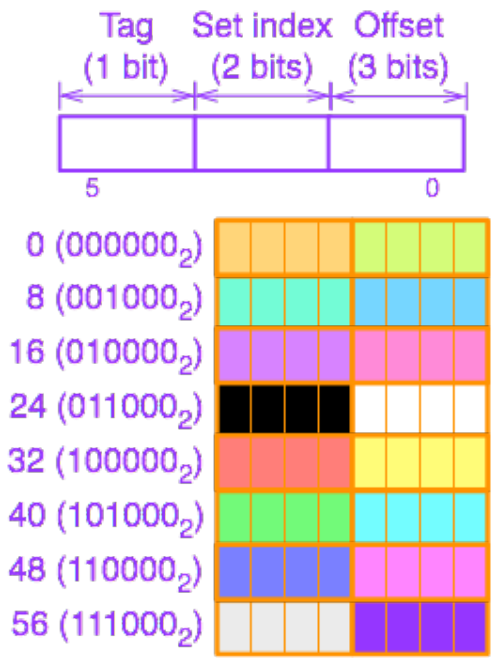


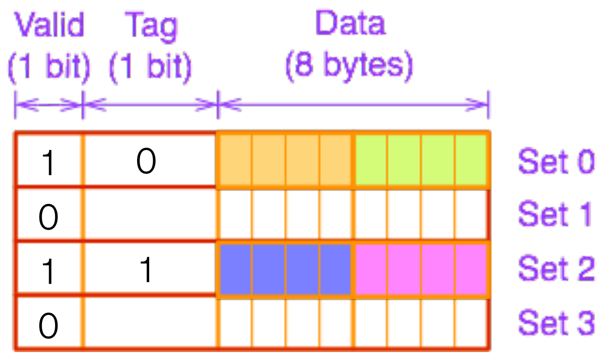
Valid	Tag	Data	
1	0		Set 0
0			Set 1
1	1		Set 2
0			Set 3

Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000_2) 4 (000100_2) 20 (010100_2) 48 (110000_2) 36 (100100_2)



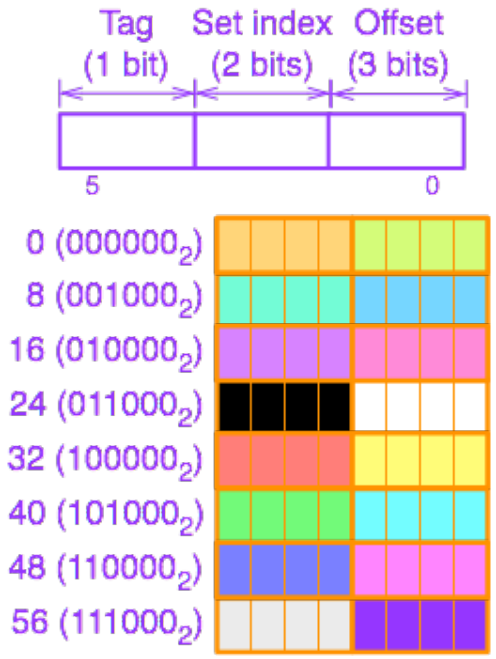


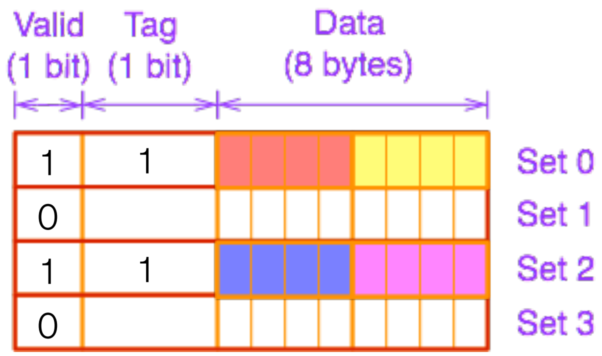
Valid	Tag	Data	
1	0		Set 0
0			Set 1
1	1		Set 2
0			Set 3

Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

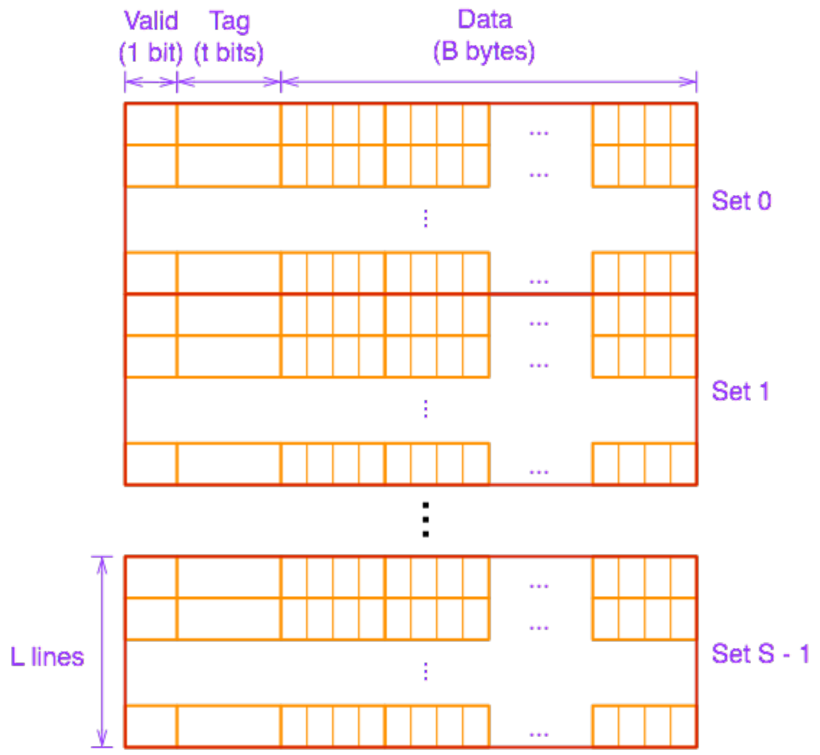
0 (000000_2) 4 (000100_2) 20 (010100_2) 48 (110000_2) 36 (100100_2)





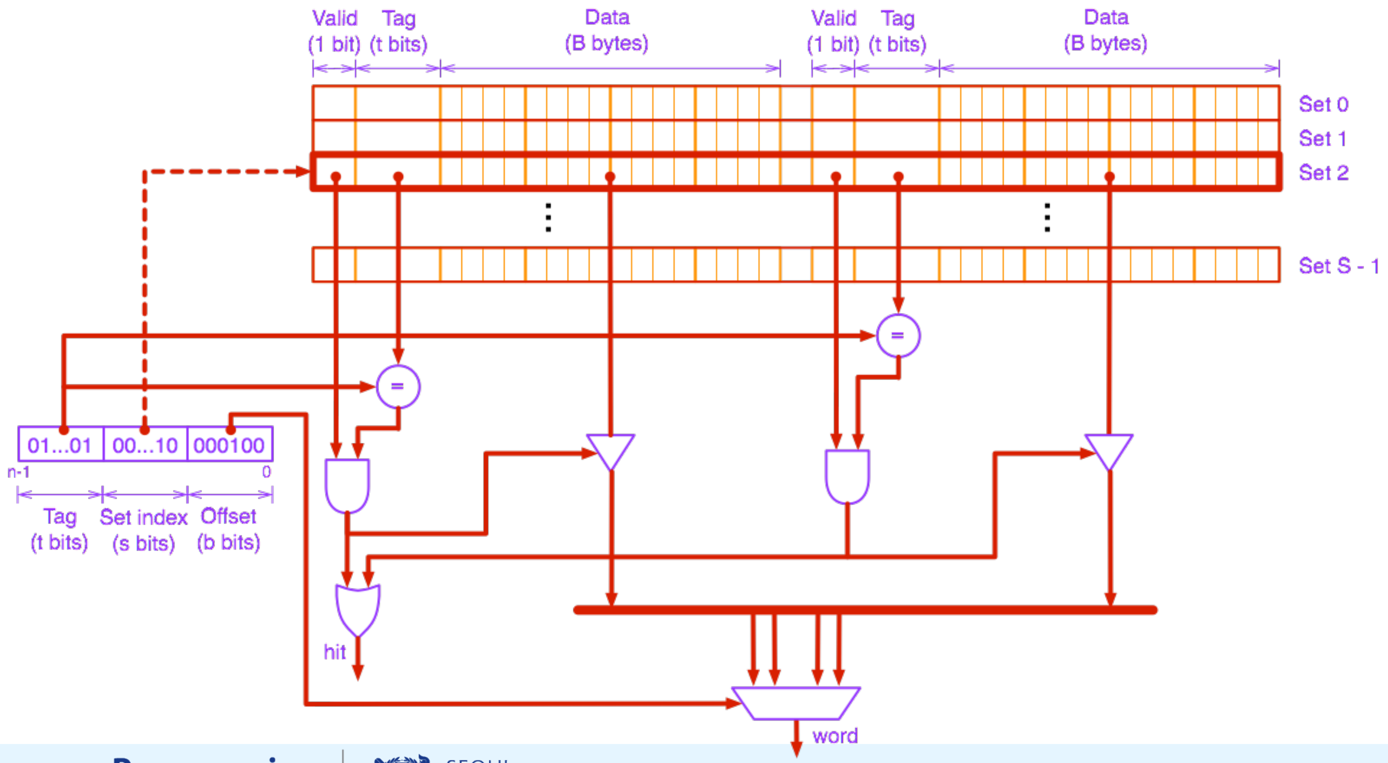
Set Associative Caches

- Data block can be in a few places in the cache
  - Need a good replacement policy
  - Less collisions between data blocks for the same cache line than the direct-mapped cache
- Complex tag comparison hardware on the lines in a set



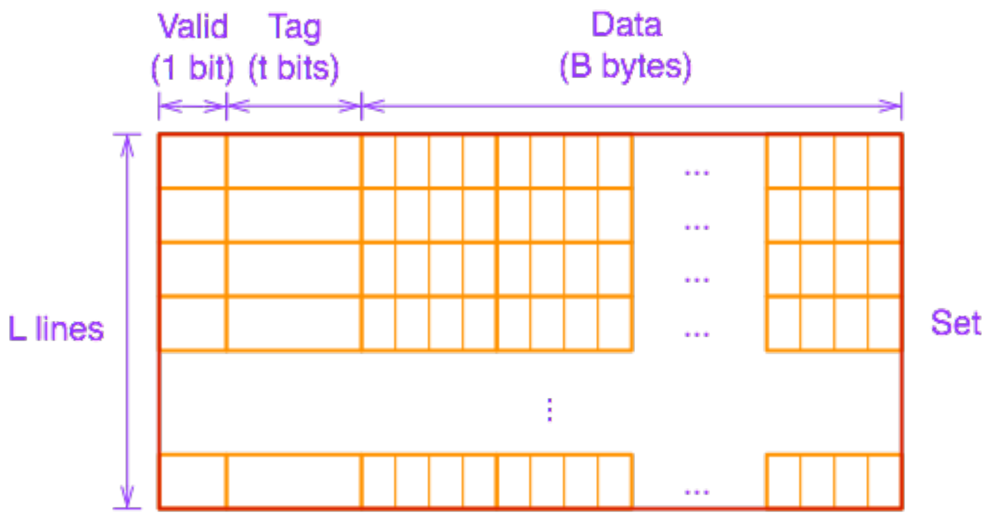
Addressing Set Associative Caches

- Find a valid line in the selected set with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the selected set by deciding which line should be used, and update the valid bit
  - Need a sophisticated replacement policy



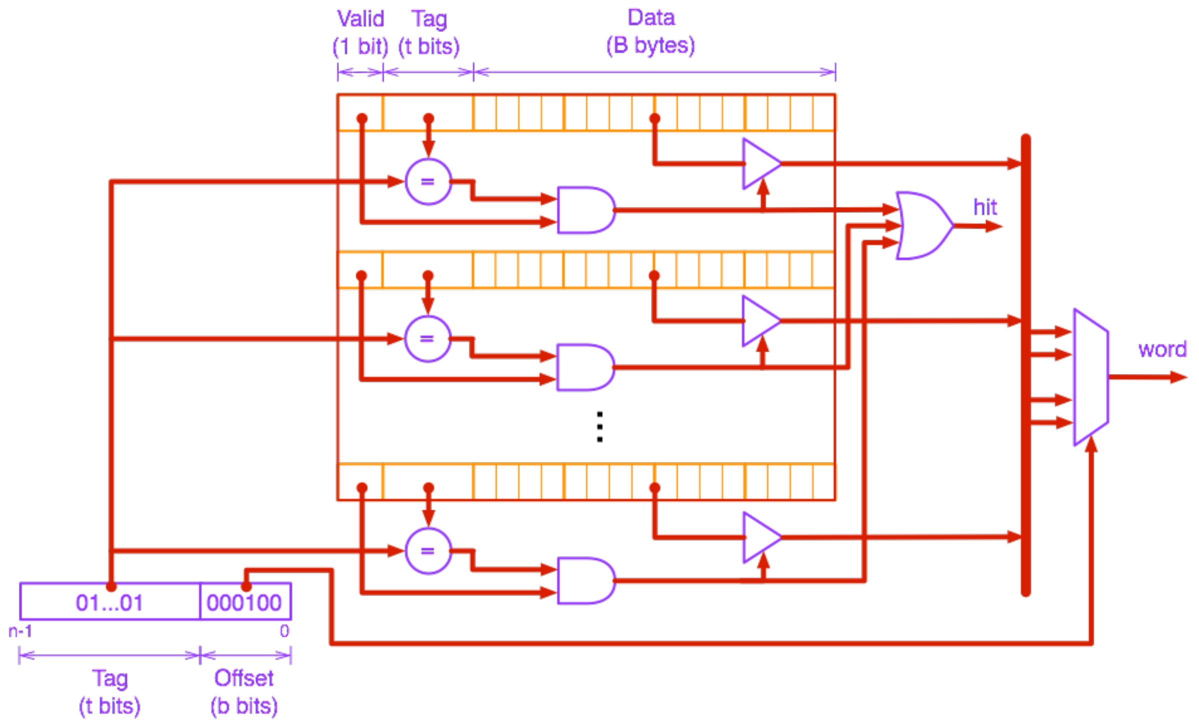
Fully Associative Caches

- Only one set
- Data block can be any place in the cache
  - Less collisions between data blocks for the same cache line than the set associative cache
- Complex tag comparison hardware on the lines in the cache



Addressing Fully Associative Caches

- Find a valid line with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the cache by deciding which line should be used, and update the valid bit
    - Need a sophisticated replacement policy



Types of Cache Misses

- Cold (compulsory) miss
  - When the cache is empty
- Conflict miss
  - When the cache is large enough, but multiple data items map to the same cache line
- Capacity miss
  - When the set of active cache lines (working set) is larger than the cache
  - Working set
    - The set of referenced blocks that are active during a given period of time

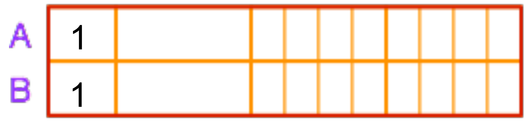
Replacement Policies

- After a miss, what cache block should be replaced with the block read from memory?
  - Which way in a multiway (i.e., set associative or fully associative) cache should be replaced?
  - Ideally, any cached data which is no longer needed would be chosen to be replaced
- LRU (Least Recently Used)
- Pseudo LRU
- FIFO (First In, First Out)
  - Select a block that has been in the set for the longest time
- Random

Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
- A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
  - Too costly
- Instead, use pseudo LRU

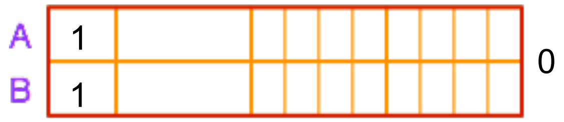
…



Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
- A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
  - Too costly
- Instead, use pseudo LRU

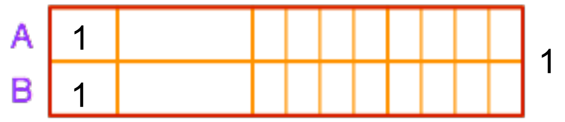
... A



Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
- A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
  - Too costly
- Instead, use pseudo LRU

… A B

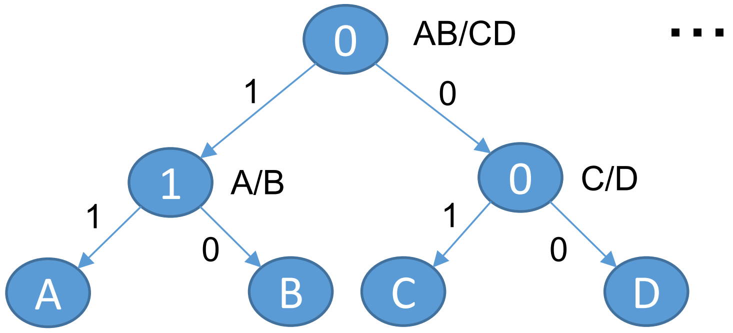


Pseudo LRU

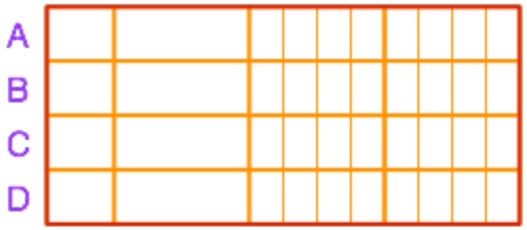
- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: (23-1) - 4 = 3 bits
  - N-way set associative cache: (2(log2N + 1) -1) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

access	next state
A	11_
B	10_
C	0_1
D	0_0

state	replace
00X	A
01X	B
1X0	C
1X1	D



AB/CD	A/B	C/D
0	1	0

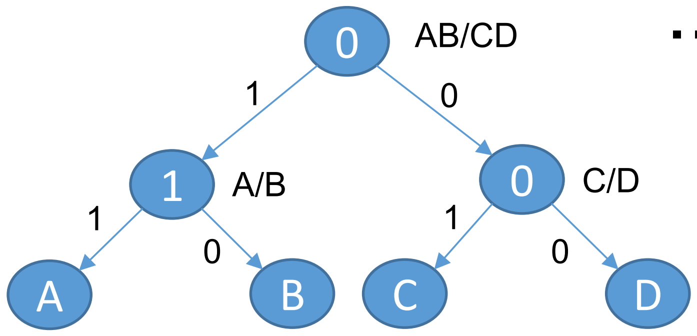


Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: (23-1) - 4 = 3 bits
  - N-way set associative cache: (2(log2N + 1) -1) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

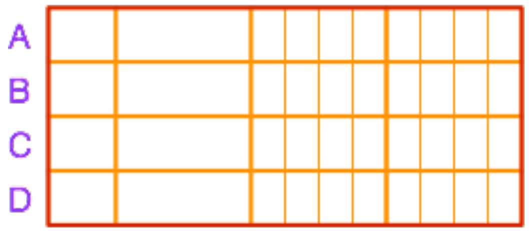
access	next state
A	11_
B	10_
C	0_1
D	0_0

state	replace
00X	A
01X	B
1X0	C
1X1	D



… A

AB/CD	A/B	C/D
0	1	0

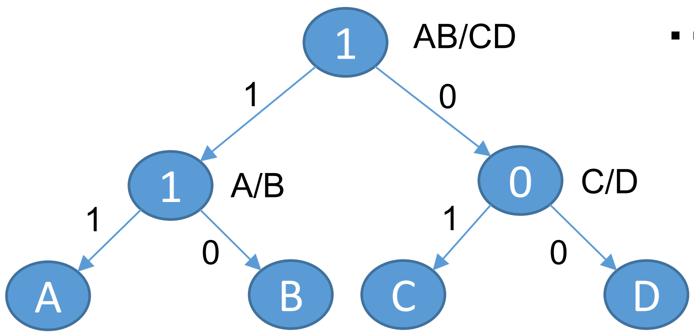


Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: (23-1) - 4 = 3 bits
  - N-way set associative cache: (2(log2N + 1) -1) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

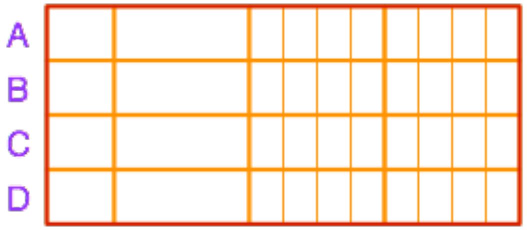
access	next state
A	11_
B	10_
C	0_1
D	0_0

state	replace
00X	A
01X	B
1X0	C
1X1	D



… A

AB/CD	A/B	C/D
1	1	0

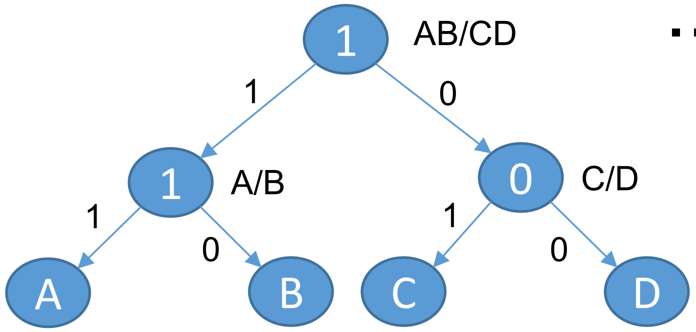


Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: (23-1) - 4 = 3 bits
  - N-way set associative cache: (2(log2N + 1) -1) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

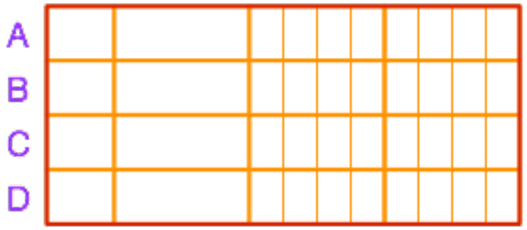
access	next state
A	11_
B	10_
C	0_1
D	0_0

state	replace
00X	A
01X	B
1X0	C
1X1	D



… A C

AB/CD	A/B	C/D
1	1	0

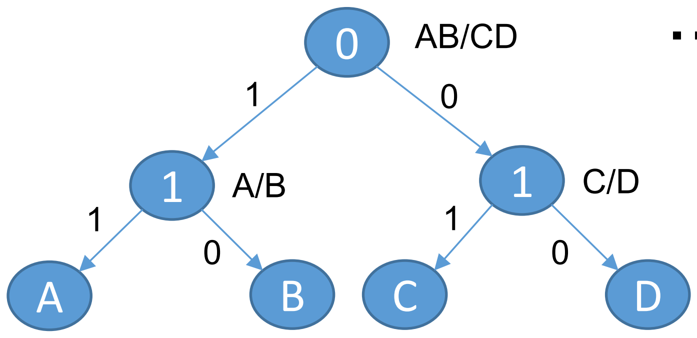


Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: (23-1) - 4 = 3 bits
  - N-way set associative cache: (2(log2N + 1) -1) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

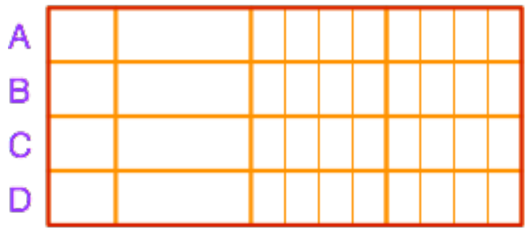
access	next state
A	11_
B	10_
C	0_1
D	0_0

state	replace
00X	A
01X	B
1X0	C
1X1	D



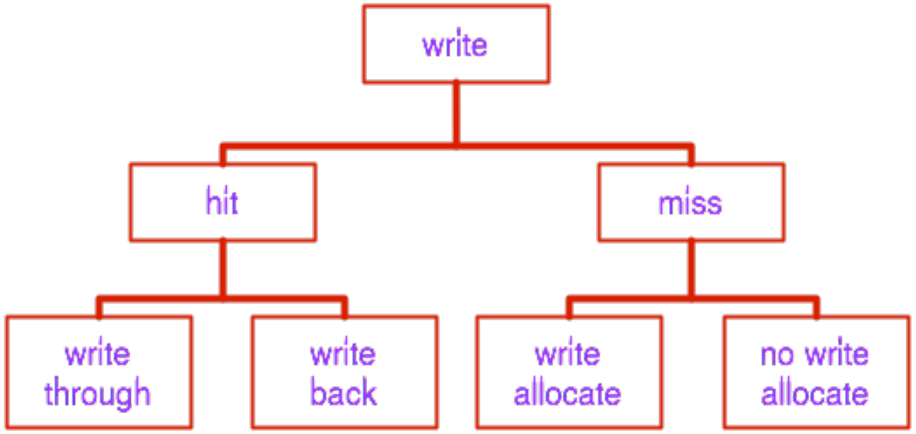
… A C

AB/CD	A/B	C/D
0	1	1



Write Policies

- For reads, the block can be read at the same time that the tag is compared
  - If a miss, just ignore the value read
- For writes, modifying the block cannot begin until the tag is compared
  - Only some part of the entire block is modified



Write Policies when a Hit

Write through	Write back
• Both the block in the cache and the blo ck in the lower level memory are modifi ed	• Only the block in the cache is modified • The block is written back to the lower lev el memory when it is replaced • A dirty bit is used to reduce the frequenc y of writing back blocks on replacement
• Simpler to implement • Writes are slower than reads • The lower level memory is always cons istent with the cache • Every write requires the lower level me mory access (need more memory ban dwidth) • Read misses never result in writes to th e lower level memory	• Harder to implement • Writes and reads are preformed at the same speed • The lower level memory is not always consistent with the cache • Multiple writes within a block require onl y one write to the lower level memory (need less memory bandwidth) • Read misses may cause writes of dirty blocks to the lower level memory due t o replacement

Write Policies when a Hit

Write through	Write back
• Both the block in the cache and the block in the lower level memory are modified	• Only the block in the cache is modified – The block is written back to the lower level memory when it is replaced – A dirty bit is used to reduce the frequency of writing back blocks on replacement
• Simpler to implement • Writes are slower than reads • The lower level memory is always consistent with the cache • Every write requires the lower level memory access (need more memory bandwidth) • Read misses never result in writes to the lower level memory	• Harder to implement • Writes and reads are preformed at the same speed • The lower level memory is not always consistent with the cache • Multiple writes within a block require only one write to the lower level memory (need less memory bandwidth) • Read misses may cause writes of dirty blocks to the lower level memory due to replacement

Write Policies when a Miss

- Write allocate
  - The block is loaded into the cache on a write miss
- No write allocate
  - The block is modified in the lower level memory and not loaded into the cache

Write through and write allocate	Write back and write allocate
• Subsequent writes to the same block will generate a write to the lower level memory anyway • Bringing the block in the cache is a waste of time	• On a miss it updates the block in the lower level memory and brings the block to the cache • Subsequent writes to the same block will hit in the cache
Write through and no write allocate	**Write back and no write allocate**
Not bringing the block in the cache on a miss saves time	Subsequent writes to the same block will generate misses

Non-Blocking/Lockup-Free Caches

- Most caches can handle only one outstanding request at a time
  - On a miss, the cache must wait for the lower level memory to supply the requested data and until then it is blocked
- A non-blocking cache continues to supply cache hits during a miss
  - Reduce effective miss penalty
  - Another option: supporting multiple outstanding misses
    - A special state need to be maintained for each outstanding miss
      - Miss Status/Information Holding Registers (MSHRs)

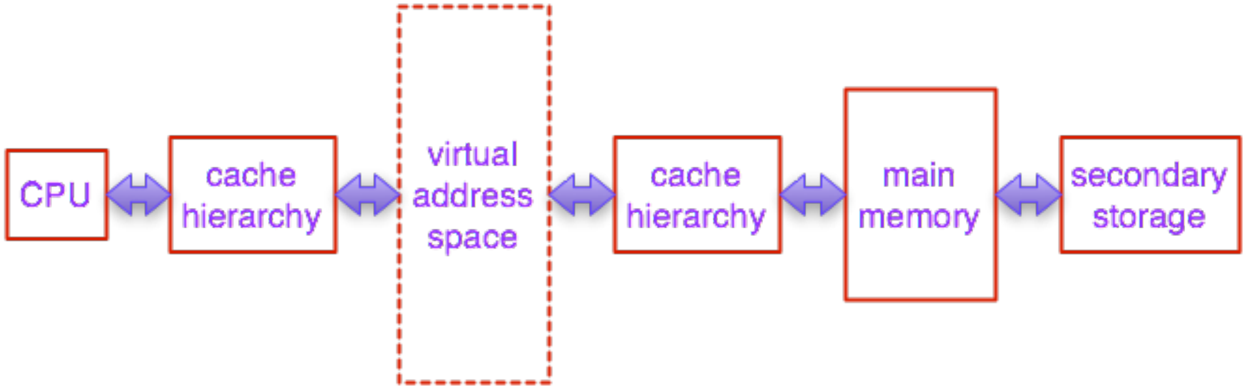
Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in the cache (misses/references)
- Hit Time
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Miss Penalty
  - Additional time required due to the miss

가상 메모리

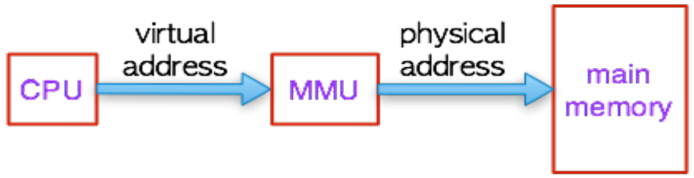
Virtual Memory

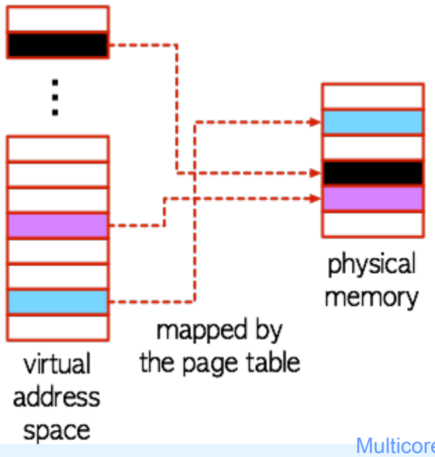
- An abstraction of main memory by the operating system
  - Provide each process with a large and uniform address space
    - The size of the address space is bigger than that of main memory
  - Protect the address space of each process from corruption by other processes
  - Treat main memory as a cache of the permanent secondary storage (hard disk)



MMU and Pages

- Each byte in main memory has a unique physical address (PA)
- The CPU generates a virtual address (VA) to access the main memory
- The memory management unit (MMU) translates the virtual address to the corresponding physical address using a look-up table (page table) stored in main memory
- The virtual address space is divided into uniform virtual pages
  - Each page is indexed by its virtual page number
- The physical memory is divided into uniform physical pages (page frames)
  - Each frame is indexed by its page frame number



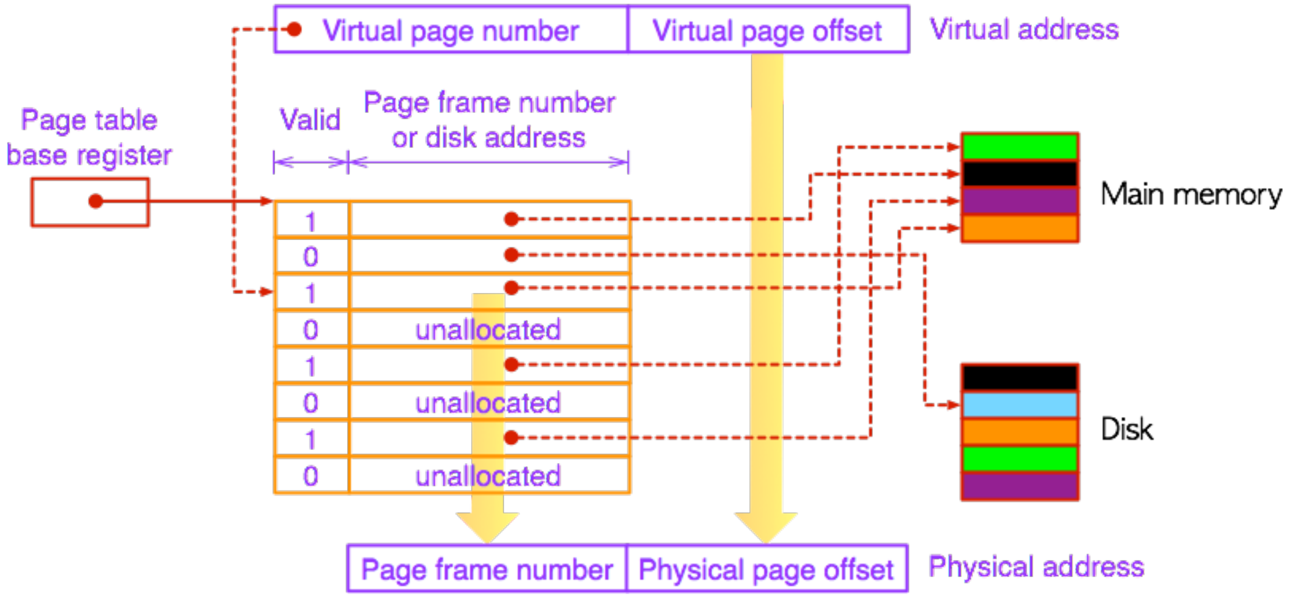


Page Tables

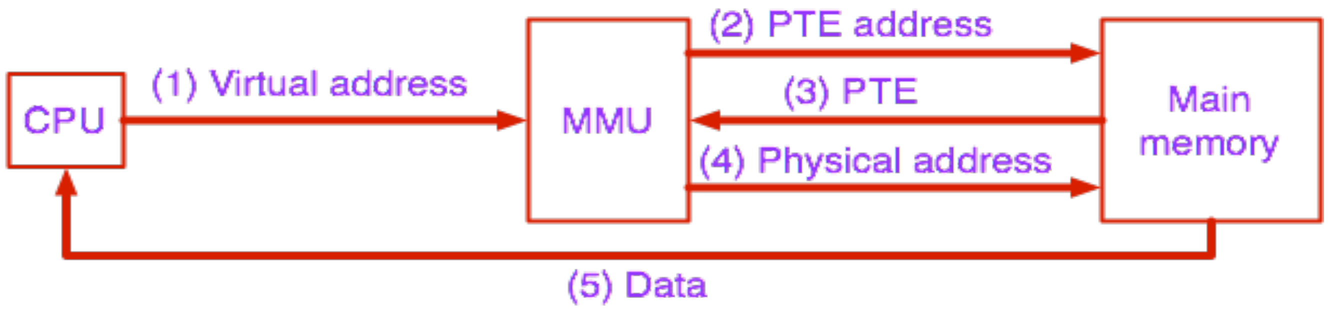
- Map virtual pages to physical pages
  - An array of page table entries (PTE)
  - A PTE consists of a valid bit and an n-bit address field (physical page frame number or secondary storage address) in addition to other page attributes
- The MMU reads the page table when it converts VA to PA
- The OS (page fault handler) takes care of maintaining the contents of the page table and transferring pages between main memory and secondary storage
- Swapping (paging)
  - The activity of transferring a page between the secondary storage and main memory
- Demand paging
  - Wait until the last moment to swap in a page when a miss (page fault) occurs

Address Translation

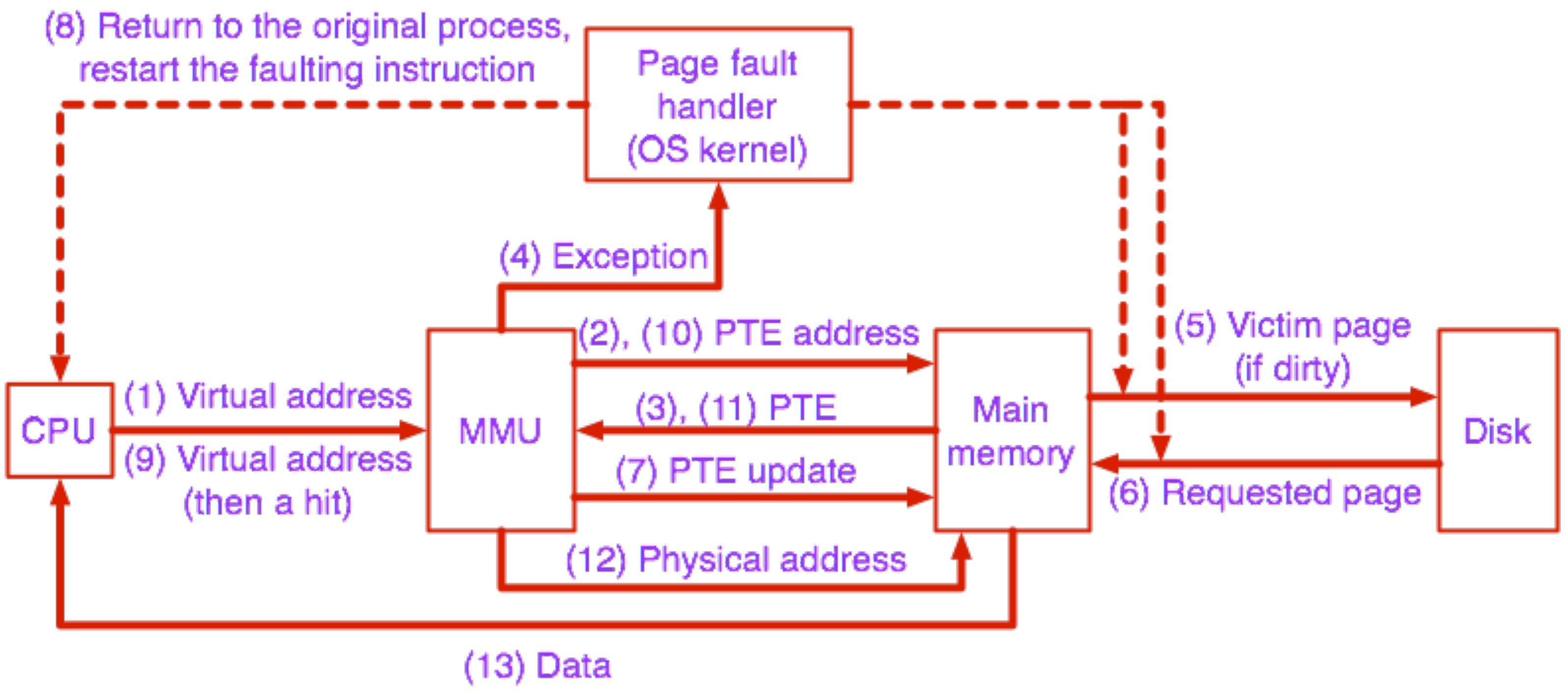
- Three types of virtual pages
  - Unallocated: Pages that have not yet been allocated by the VM (no space on secondary storage)
  - Cached: Allocated pages that are currently cached in main memory
  - Uncached: Allocated pages that are not cached in main memory (reside on secondary storage)
- A single page table for the entire address space is large
  - 32-bit address space, 4KB pages, and 4B PTEs result in 4MB page table resident in main memory
  - Use a hierarchy of page tables and demand paging for the tables



Page Hit

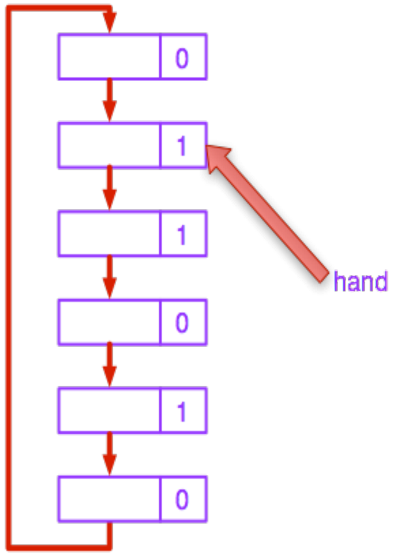

CPU
(1) Virtual address
MMU
(2) PTE address
(3) PTE
(4) Physical address
Main memory
(5) Data

Page Fault


(8) Return to the original process, restart the faulting instruction
Page fault handler (OS kernel)
(4) Exception
(2), (10) PTE address
(5) Victim page (if dirty)
CPU
(1) Virtual address
MMU
(3), (11) PTE
Main memory
Disk
(9) Virtual address (then a hit)
(7) PTE update
(6) Requested page
(12) Physical address
(13) Data

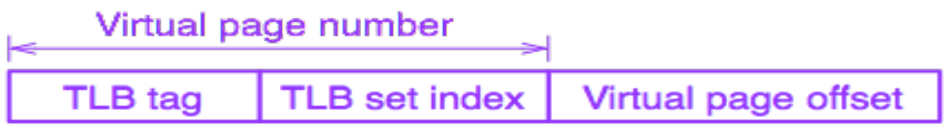
Page Replacement Policies

- LRU
- FIFO
- Second chance
- Clock
  - A bit (R) that indicates whether the page is referenced or not
    - When a page is first loaded in memory, R = 0
    - When the page is referenced, R = 1
  - Maintain a circular list of pages in memory
    - The hand points to the current page in the list
    - When it is time to replace a page, the first frame with R = 0 encountered is replaced
    - During the search for replacement, each reference bit set to 1 is changed to 0

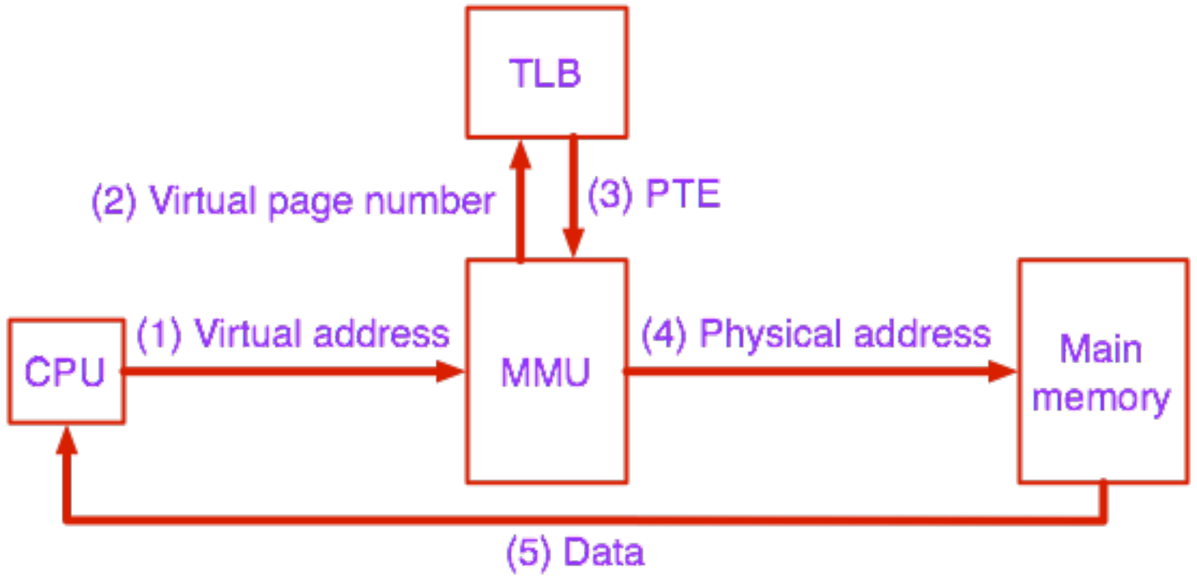


Translation Lookaside Buffer (TLB)

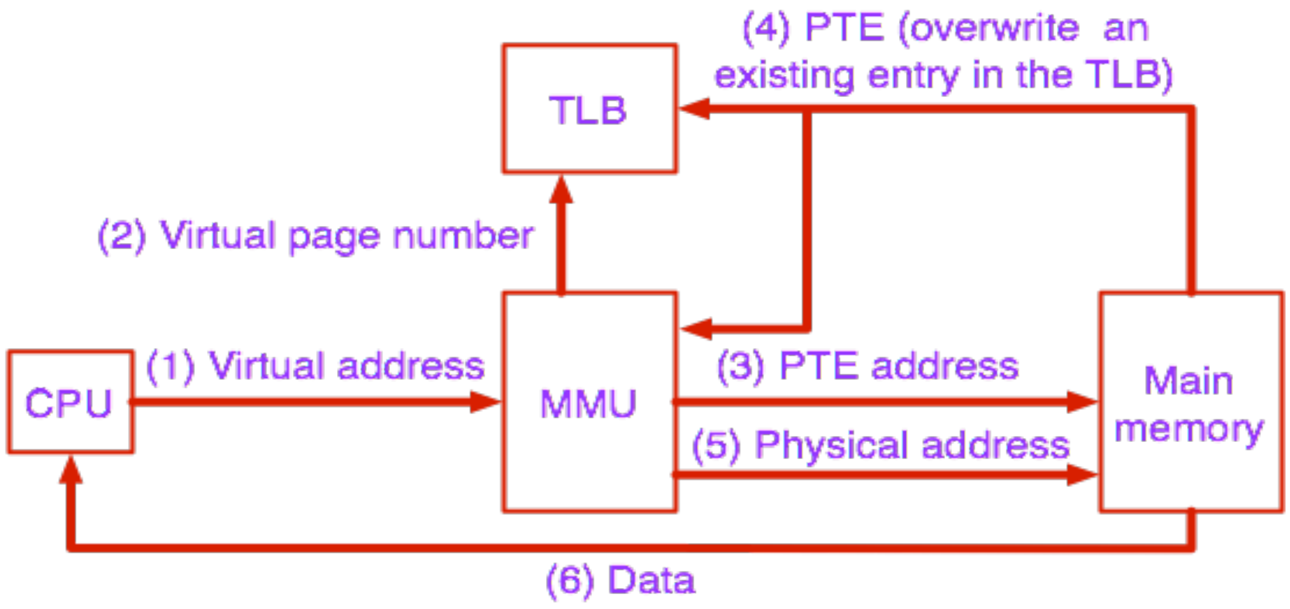
- Every time the CPU generates a virtual address, the MMU must refer to the page table for address translation
  - High overhead
- A small, virtually addressed cache where each line holds a block consisting of a single PTE
  - Has a high degree of associativity
- Micro-TLB
  - A small TLB placed over the main TLB to boost the speed of address translation for cache accesses
  - The main TLB handles micro-TLB misses
  - Smaller number of entries than the main TLB



TLB Hit


TLB
(2) Virtual page number
(3) PTE
CPU
(1) Virtual address
MMU
(4) Physical address
Main memory
(5) Data

TLB Miss

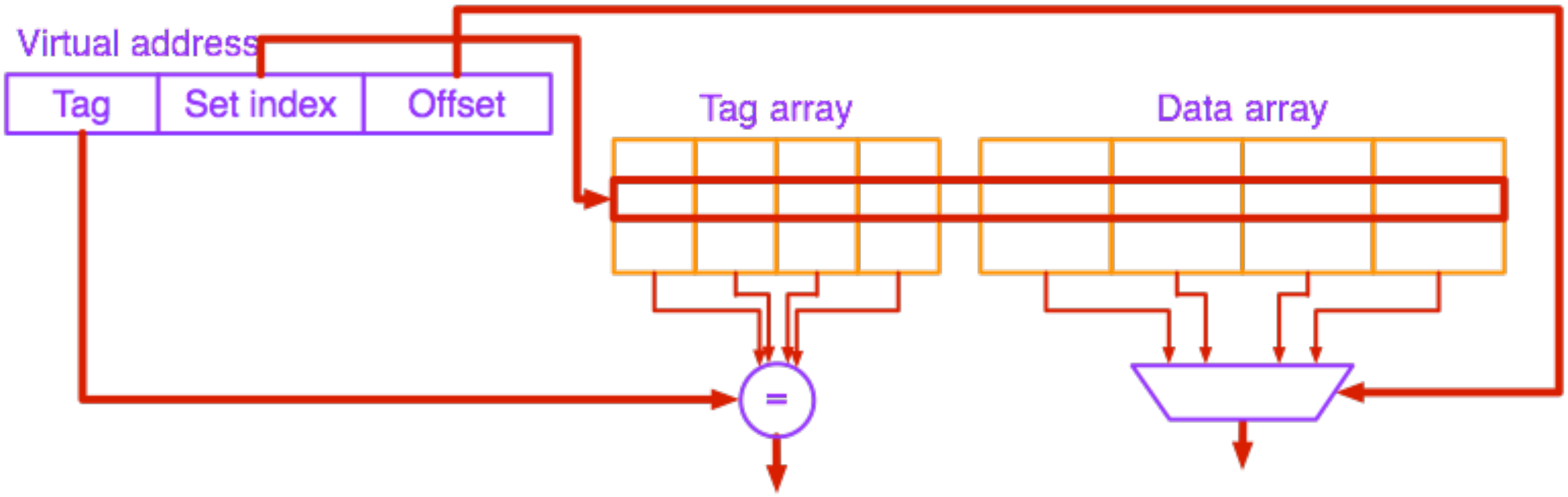

(4) PTE (overwrite an existing entry in the TLB)
TLB
(2) Virtual page number
CPU
(1) Virtual address
MMU
(3) PTE address
(5) Physical address
Main memory
(6) Data

Caches and Virtual Memory

- Virtually-addressed caches vs. physically-addressed caches
  - Which address do we send to the cache?
  - Virtually-addressed cache: faster (no address translation) but security issues (requires cache flushing by the OS on context switching)
  - Physically-addressed cache: slower but no security issues (no OS intervention)
- Four possible combinations
  - Physically indexed, physically tagged
  - Physically indexed, virtually tagged
  - Virtually indexed, physically tagged
  - Virtually indexed, virtually tagged

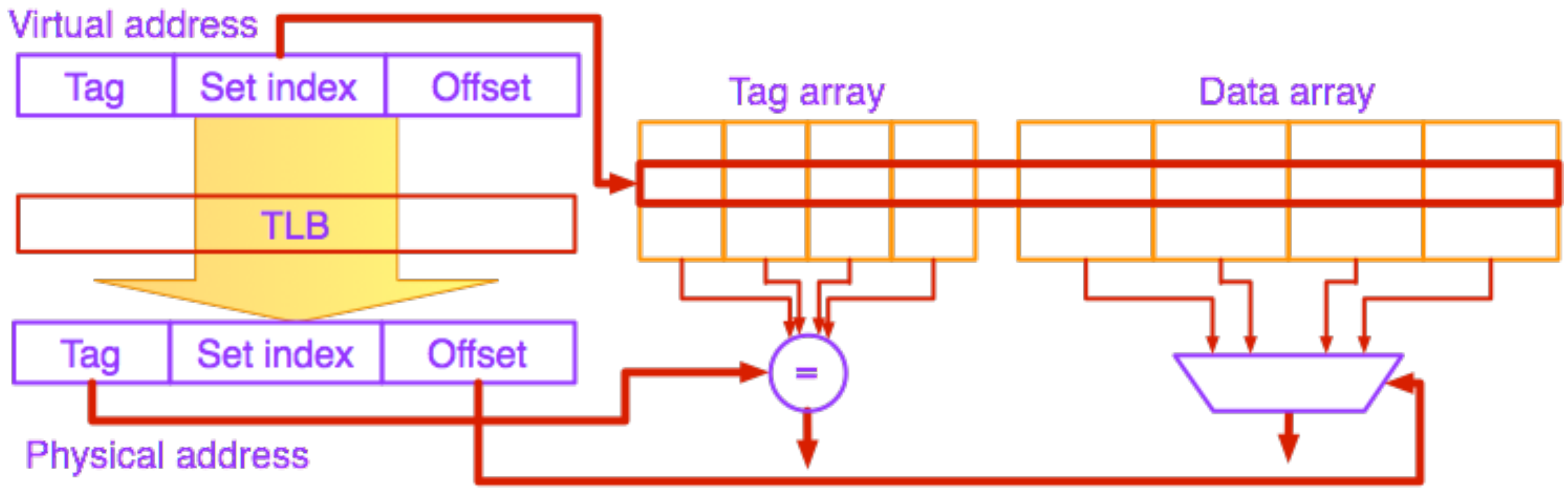
Virtually Indexed, Virtually Tagged

- Address translation occurs on a cache miss
- TLB (address translation) is not in the critical path



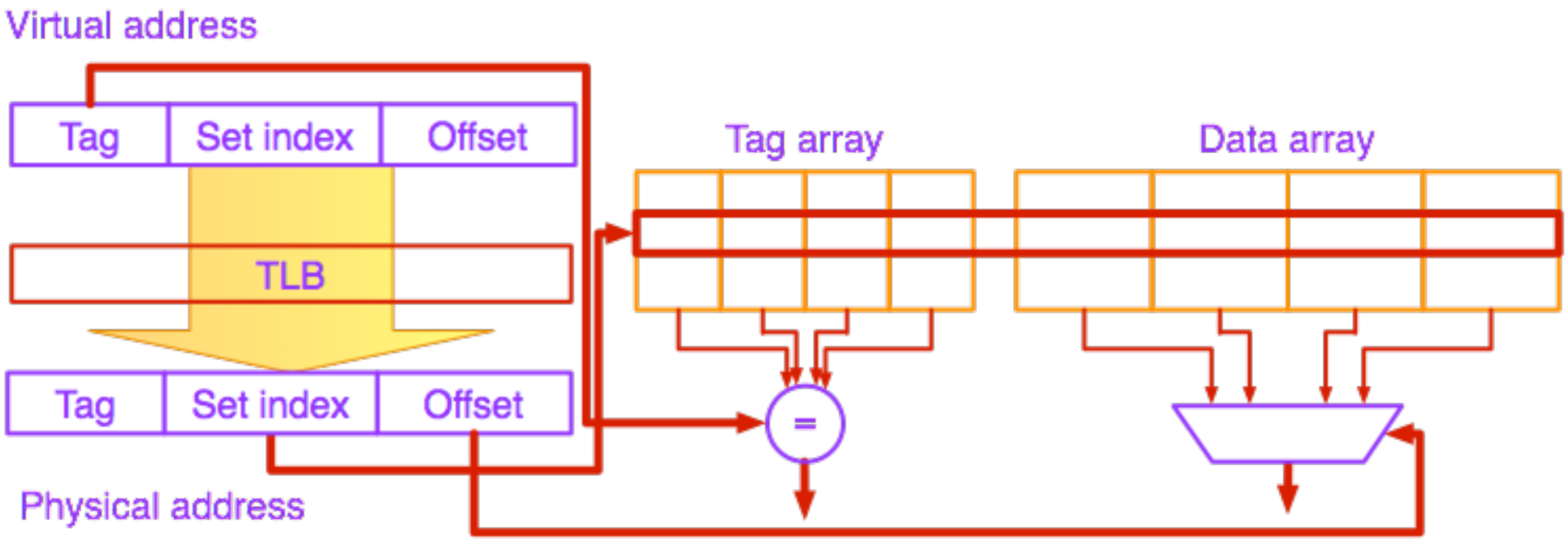
Virtually Indexed, Physically Tagged

- Common in real systems
- The address translation can happen at the same time as the cache indexing
- TLB is not in the critical path
- Much faster than physically-indexed caches



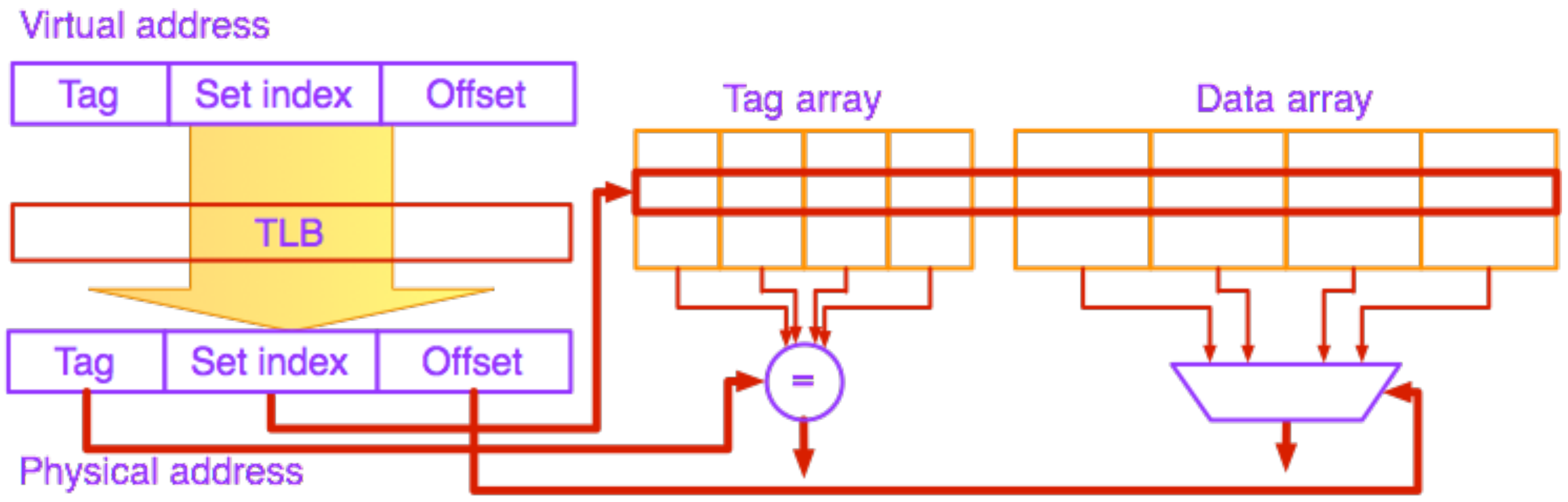
Physically Indexed, Virtually Tagged

- Never used
- No OS intervention for cache management
- TLB is in the critical path



Physically indexed, physically tagged

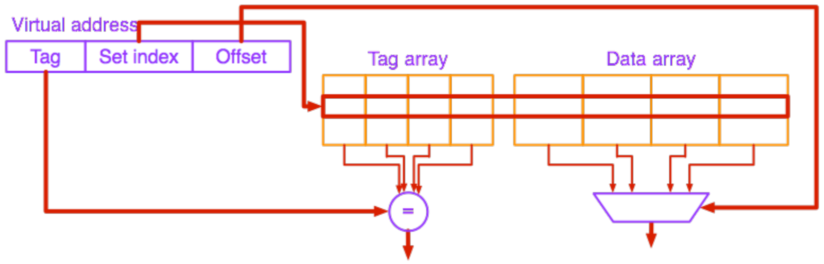
- No OS intervention for cache management
- TLB is in the critical path



Virtual Addressing vs. Physical Addressing

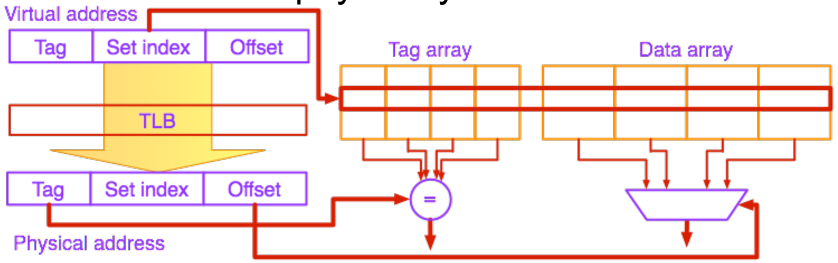
Virtually indexed, virtually tagged

- Address translation occurs on a cache miss
  - TLB (address translation) is not in the critical path



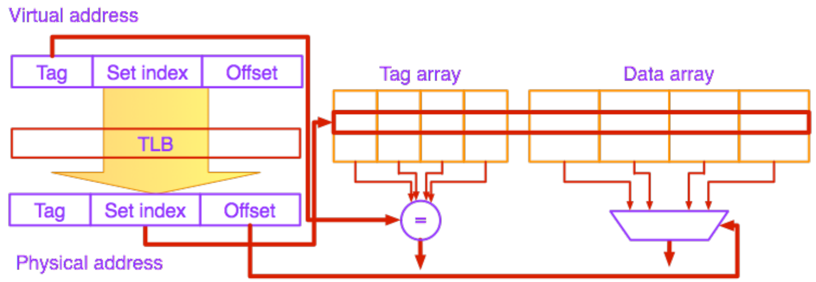
Virtually indexed, physically tagged

- Common in real systems
- The address translation can happen at the same time as the cache indexing
  - TLB is not in the critical path
- Much faster than physically-indexed caches



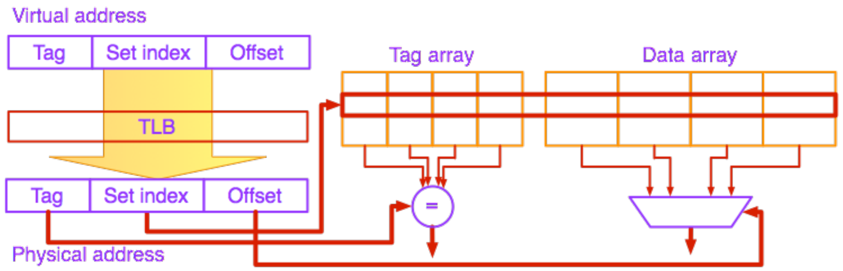
Physically indexed, virtually tagged

- Never used
- No OS intervention for cache management
- TLB is in the critical path



Physically indexed, physically tagged

- No OS intervention for cache management
- TLB is in the critical path


Virtual address
Tag
Set index
Offset
TLB
Tag
Set index
Offset
Physical address
Tag array
Data array
=