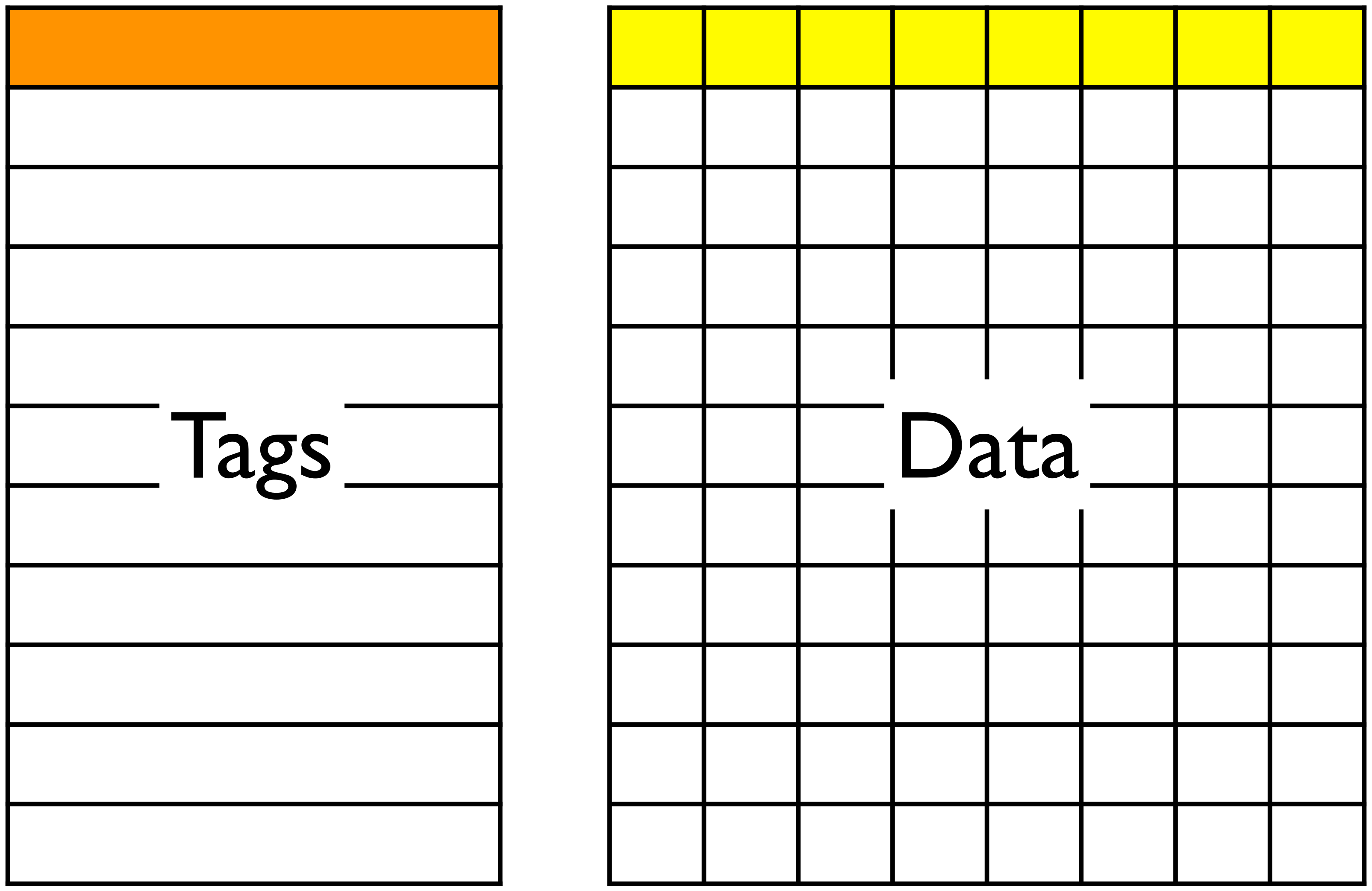
Cache Memories

- Introduced in 1950's by Wilkes ("slave store"), used in IBM S/360 in 1960's.

Cache Overview



Definitions

- Line — Unit of memory (2^n bytes)
  - Address uniquely refers to specific location in main memory
- Line Frame — Slot in cache used to hold a line
  - Location and presence in cache is dynamic

Motivations

- CPU cycle times decreasing faster than memory access times.
- Pragmatic limits on register file size
  - Too many registers -- slower access, longer context switch time
  - ISA limitations
- Locality of reference
  - High probability of finding needed data in smaller buffer over short interval -- *Working Set*

Required Features of a Cache

- Speed (high bandwidth, low latency)
- Fast lookup logic (hit / miss)
- Fast, automatic management
  - Fast but smart replacement algorithm
  - Placement policy
  - Write policy and write allocation policy

Intelligent Caches

- Dynamic tracking of program locality
  - Track and exploit temporal locality on-demand
  - Proactive prefetching

Cache Structure

- Cache line / block
  - Cache storage tracked in small units
    - 2^n for small n, e.g. 32 bytes
  - Main memory accessed in whole lines
    - Exploits spatial locality
  - Replacement algorithm ensures good tracking of locality

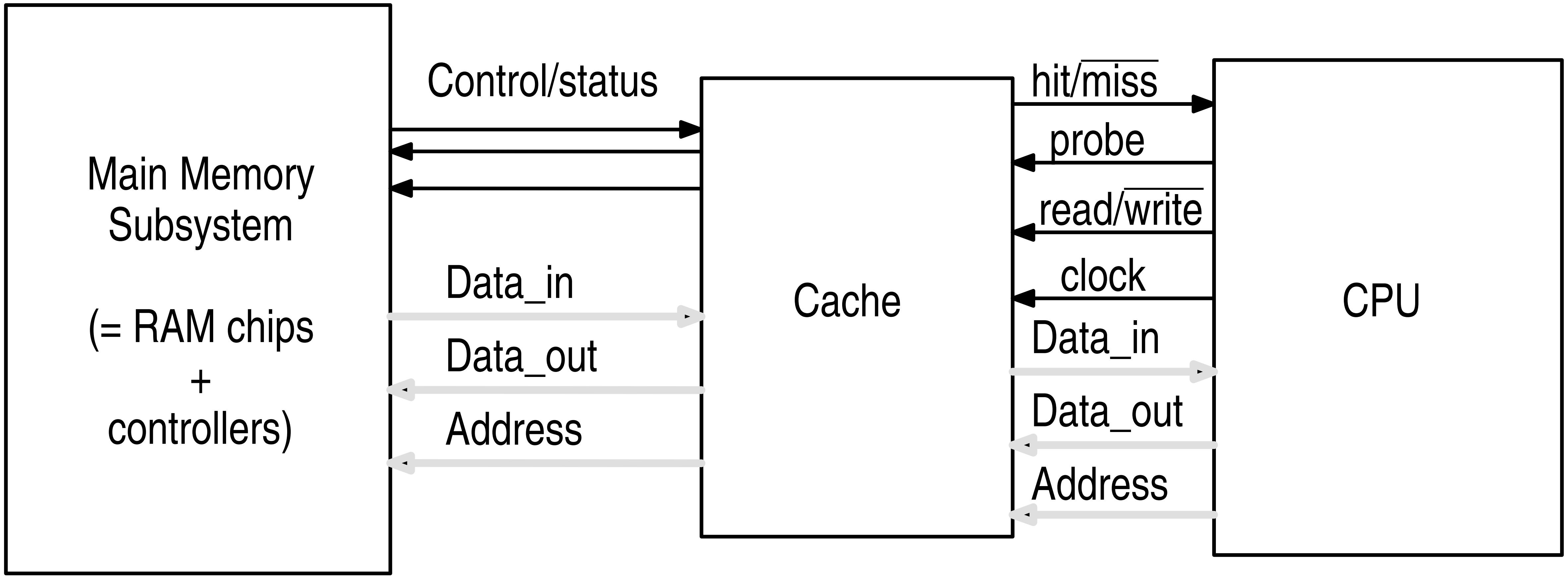
Transparency

- Cache must be invisible to software
  - Affects performance but not correctness, semantics, or behavior

Caches vs. registers:

Feature	Registers	Caches
Tracking of locality	Static, compiler can "look ahead"	Dynamically: based on past behavior
Expandability	Typically, none for architectural registers	Easy
How managed	Software – by the compiler	By hardware
ISA visibility	visible	Mostly invisible

Simple Cache–CPU Interface and Operation:



Cache Read Operation

- Probe from CPU:
  - Set address, read/write, and probe
  - After pipeline delay, get hit/miss response from cache
- Hit:
  - Cache has provided data. Done.
- Miss:
  - CPU stalls
  - Cache selects victim line
  - Flush if dirty, then invalidate (evict)
  - Fetch requested line from memory and forward requested data to CPU.

Cache Write Operation

- Probe from CPU:
  - Set address, **data**, read/write, and probe
  - After pipeline delay, get hit/miss response from cache
- Hit:
  - Data is stored in cache.
  - Write-back: Mark line dirty
  - Write-through: Forward data to DRAM
- Miss:
  - CPU stalls
  - Apply write allocation policy.....

Write Allocation Policy

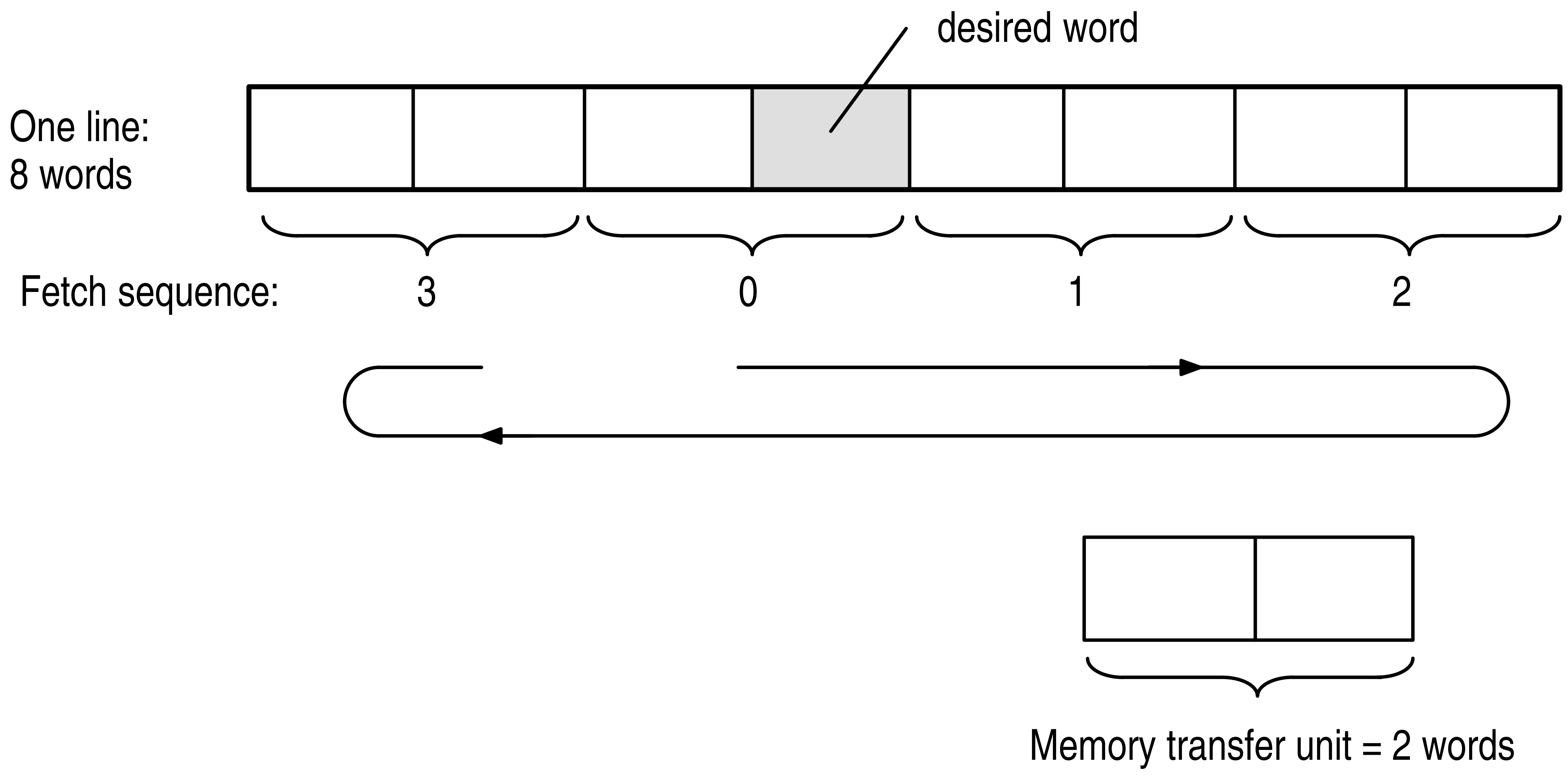
- Write-allocate:
  - Select victim line, flush if dirty, invalidate
  - Fetch requested line from DRAM
  - Update word being written
  - Forward to DRAM if write-through
- No-write-allocate
  - Forward write directly to DRAM

Cache Write Policies

- Write-through (store-through)
  - Main memory updated with cache
  - Preferred when main memory must be kept up to date
  - Can bottleneck at memory controller
- Write-back (copy-back)
  - Line marked dirty on write
  - Reduced bandwidth to main memory

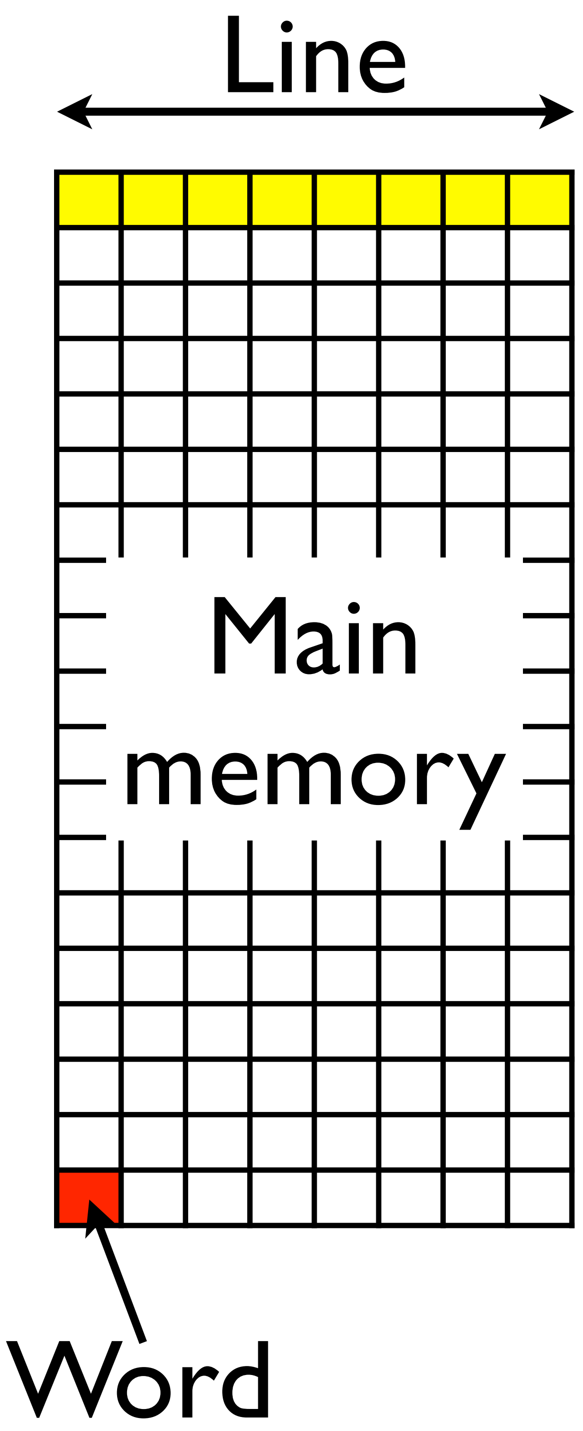
Cache Fetch Policies

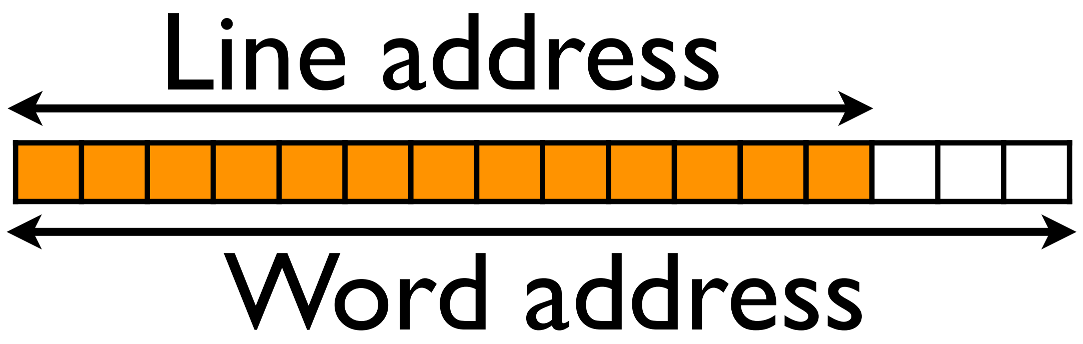
- Simple fetch -- Load entire line from memory, from the beginning
- Load-through (read-through)
- Wrap-around load:

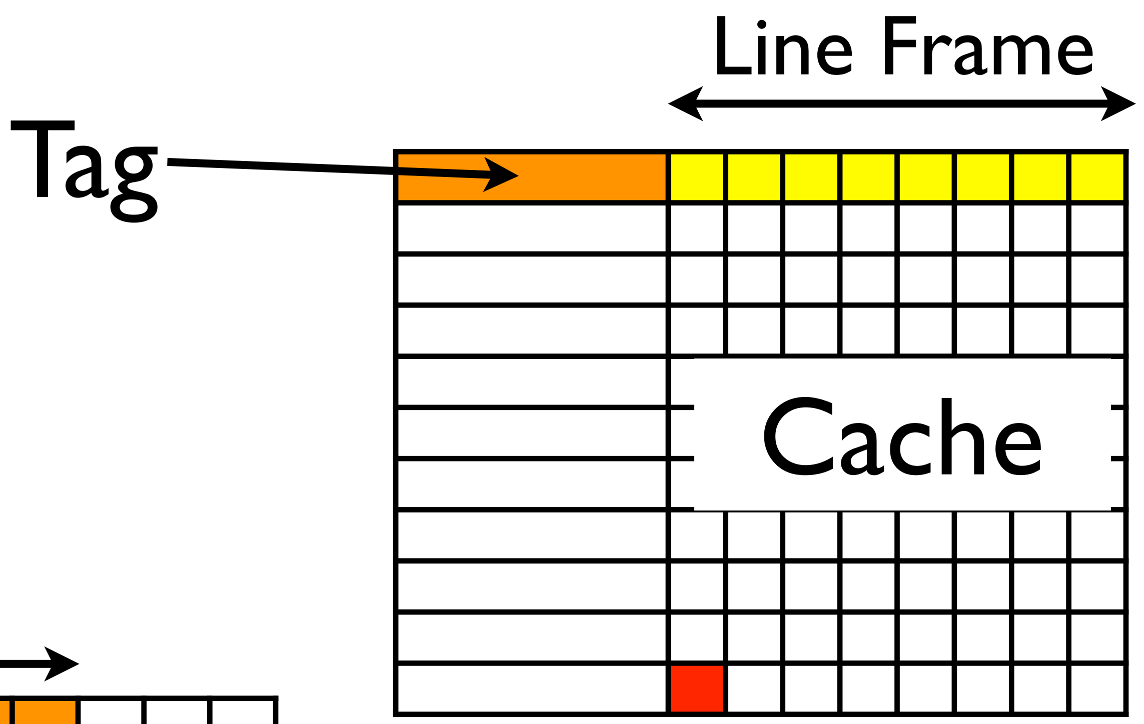


Chopping Up Memory

- Divide main memory into aligned blocks
  - 2^L words for small integer L
  - Called a “line”
- Lines stored in cache in “line frames”

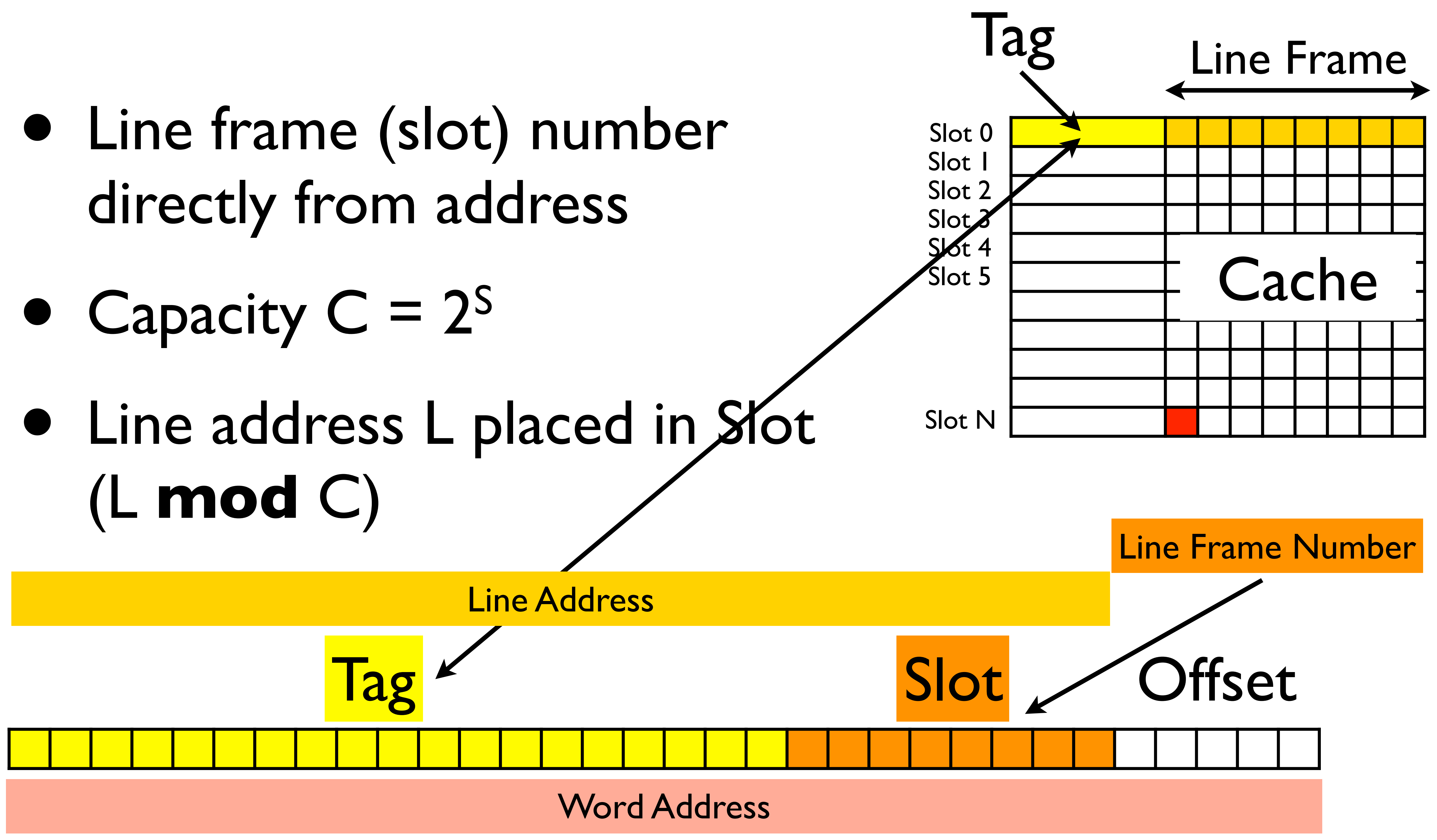






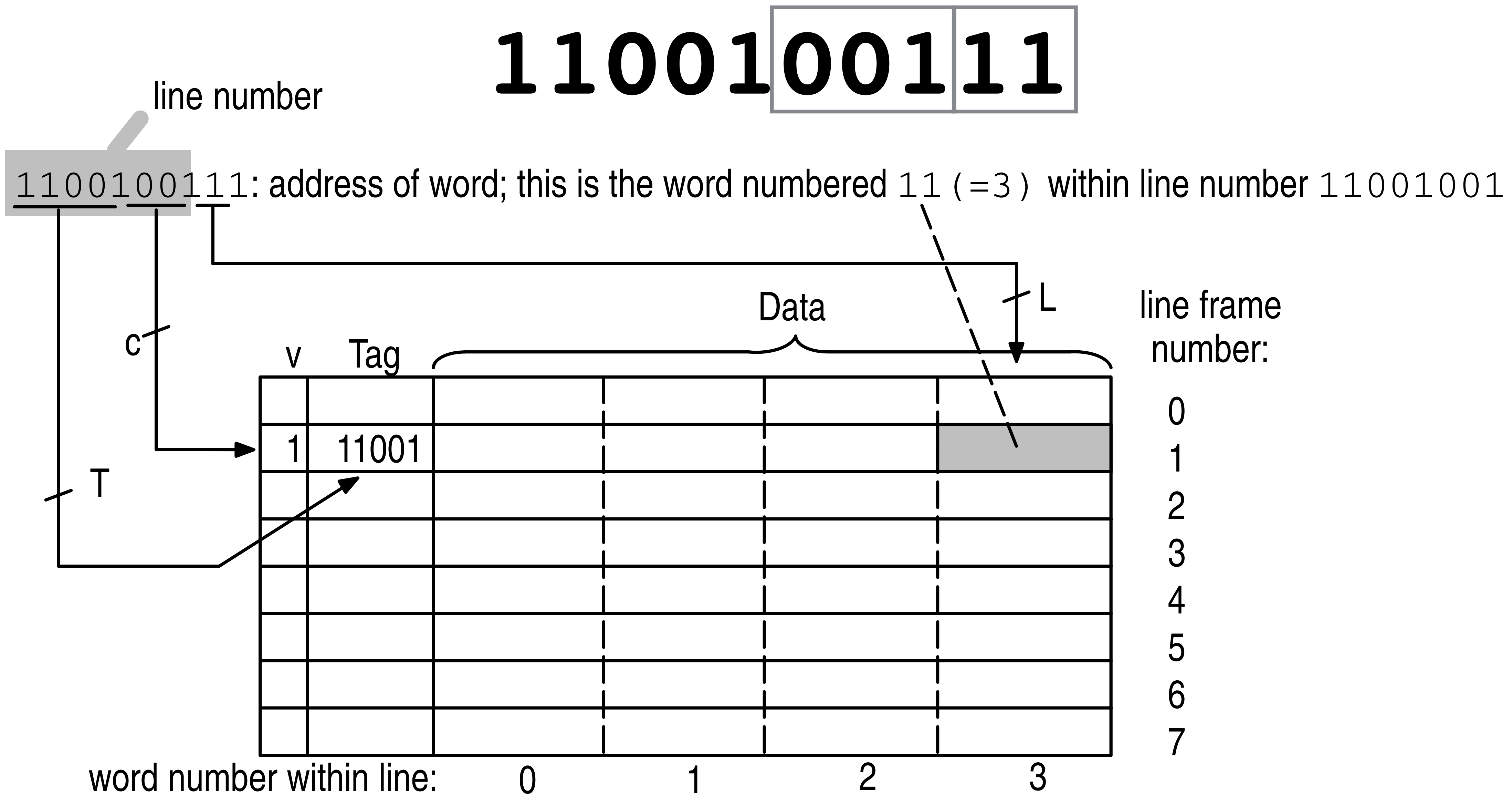
Direct Mapped Cache

- Line frame (slot) number directly from address
- Capacity $C = 2^S$
- Line address L placed in Slot (L **mod** C)

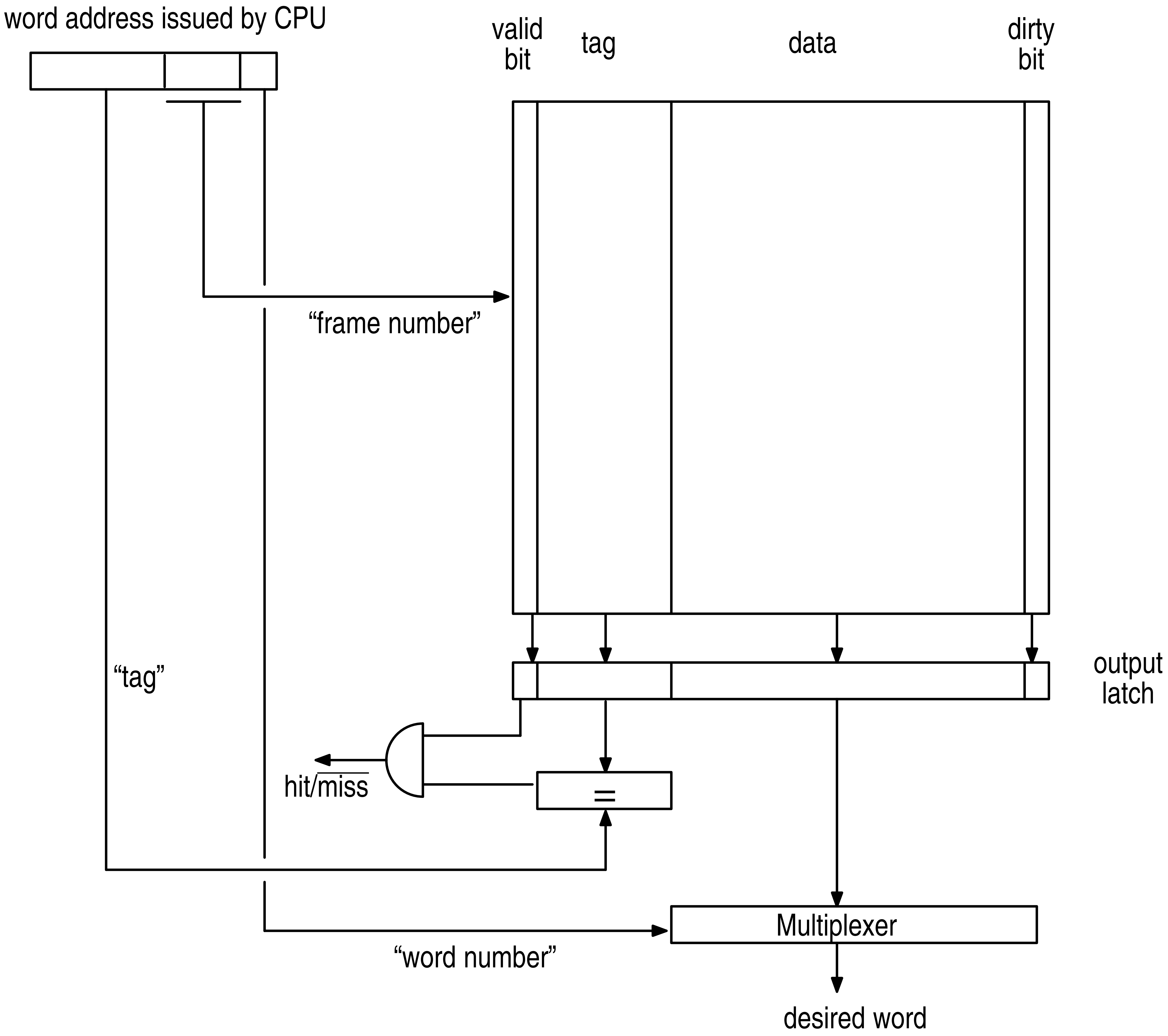


- An example – small cache (unrealistic), but shows the major ideas:

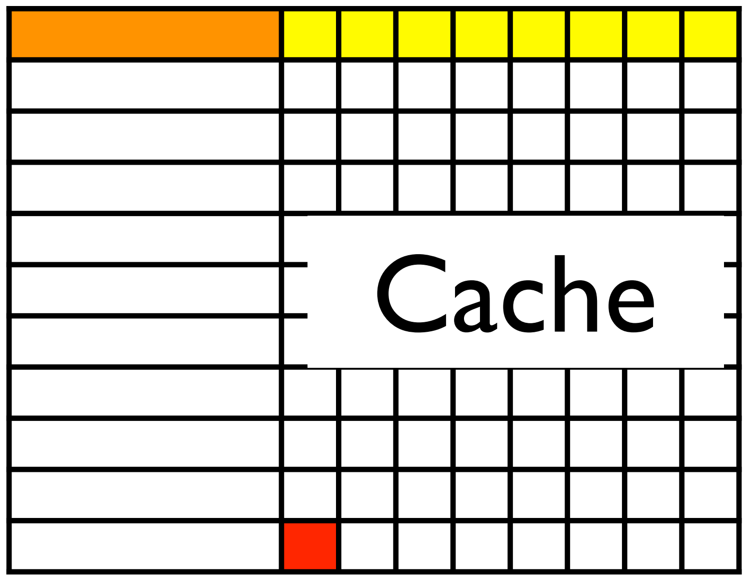
  Number of line frames in cache = 8 (=> *c* = 3)
  Number of bits in memory address of a word, *N* = 10 (say)
  Number of words per line = 4 (=> *L* = 2)
  Size of tag field = *N* – c – *L* bits = 5–bits
  Value of tag field = leading *T* = (*N* – c – *L*) bits of the line address



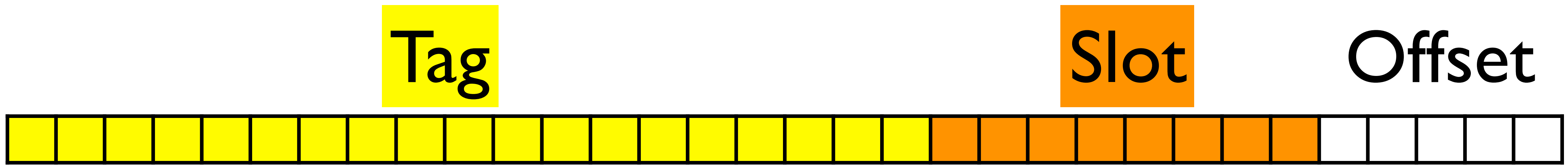
A cached word and its associated tag in the example direct–mapped cache


word address issued by CPU
valid bit
tag
data
dirty bit
"frame number"
output latch
"tag"
hit/$\overline{\text{miss}}$
=
Multiplexer
"word number"
desired word

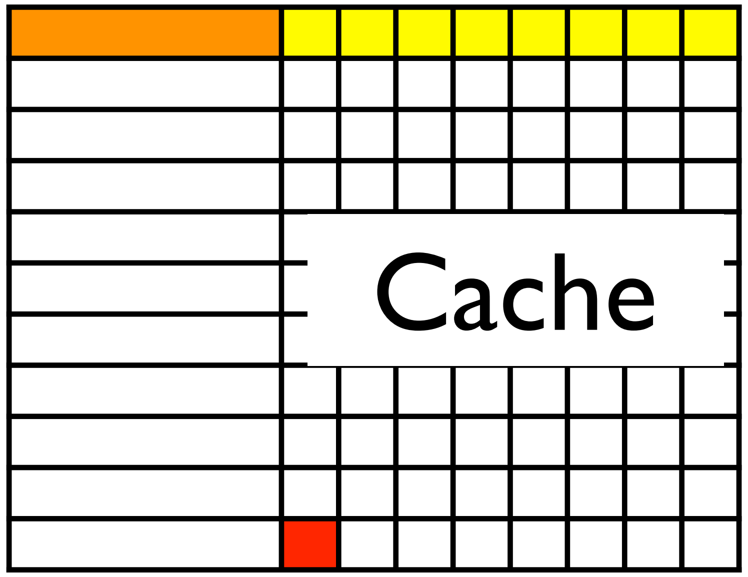
Direct-Mapped Read



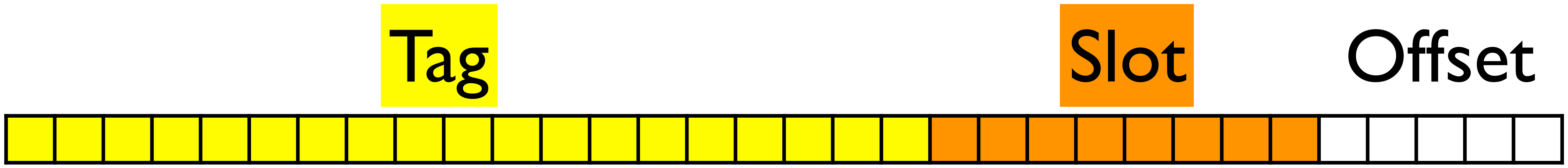
- **Read:** *Latch* ← *Cache[slot]*
- **Compare:** *(Latch.tag == Address.tag) && Latch.valid ?*
- **Steer:** *Latch.words[offset]* → CPU



Direct-Mapped Write

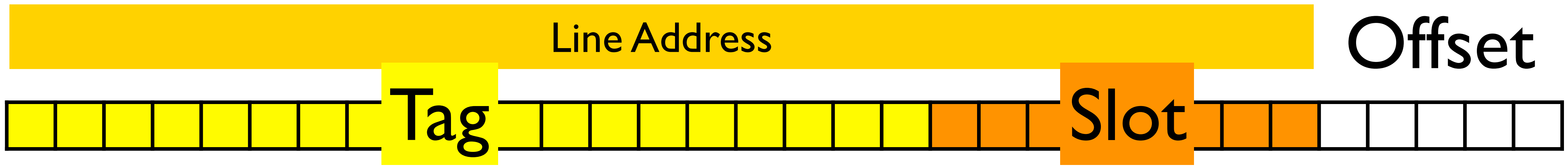


- **Read:** *Latch ← Cache[slot]*
- **Compare:** *(Latch.tag == Address.tag) && Latch.valid ?*
- On hit:
  - **Update:** *Latch.words[offset] ← data*
  - **Update:** *Cache[slot] ← Latch*

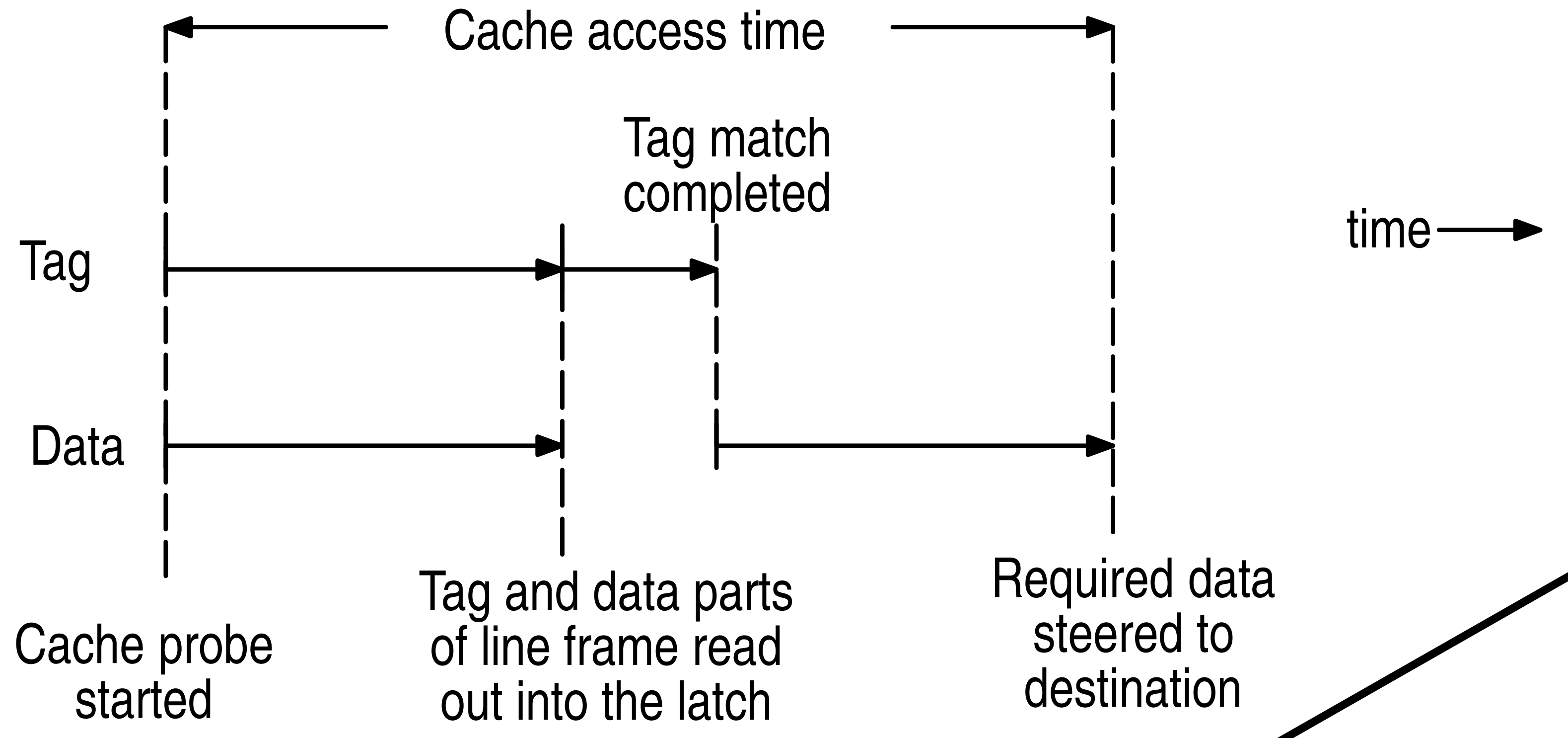


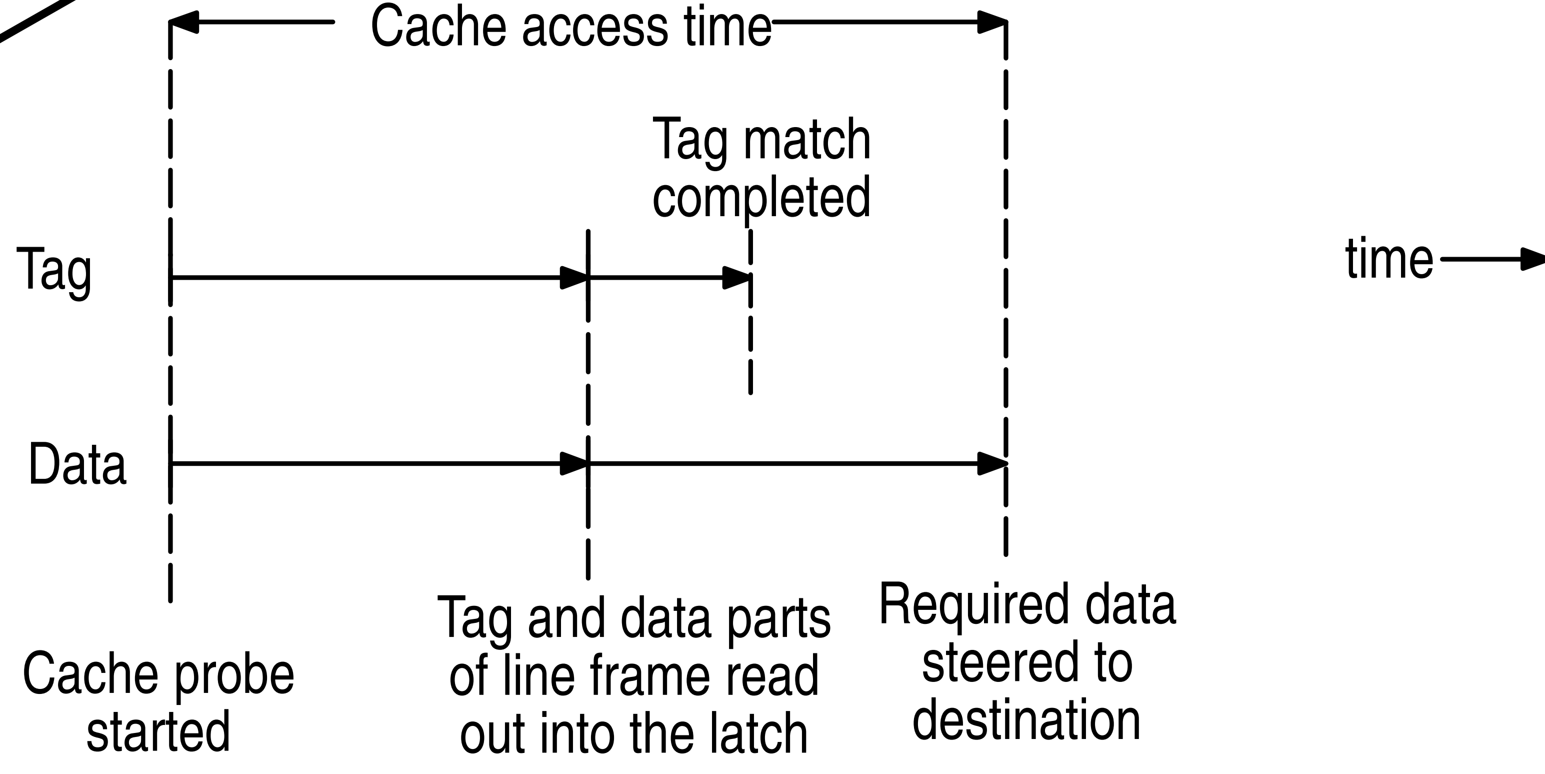
Direct-Mapped Miss-Handling

- **Read:** *Latch* ← *Cache[slot]*
- **Evict:** If *(Latch.valid && Latch.dirty)*, flush line to *Memory[{Latch.tag, slot}]*
- **Tag:** *Latch.tag* ← *Address.tag*
- **Valid:** *Latch.valid* ← *true*
- **Load:** Read line (2^S words)
  - *Memory[Address.line]* → *Latch*
- Read or write word
- **Store:** *Cache[slot]* ← *Latch*



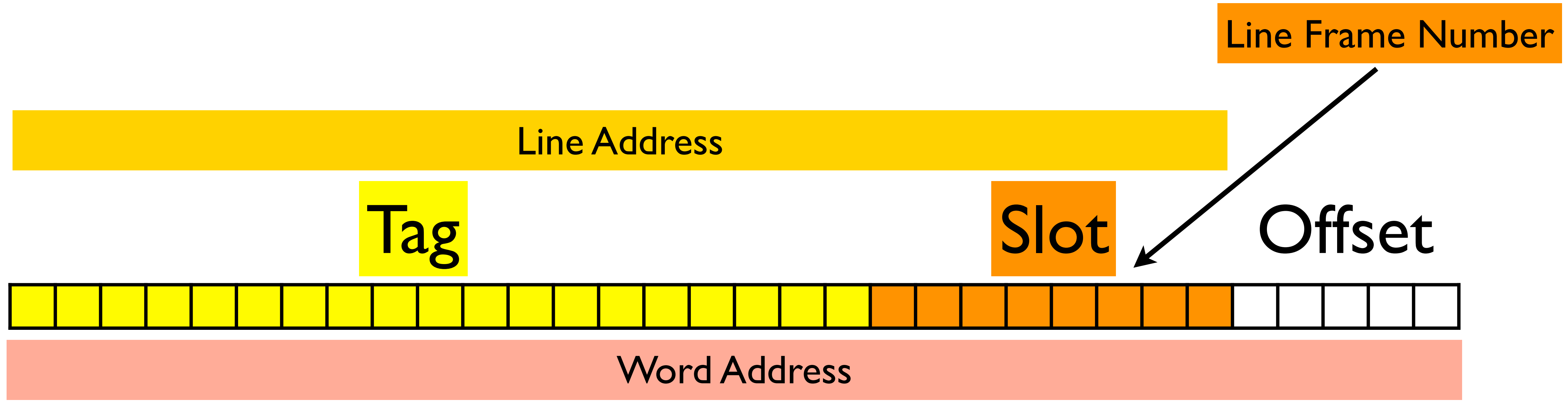
Cache Access Time


Cache access time
Tag match completed
time
Tag
Data
Cache probe started
Tag and data parts of line frame read out into the latch
Required data steered to destination


Cache access time
Tag match completed
time
Tag
Data
Cache probe started
Tag and data parts of line frame read out into the latch
Required data steered to destination

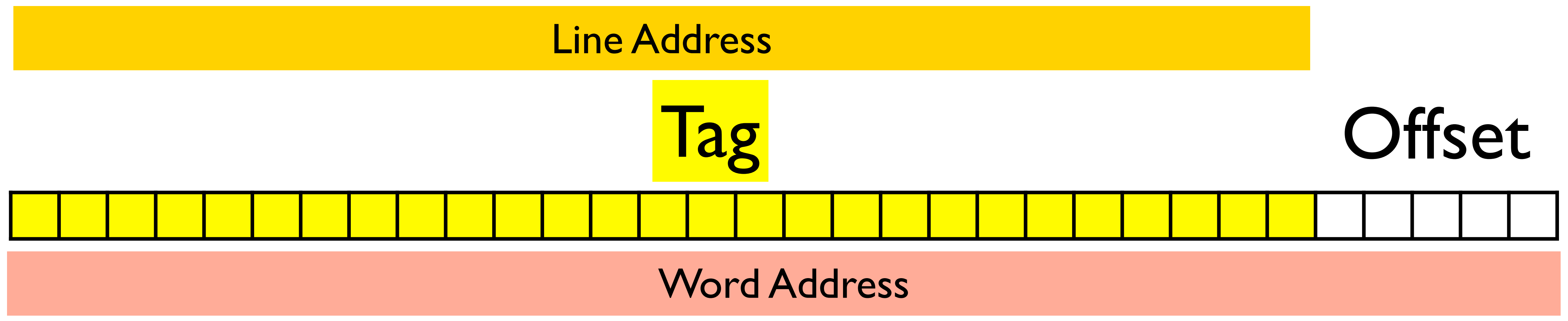
Direct Mapped Cache

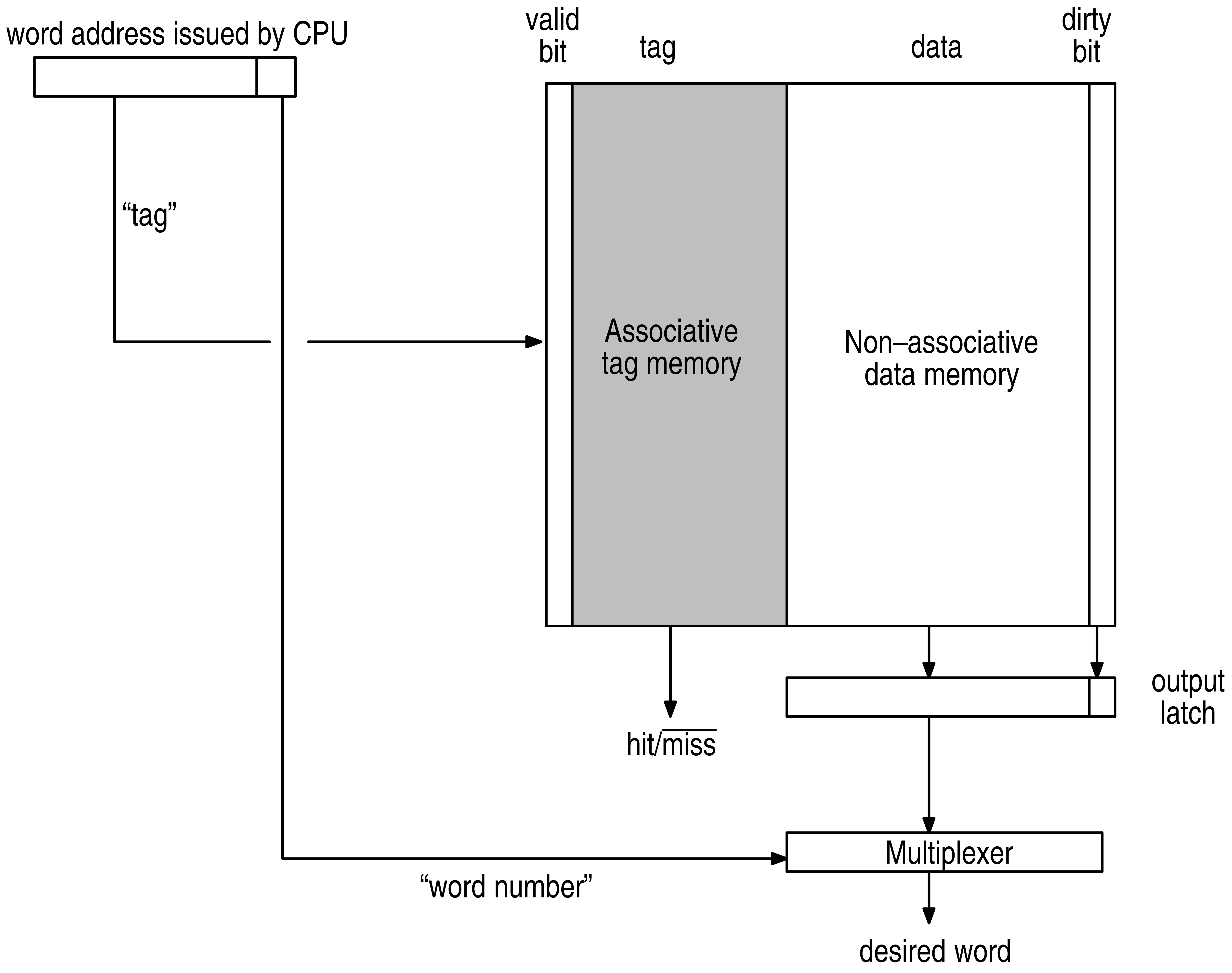
- Line frame (slot) number directly from address
- Capacity C = 2^S
- Line address L placed in Slot (L **mod** C)



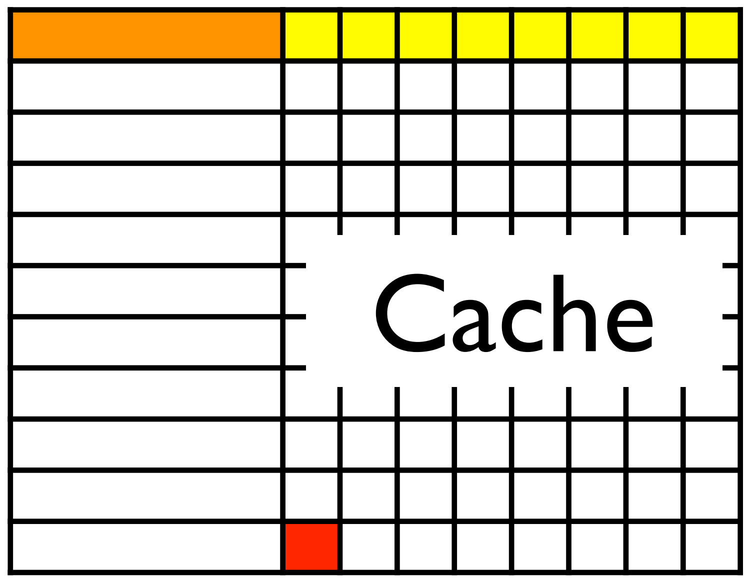
Fully-Associative Cache

- Memory line may be placed in **any** line frame
- More freedom to choose more optimal victims
- Arbitrary capacity
- Lookup very expensive

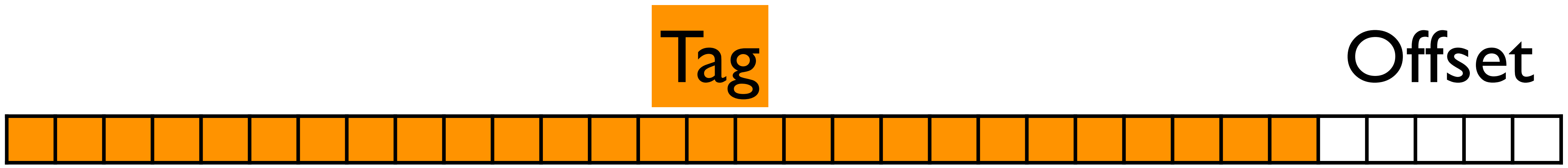



word address issued by CPU
"tag"
"word number"
valid bit
tag
data
dirty bit
Associative tag memory
Non–associative data memory
hit/$\overline{\text{miss}}$
output latch
Multiplexer
desired word

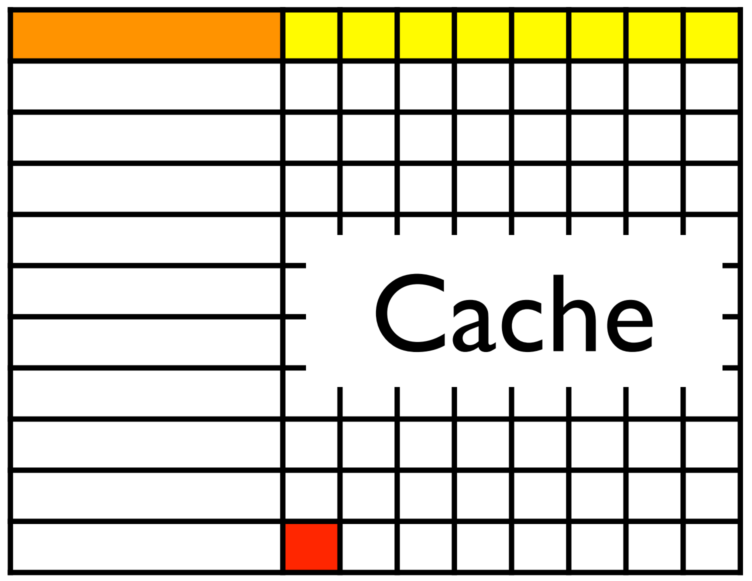
Fully-Associative Read



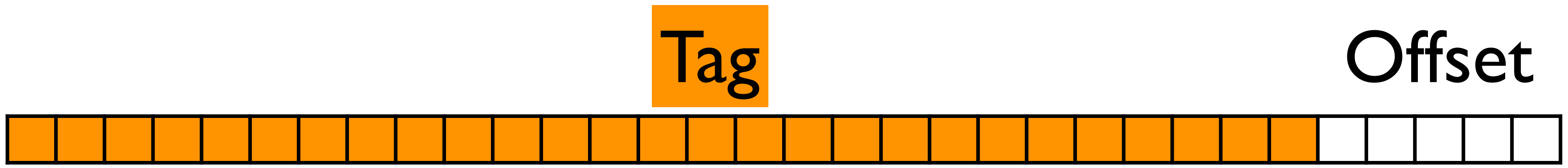
- **Match:** Compare *Address.tag* to every cache tag in parallel (expensive!)
- **Read:** *Latch* ← *Cache[slot]*
- **Steer:** *Latch.words[offset]* → CPU
- Update statistics



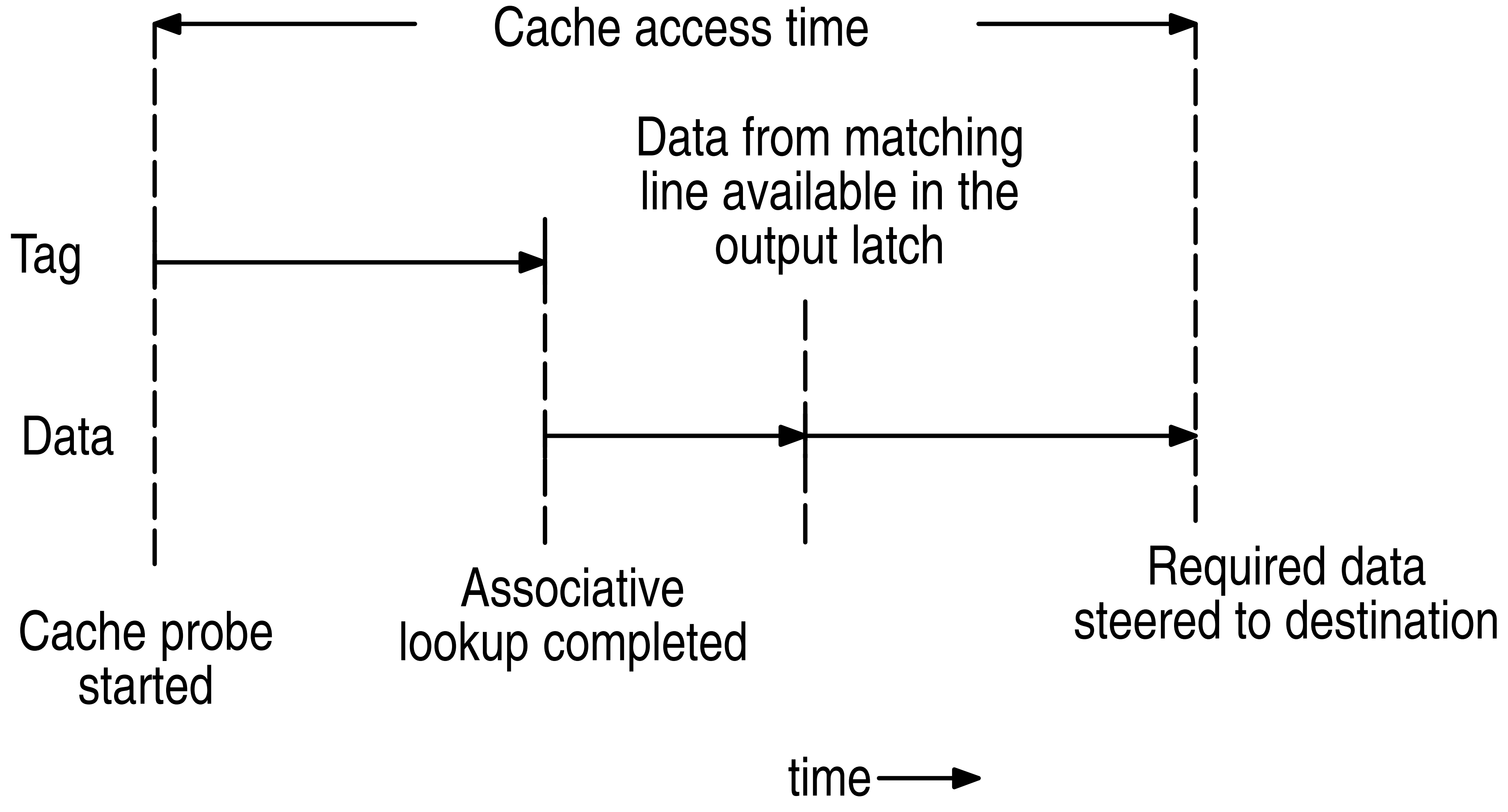
Fully-Associative Write



- **Match:** Compare *Address.tag* to every cache tag in parallel
- **Read:** *Latch* ← *Cache[slot]*
- On hit:
    - **Update:** *Latch.words[offset]* ← *data*
    - Update Statistics
    - **Update:** *Cache[slot]* ← *Latch*



- Timing issues: In the fully–associative cache, the data can be steered only after the associative lookup is completed:

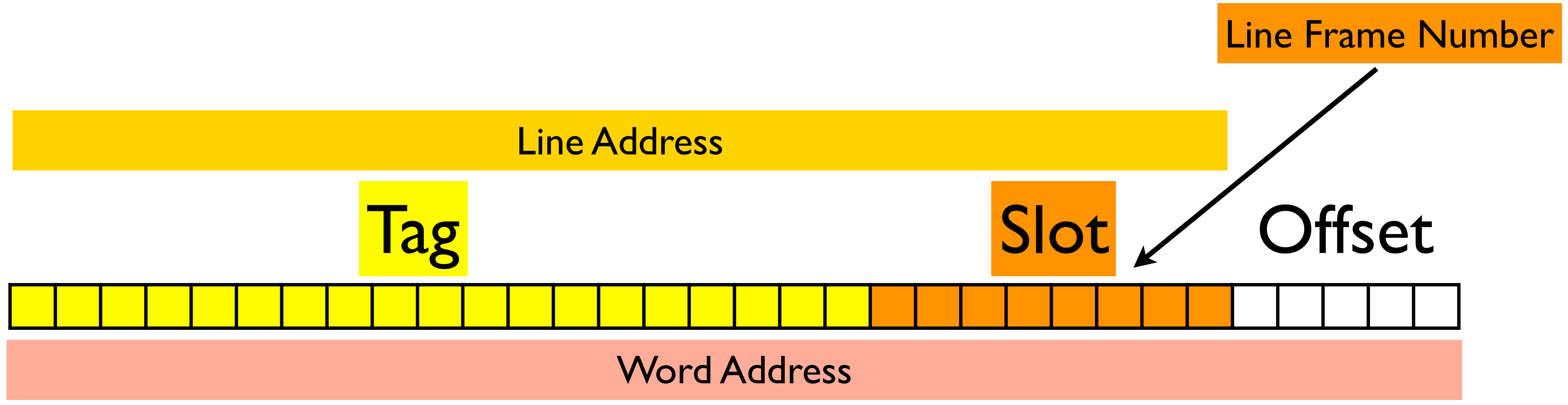


Cache access time
Data from matching line available in the output latch
Tag
Data
Cache probe started
Associative lookup completed
Required data steered to destination
time

Demo

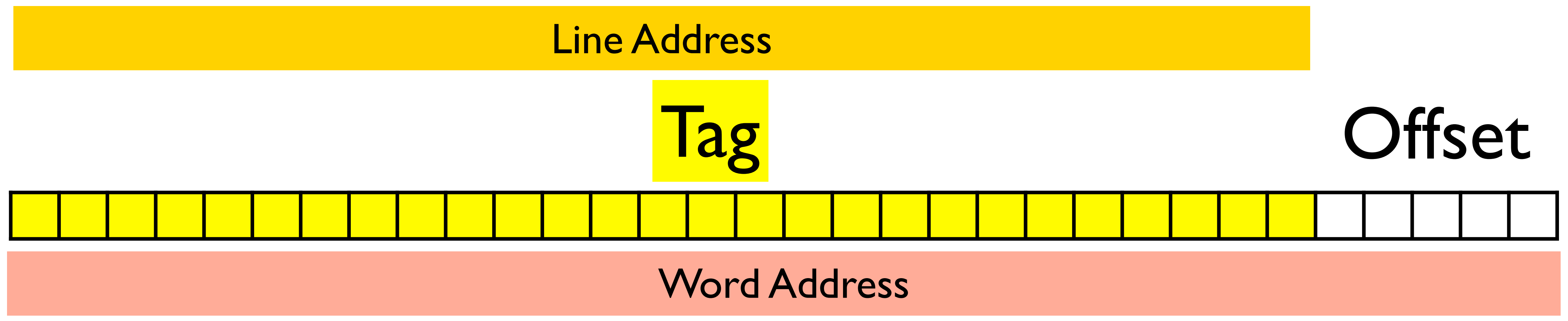
Recap: Direct Mapped Cache

- Line frame (slot) number directly from address
- Capacity C = 2^S
- Line address L placed in Slot (L **mod** C)



Recap: Fully-Associative Cache

- Memory line may be placed in **any** line frame
- More freedom to choose more optimal victims
- Arbitrary capacity
- Lookup very expensive



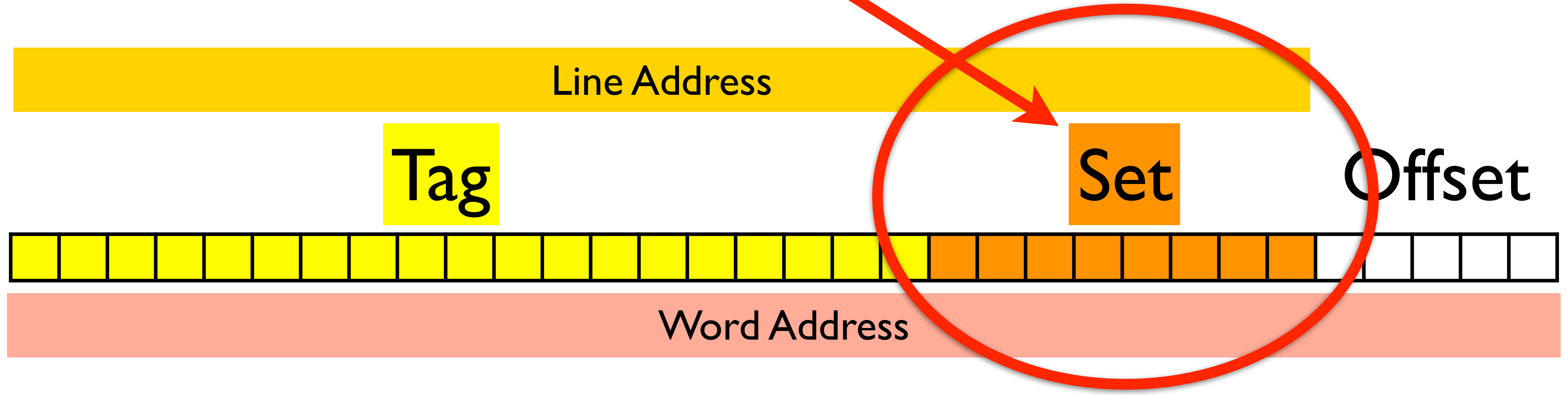
The set–associative cache:

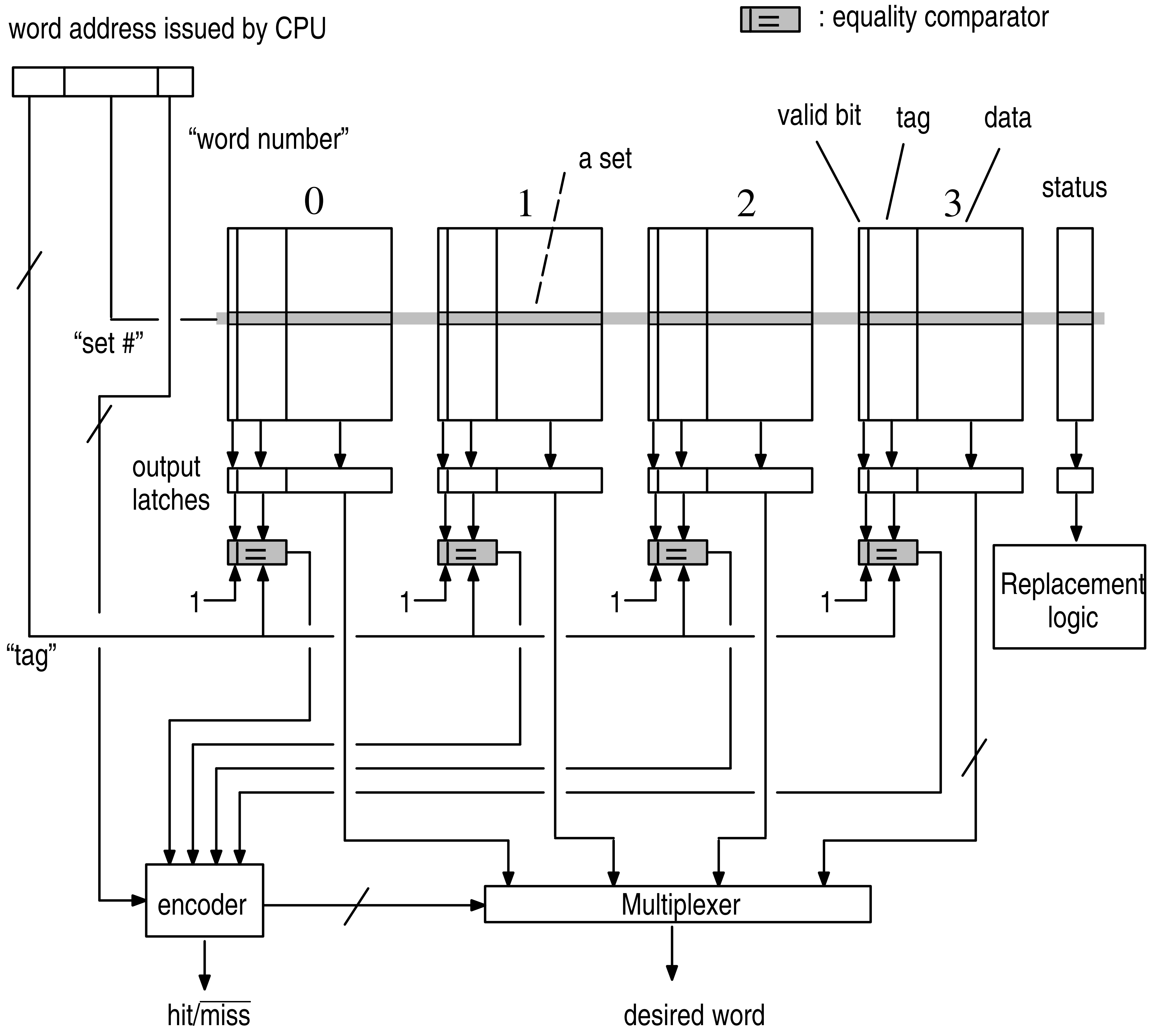
- The design of the set–associative cache is motivated by the contrasting features of the direct–mapped cache and the fully–associative cache:

Feature	Direct–mapped cache	Fully–associative cache
cycle time	fast	relatively slower
amount of hardware needed beyond a RAM	small: latch + comparator + multiplexer	substantial: associative logic + latch + multplexer + replacement logic
hit ratio	low for small sizes	high, even for small capacities

Set Associative Caches

- Combine **P** direct-mapped caches into one
- Set number is slot number for each sub-cache.
- P lines frames with the same frame number make up a ***set***.
- Any one can hold any line with same set #.
- With C=2^S, capacity is C×P





A 4–way set–associative cache

Set Associative Benefits

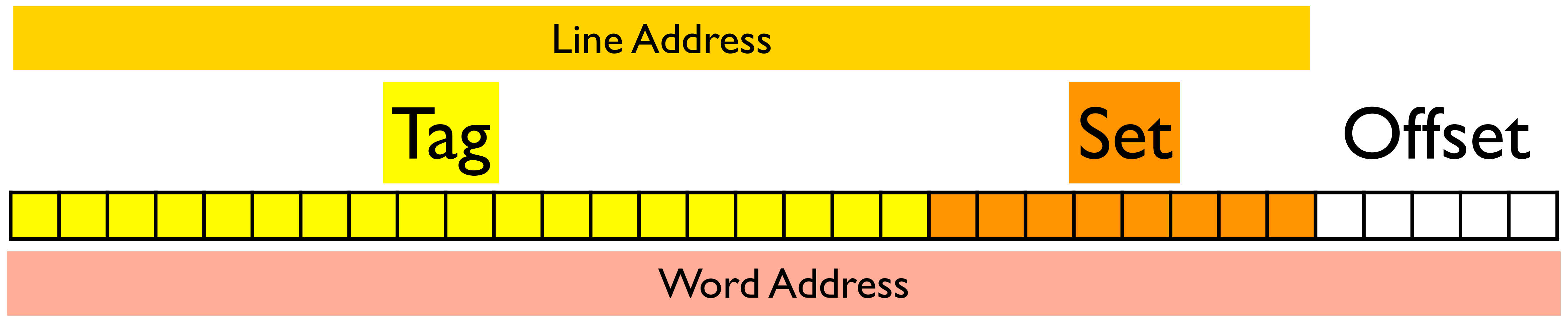
- More freedom to place lines into line frames
- More freedom to select victims
- Higher hit ratio, approaching fully associative.
- Fast access time, approaching direct mapped.

Choices

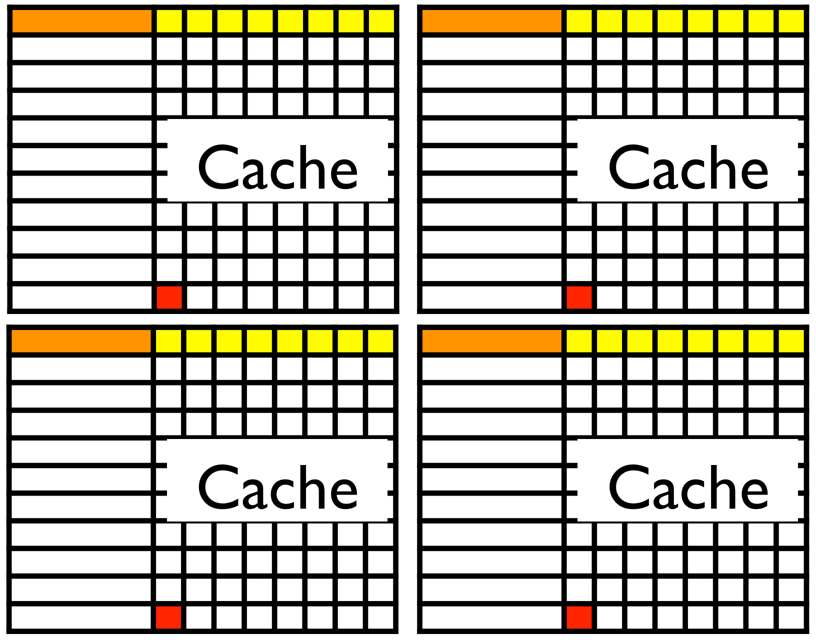
- Placement policy: **P** choices
- Replacement policy: **P** potential victims
- To make intelligent choice, must keep statistics

Set Associative Lookup

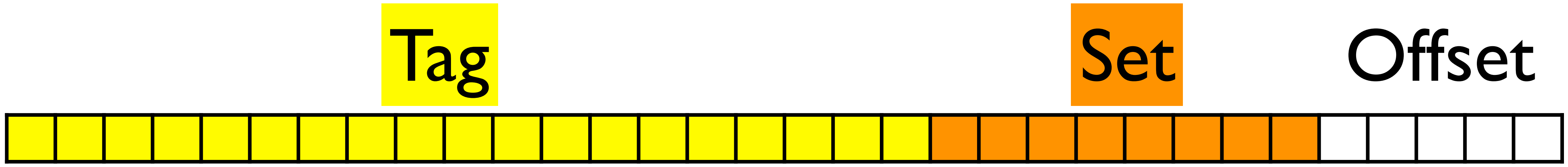
- Direct lookup of *Cache[Set]* in each sub-cache
- Fully associative compare *Address.Tag* to every tag in set.



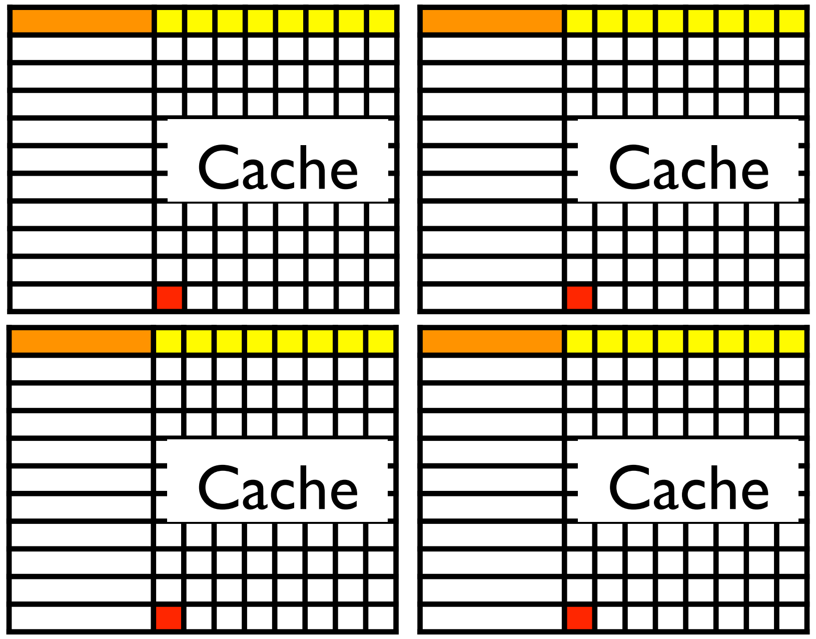
Set Associative Read



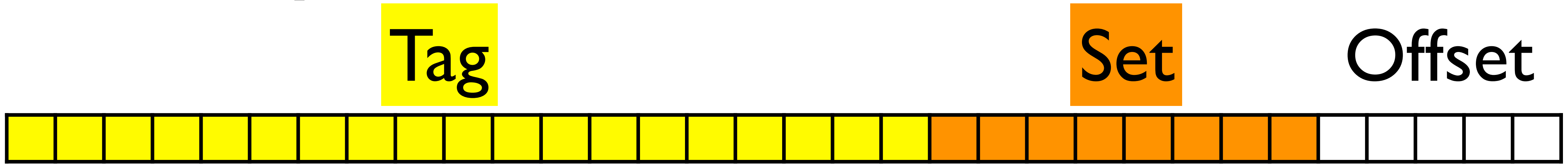
- **Read:**
  - *Latch[0] ← Cache[0][set]*
  - *Latch[1] ← Cache[1][set]*
  - *Latch[2] ← Cache[2][set]*
  - *Latch[3] ← Cache[3][set]*
- **Compare:**
  - *(Latch0.tag == Address.tag) && Latch0.valid ?*
  - *(Latch1.tag == Address.tag) && Latch1.valid ?*
  - *(Latch2.tag == Address.tag) && Latch2.valid ?*
  - *(Latch3.tag == Address.tag) && Latch3.valid ?*
- **Steer:** *Latch[match].words[offset]* → CPU



Set Associative Write



- **Read:**
  - *Latch[0] ← Cache[0][set]*
  - *Latch[1] ← Cache[1][set]*
  - *Latch[2] ← Cache[2][set]*
  - *Latch[3] ← Cache[3][set]*
- **Compare:**
  - *(Latch0.tag == Address.tag) && Latch0.valid ?*
  - *(Latch1.tag == Address.tag) && Latch1.valid ?*
  - *(Latch2.tag == Address.tag) && Latch2.valid ?*
  - *(Latch3.tag == Address.tag) && Latch3.valid ?*
- On hit:
  - **Update:** *Latch[match].words[offset] ← data*
  - **Update:** *Cache[match][set] ← Latch[match]*

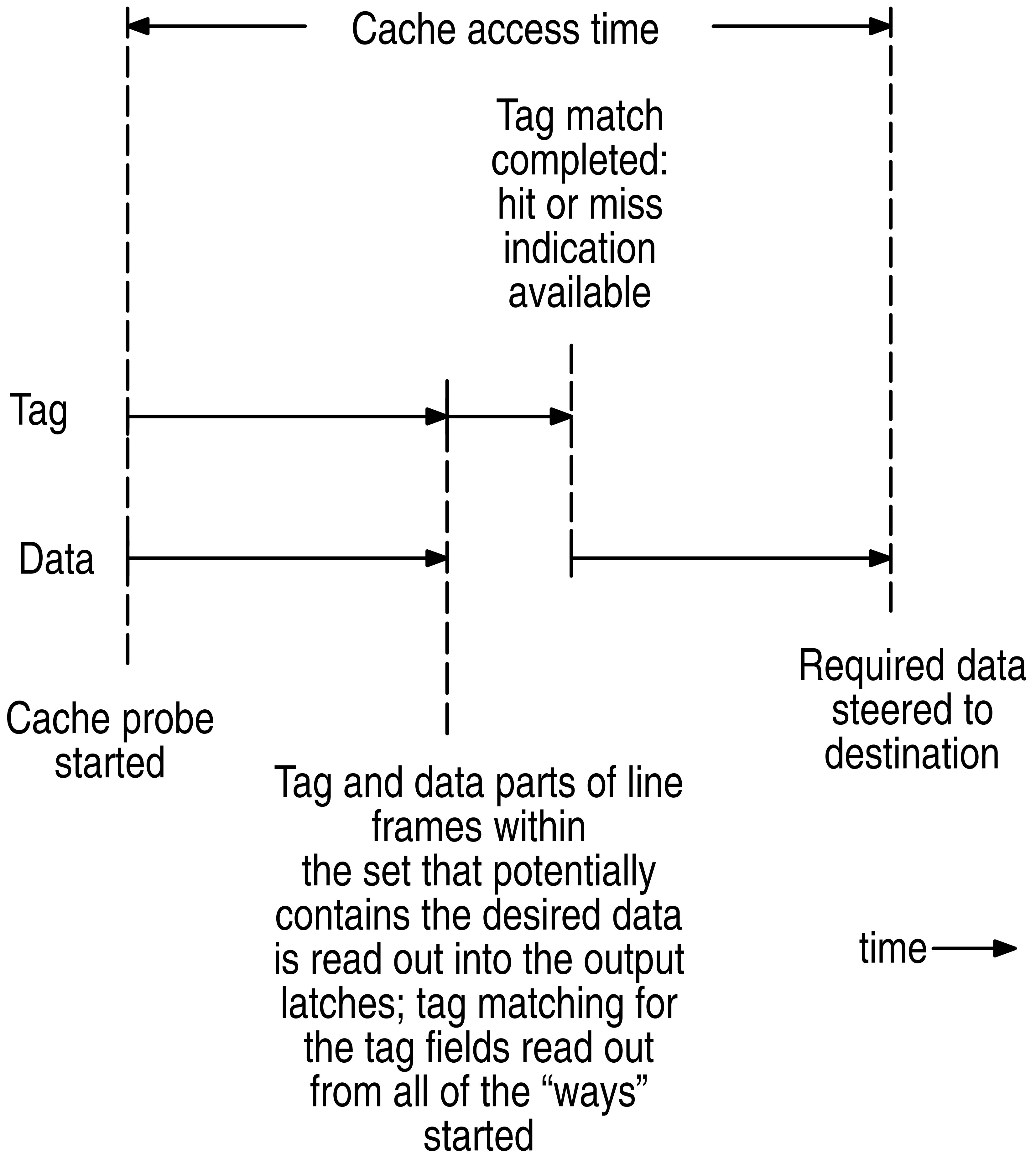


- Note the two extremes of a p–way set–associative cache:

  <u>$p = 1$</u>: this is a direct–mapped cache

  <u>$p = C$</u>: this is a fully–associative cache.

Verilog Source for Set Associative Cache



Cache access time
Tag match completed: hit or miss indication available
Tag
Data
Cache probe started
Tag and data parts of line frames within the set that potentially contains the desired data is read out into the output latches; tag matching for the tag fields read out from all of the “ways” started
Required data steered to destination
time

<u>A simple analysis</u>: effective memory access time with the cache, T_{eff}

- Assumes:
  - Time to detect a cache hit or miss = t_c
  - Additional time for a read hit = 0
  - Additional time needed for a write on a hit = t_w (total time for a write hit is thus $t_c + t_w$)
  - Cache does not overlap steps of consecutive accesses
  - hit ratio (for both read & write) = h
  - cache is write through with write allocate; on a write miss, memory is first updated and the missing line is then read into the cache
  - time to update a word in memory = t_M
  - time to fetch a line from memory = t_L
  - time for selecting a victim is overlapped with cache probe
  - fraction of write accesses = *w*

- The table given below shows the timings in the four different cases hat arise on a cache access:

access type / outcome –>	cache hit: probability = h	cache miss: probability = (1 – h)
read access (probability = 1 – *w*)	t_c	$t_c + t_L$
write access (probability = *w*)	$t_c + t_w$	$t_c + t_M + t_L$

- Weighting the timing figures in the above table with the appropriate probabilities for the cases, we get:

$$
\begin{aligned}
T_{eff} &= t_c.h.(1 - w) + (t_c + t_L).(1 - h).(1 - w) + (t_c + t_w).h.w \\
&\quad + (t_c + t_M + t_L).(1 - h).w \\
&= t_c + (1 - h).t_L + h.w.t_w + (1 - h).w.t_M \qquad (1)
\end{aligned}
$$

– Note a few obvious things: T_{eff} comes down as h increases or as w decreases, and in general, as the miss handling times come down.

- Lets now plug in some typical values, with times in *CPU cycles*:

  $t_c = 1, t_w = 1, t_M = 4, t_L = 16$
  $w = 0.3$

  This gives:

$$\begin{aligned} T_{eff} &= 1 + (1 - h) * 4 + h * 0.3 + (1 - h) * 4.8 \\ &= 9.8 - 8.5 * h \qquad (2) \end{aligned}$$

- When h = 0.95 (this is easily achieved in practice for typical caches used)

  T_{eff} = 1.725

- When h = 0.97 (this is again easily achieved in practice for typical caches used)

  T_{eff} = 1.555

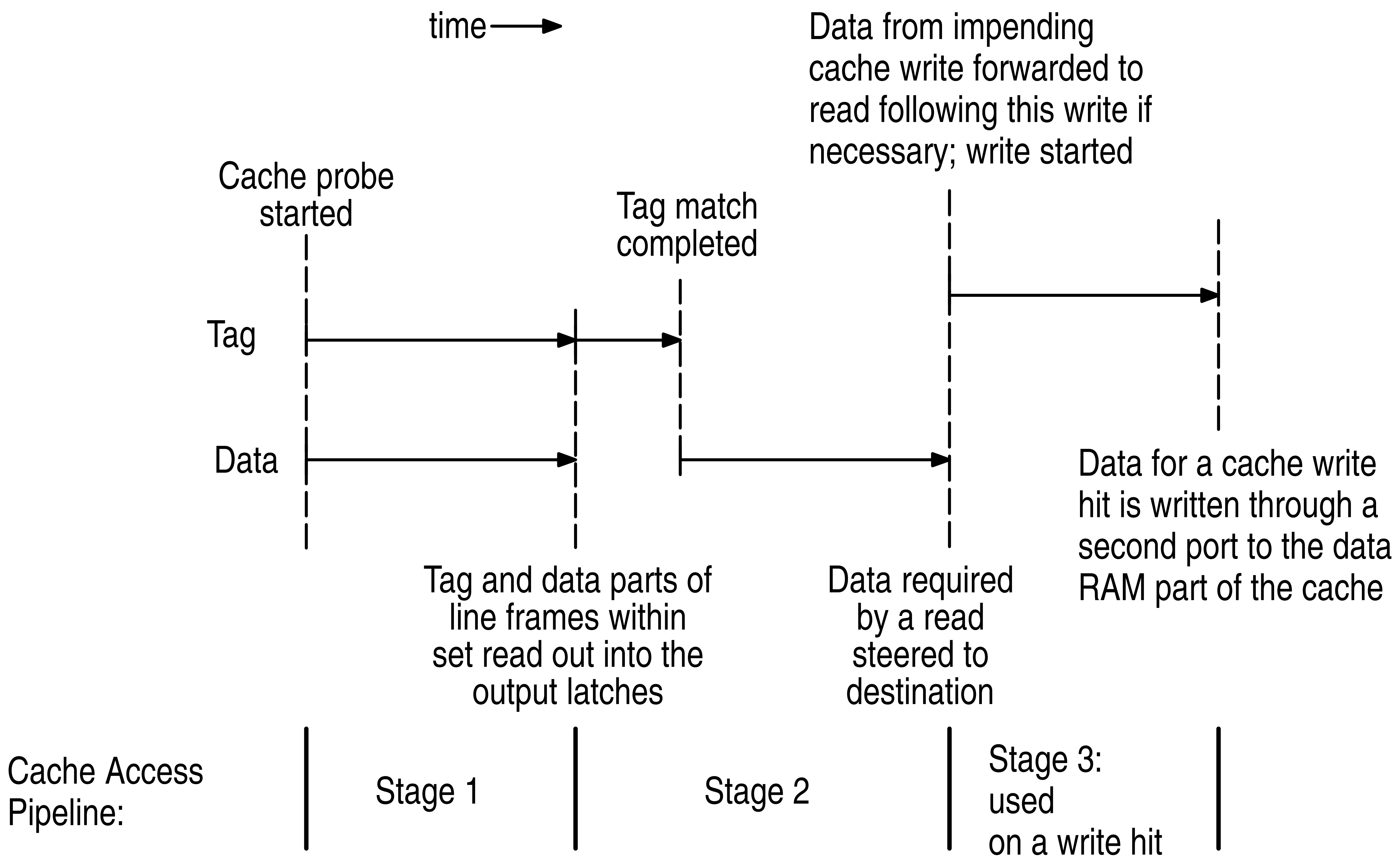
- When h = 0.98 (this is also achieved in practice for somewhat bigger caches)

  T_{eff} = 1.415

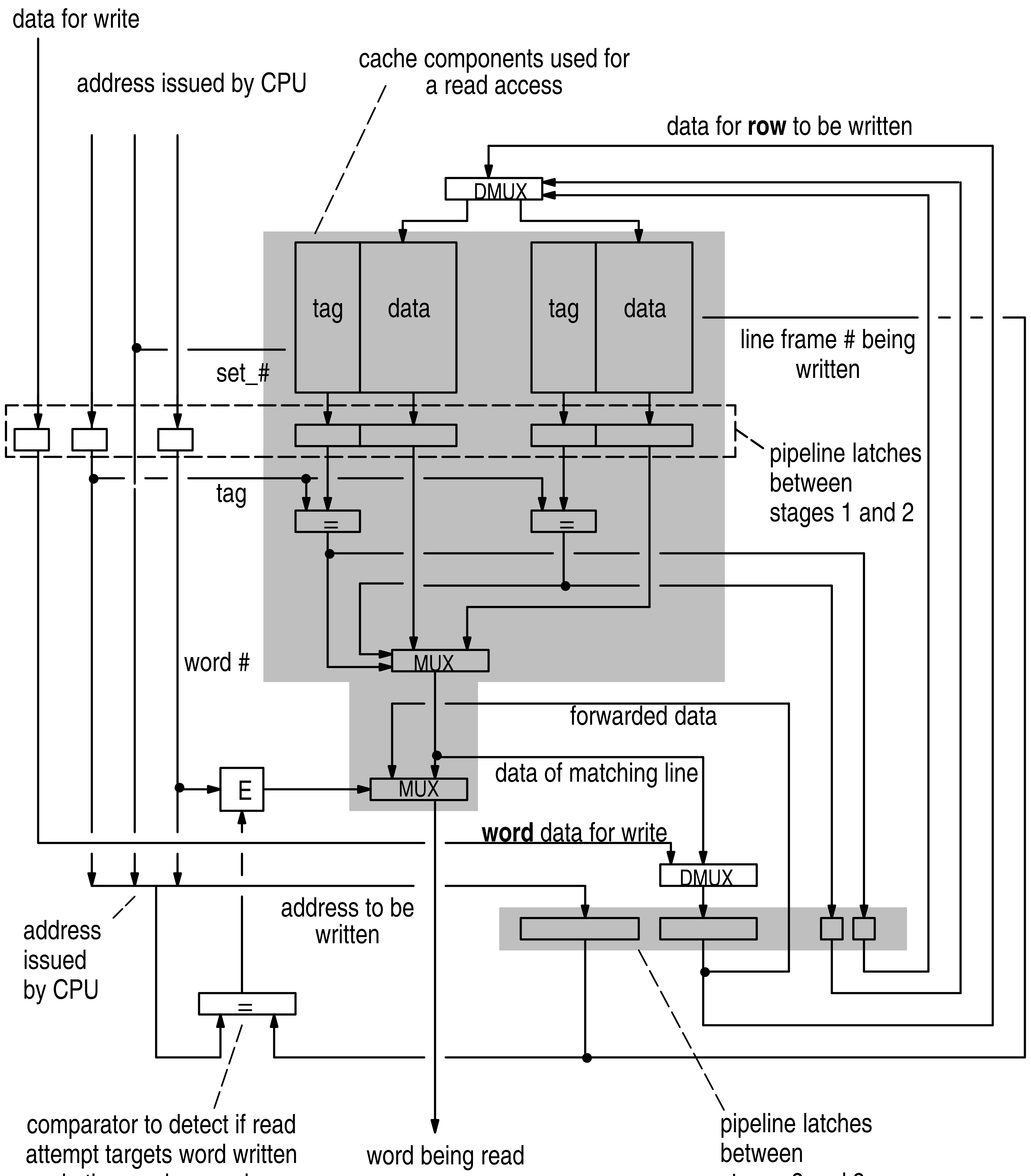
  – The effective memory access time is roughly 70%, 50% and 40% more than the cache read access time

- Best value of T_{eff} is 1.3 occurs when there are no misses (h = 1), which is, of course, impossible in practice. Note that the extra 0.3 cycles beyond the 1 cycle needed for a read access come from the additional time needed for a write during a write hit. A subsequent cache access is delayed till the write is completed.

- Resulting timings:


time
Data from impending cache write forwarded to read following this write if necessary; write started
Cache probe started
Tag match completed
Tag
Data
Data for a cache write hit is written through a second port to the data RAM part of the cache
Tag and data parts of line frames within set read out into the output latches
Data required by a read steered to destination
Cache Access Pipeline:
Stage 1
Stage 2
Stage 3: used on a write hit

Pipelined Cache Writes and Reads: one cache access per cycle maintained on read or write hits


data for write
address issued by CPU
cache components used for a read access
data for row to be written
DMUX
tag
data
tag
data
line frame # being written
set_#
pipeline latches between stages 1 and 2
tag
=
=
MUX
word #
forwarded data
data of matching line
E
MUX
word data for write
DMUX
address to be written
address issued by CPU
=
comparator to detect if read attempt targets word written in the previous cycle
word being read
pipeline latches between stages 2 and 3

Types of Cache Misses

- Compulsory or cold-start (first access)
- Capacity (cache is legitimately full)
- Conflict (competition for line frame or set)

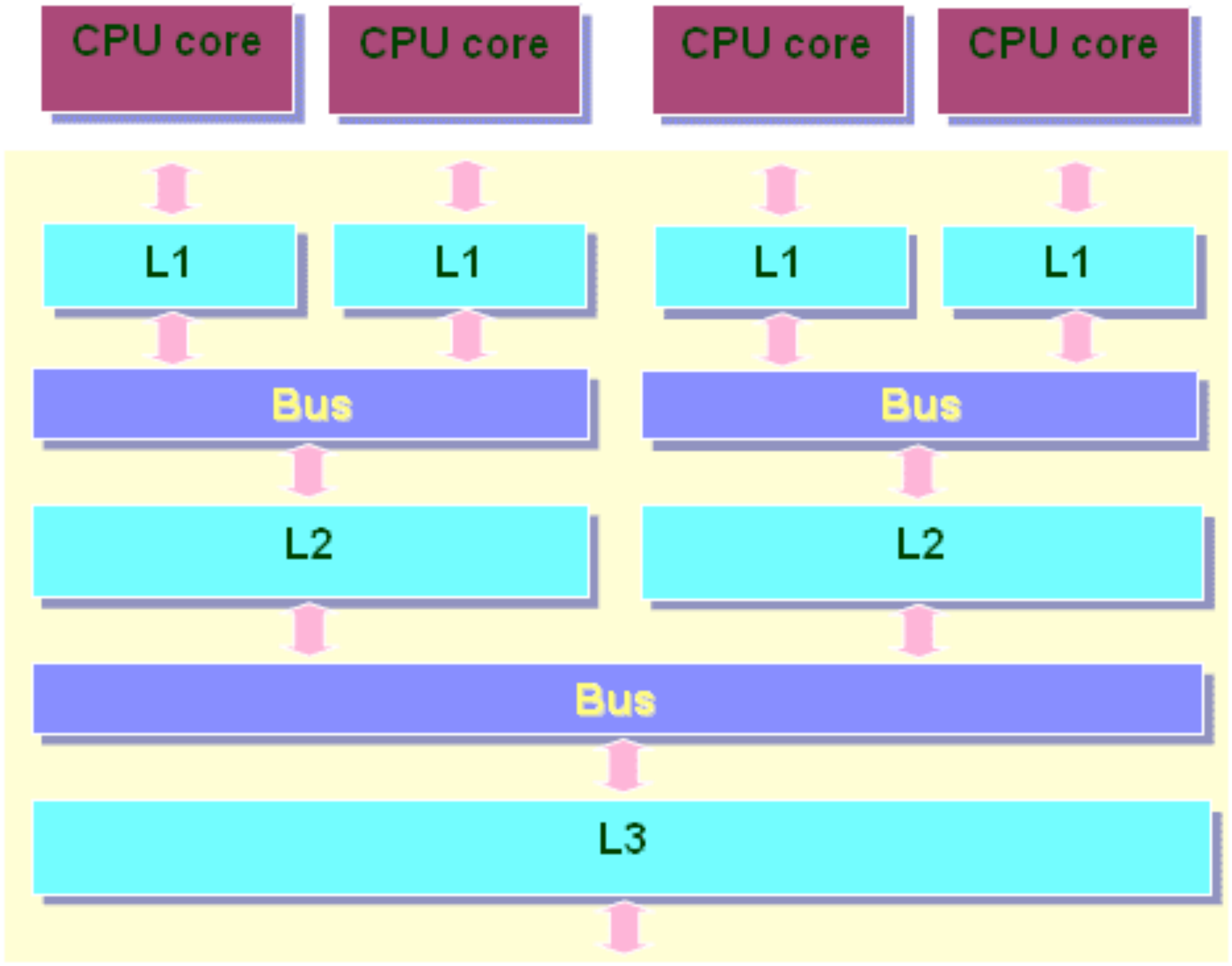
Optimizing Associativity

- Associative cache has the same miss rate of a direct-mapped cache of 2X capacity.
- Larger cache: Longer access time
- Larger associativity: Longer access times
- Associativity beyond 8 has diminishing returns

Improving Cache Performance

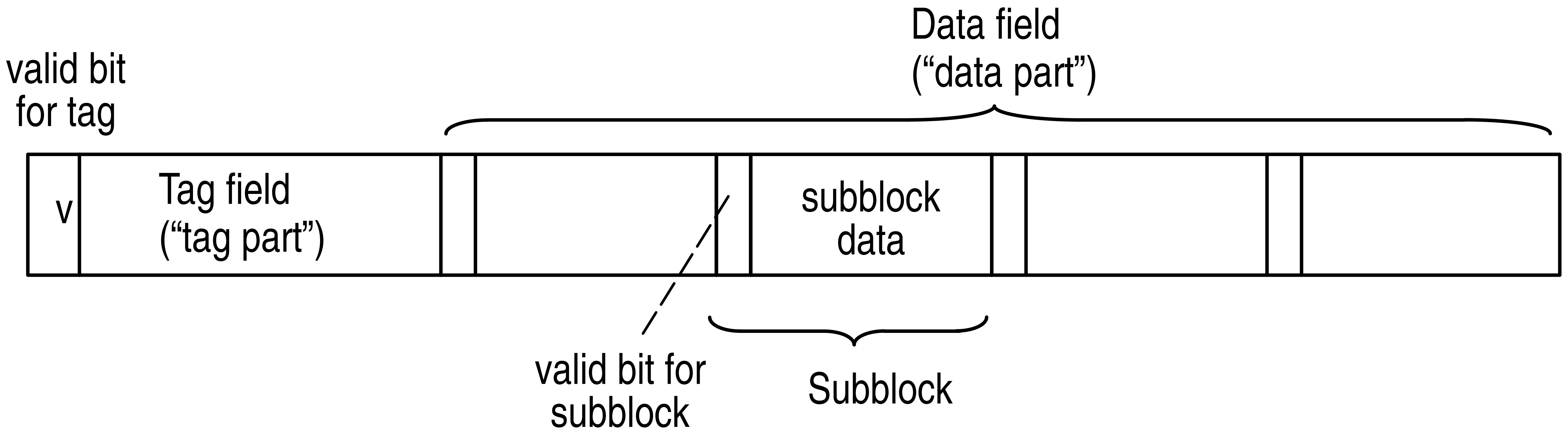
- Separate I-Cache and D-Cache
- Read-through + wrap-around
- Write buffering
- Load bypassing
- Multiple levels of cache
- Non-blocking caches
- Victim caches

Cache Hierarchy

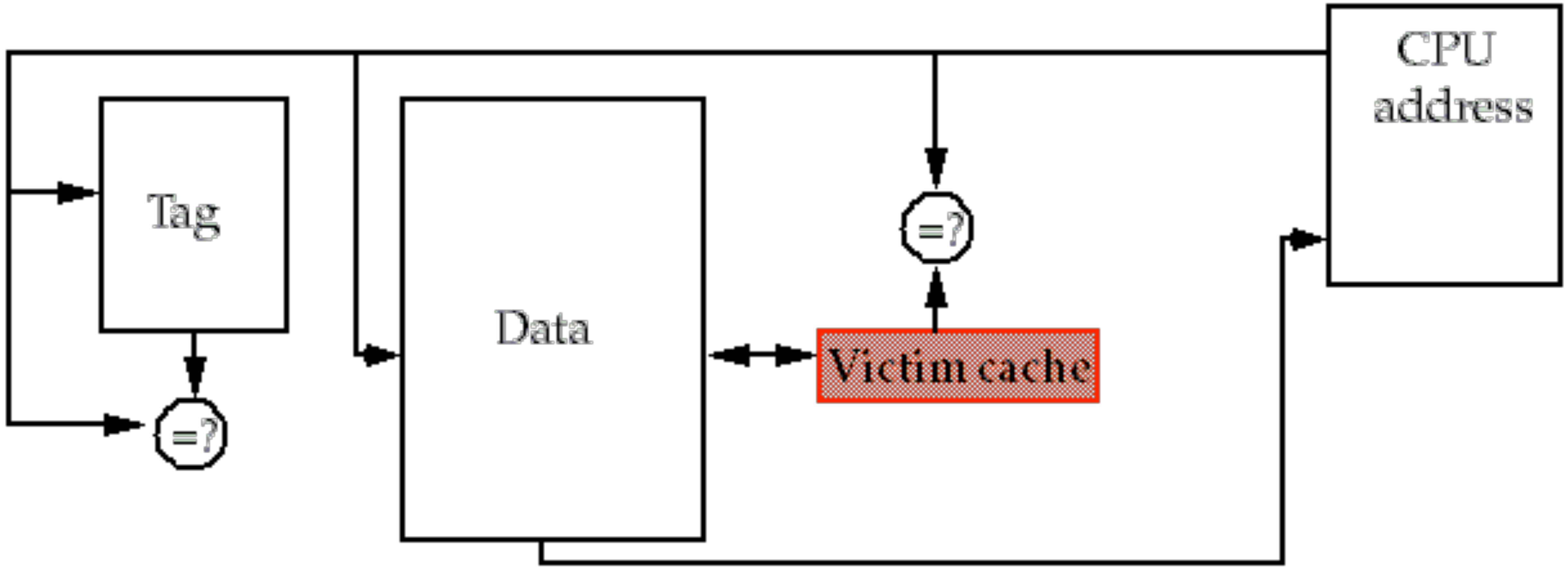


Sub-Blocking

- Longer cache lines, subdivided
- Valid bit for tag (anything in the line at all?)
- Valid bits for each subblock



Victim Cache



Cache-Aware Software

- Data Padding
- Array Merging

```
for (i = 0; i< N; i++) {
  A[i] = B[i] + C[i];
  B[i] = B[i]/2;
}
```

- Loop Fusion

```
for (i=0; i<N; i++) {
    B[i] = A[i] * C;
}
for (i=0; i<N; i++) {
    D[i] = A[i] * E[i];
}
```

Cache-Aware Software

- Loop Interchange

```
for (j = 0; j < N; j++)
   for (i = 0; i < N; i++)
      if (A[i, j] > s) A[i,j] = A[i,j] - s;
```

- Blocking
    - E.g. 2D matrix multiply

Cache Replacement Algorithms

Cache Replacement Algorithms

- Select a line to evict due to capacity or conflict miss
- Fast -- minimize miss handling time
- Small -- minimize area and energy requirements
- Preferable: Victim is chosen in advance, based on earlier accesses

Bélády's Algorithm

- Optimal replacement algorithm
- Clairvoyant
- Discard block not needed longest into the figure -- LRU in the future
- Impossible to implement in real systems
- Computable from traces, useful for setting upper bound in experiments

Least Recently Used

- Discards the oldest line
- Expensive to keep track of
- Preferred method
- Typically approximated
- http://www.vldb.org/conf/1994/P439.PDF

Hot/Cold Bit

- Used in 2-way set associative caches
- One status bit indicates which way was last accessed (hot)
- The cold way is the next victim

Pseudo-LRU

- What we call LRU approximations
- Discards **one of** the least recently used lines
- Almost as good as LRU

Least Frequently Used

- Keeps hit count for each line
- Discards line hit the least number of times
- Poor at handling software phase changes

Not Last Used

- Keep track of way last accessed
- Select victim at random, different from way last accessed
- Slightly better than purely random

Rotating Pointer

- For associativity P, keep counter of $\log_2 P$ bits for each set
- Counter selects next victim
- On hit:
  - Increment pointer
  - Increment again if pointer==selected way
- Victim selection functionally random, but round-robin for placement

Most Recently Used

- Replaces the most recently accessed line
- Useful for streaming data with high spacial but no temporal locality
- Applied adaptively alongside other algorithms

Random Replacement

- Chooses victim randomly
- Requires no statistics storage
- Has good performance stochastically
- Probability of choosing victim that will be accessed next is 1/P
  - Better than direct-mapped

Adaptive Replacement Cache

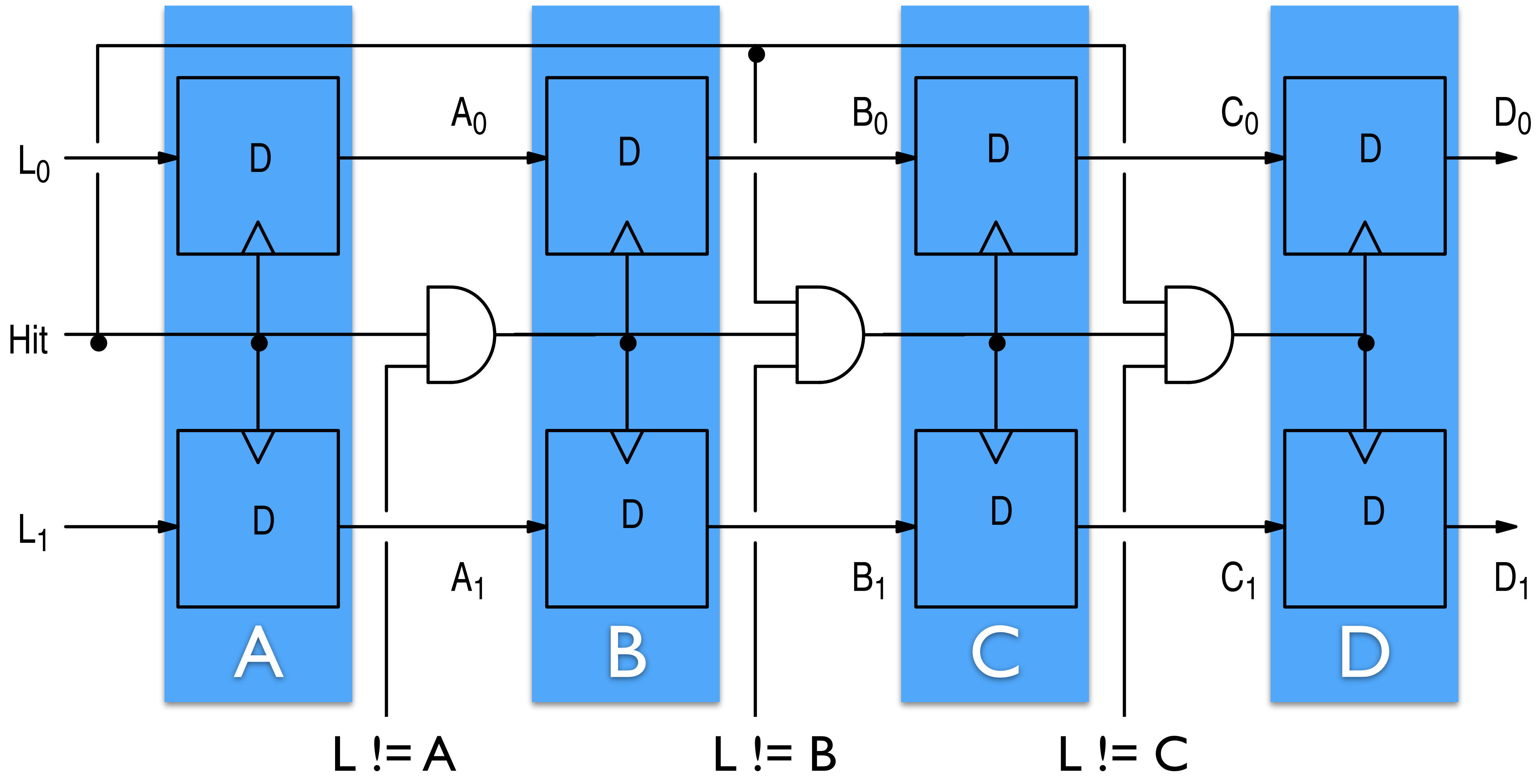
- Splits cache into LRU and LFU segments, to get some benefits of each
- Uses recent eviction history to actively adjust split

Other algorithms

- Second chance
- SLRU -- Divides LRU cache into probationary and protected segments
- FIFO
- CLOCK
- CLOCK with Adaptive Replacement (CAR)
- Multi Queue algorithms

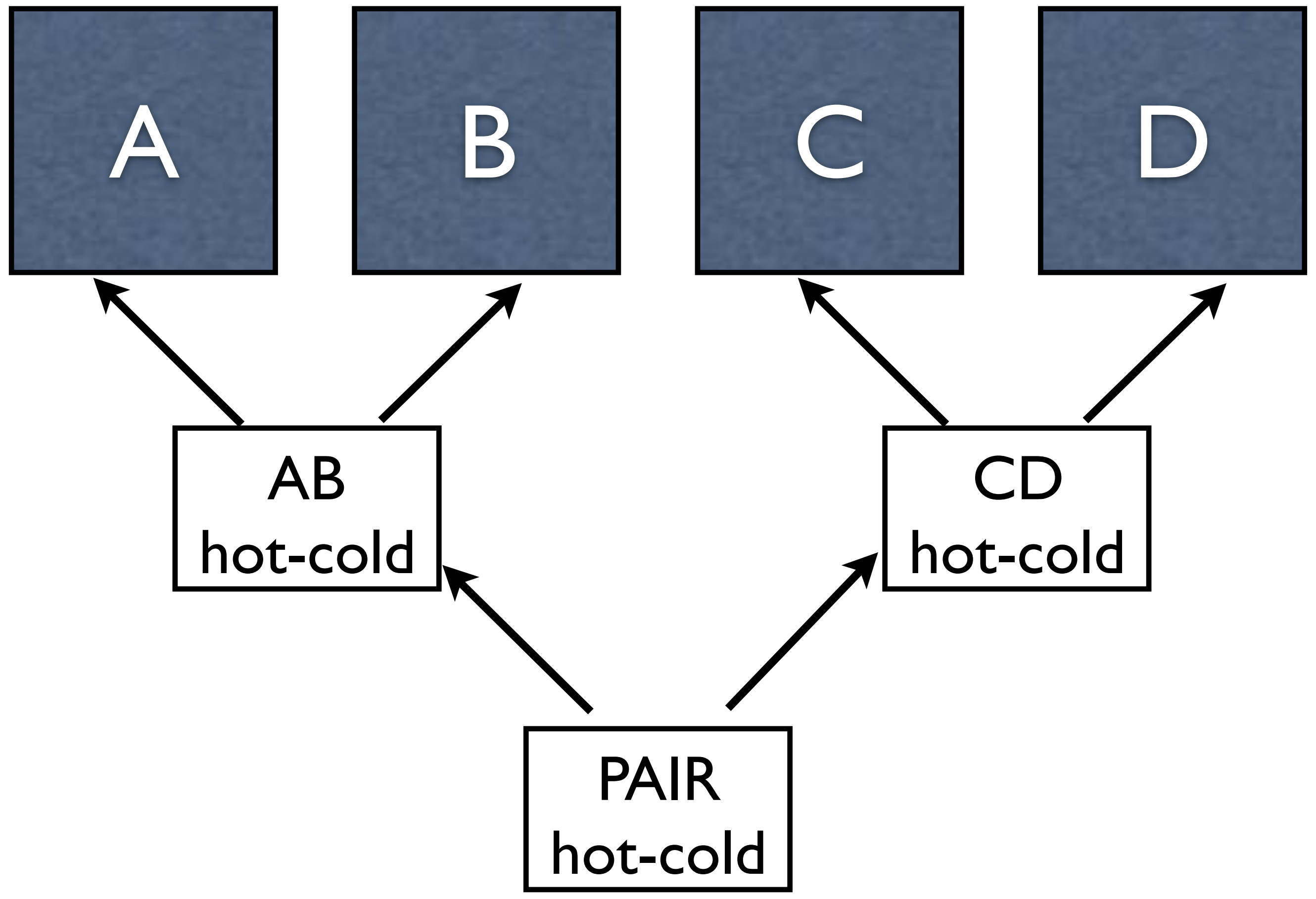
Least Recently Used

- Track order in which lines have been accessed

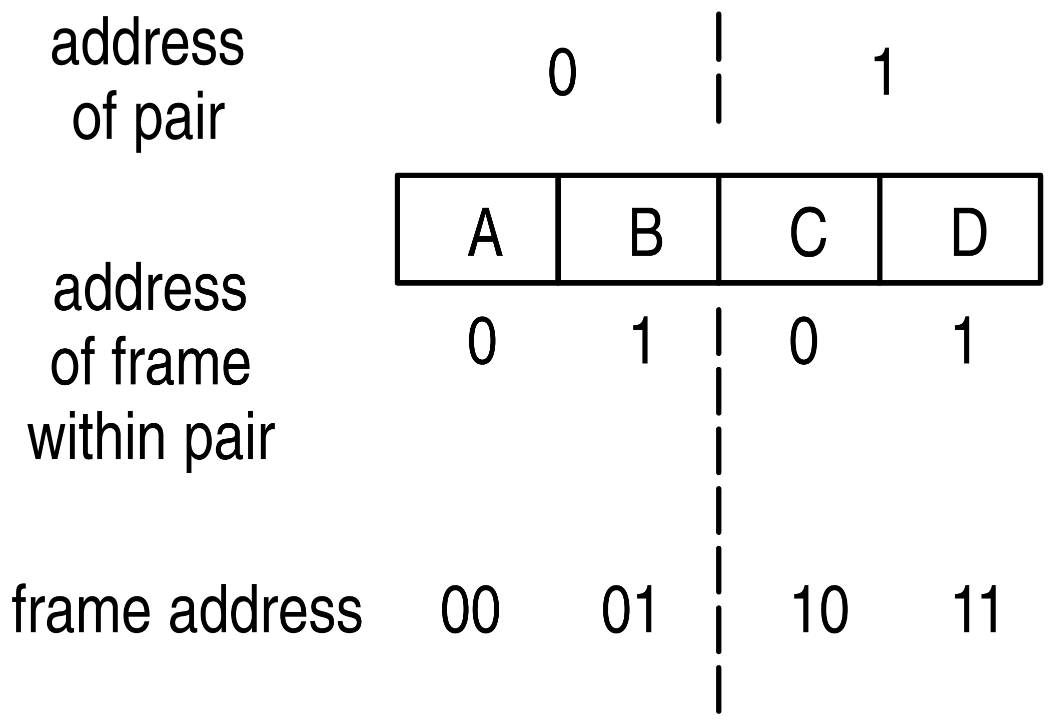


Approximate LRU

- Multi-level hot/cold



- An example trace; this assumes that initially all three status bits are cleared.

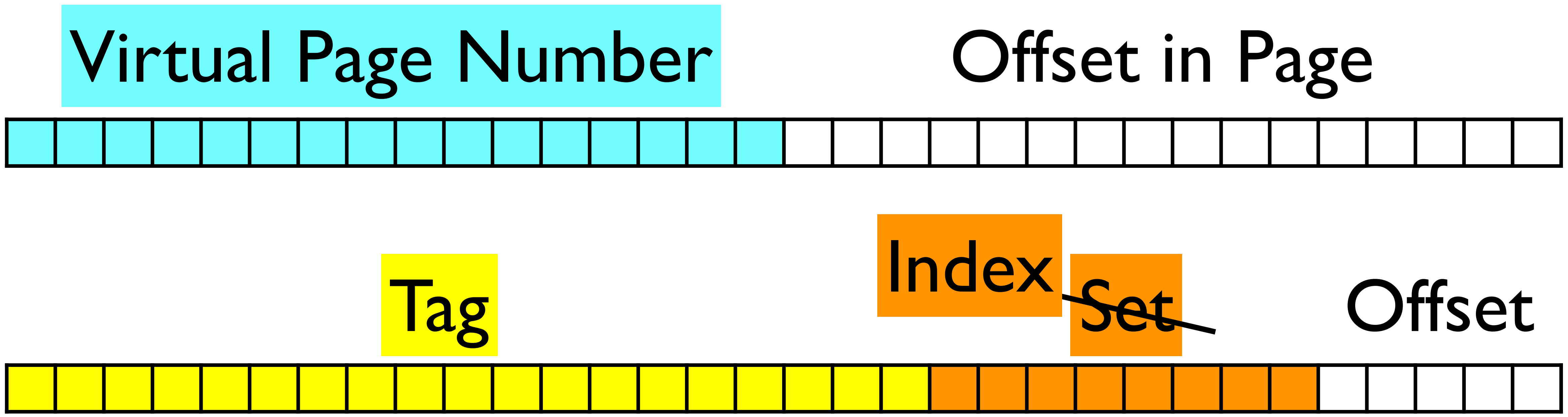


Frame accessed	Status bits			
	Pair	frame within pair 0	frame within pair 1	victim
A	0	0	0	D
B	0	1	0	D
C	1	1	0	A
D	1	1	1	A

– Note why this is approximate LRU: the victim selected after accessing C should have been D if true LRU was implemented.

Virtual Memory Addressing

- Virtual memory managed in units of *pages*
- Virtual addresses translated to physical addresses before accessing memory
- Complicates caching



Translation Lookaside Buffer

- Small, fully-associative cache
- **Tag:** Virtual page number & AS-id
- **Data:** Physical page number
- Typically uses simple replacement algorithms like random, rotating pointer, or NLU

TLB facts

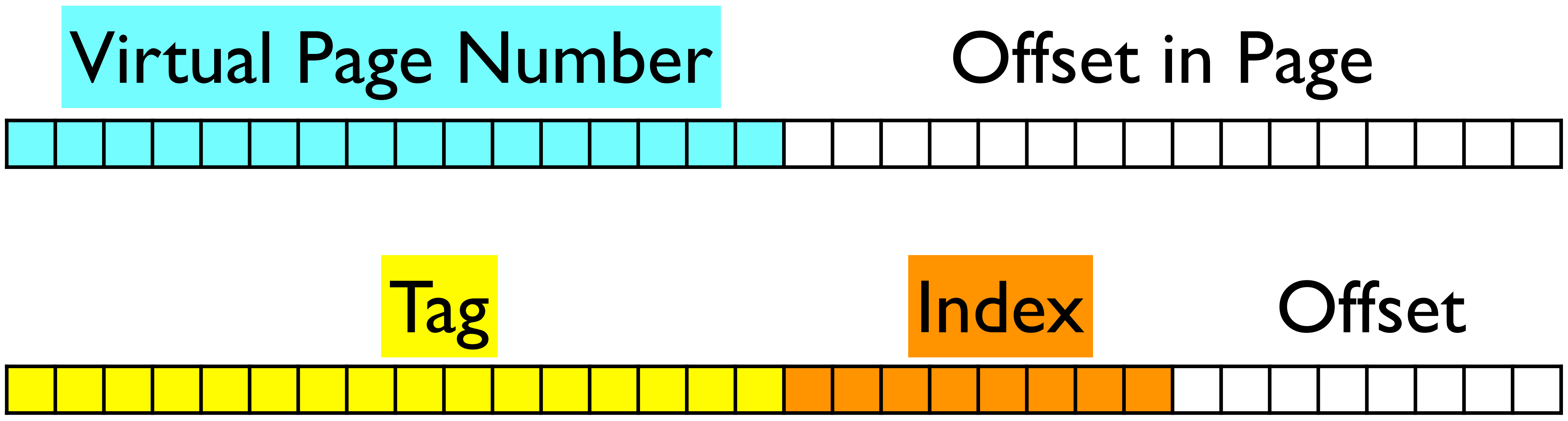
- Access time usually shorter than cache tag array.
- High hit rate: One VPN corresponds to entire page.
- Separate TLBs for instructions and data
- TLBs also store permissions (e.g. write, execute)

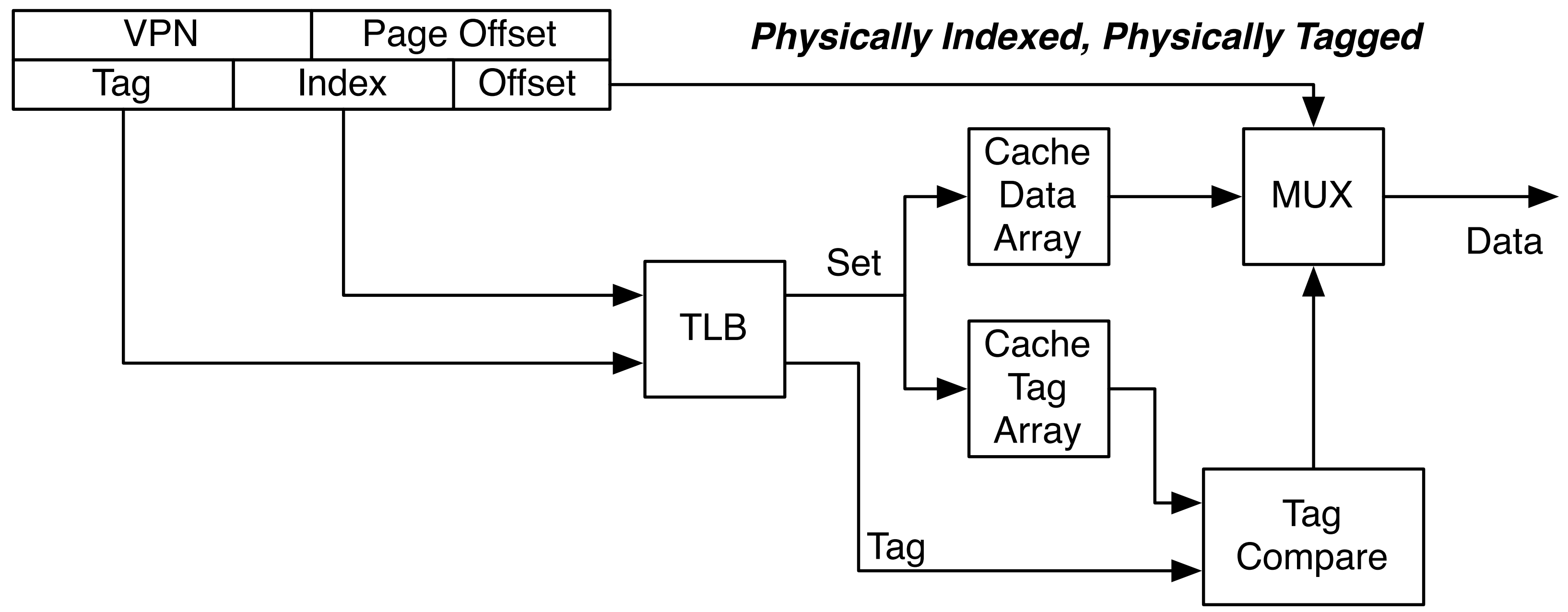
When to translate virtual page number?

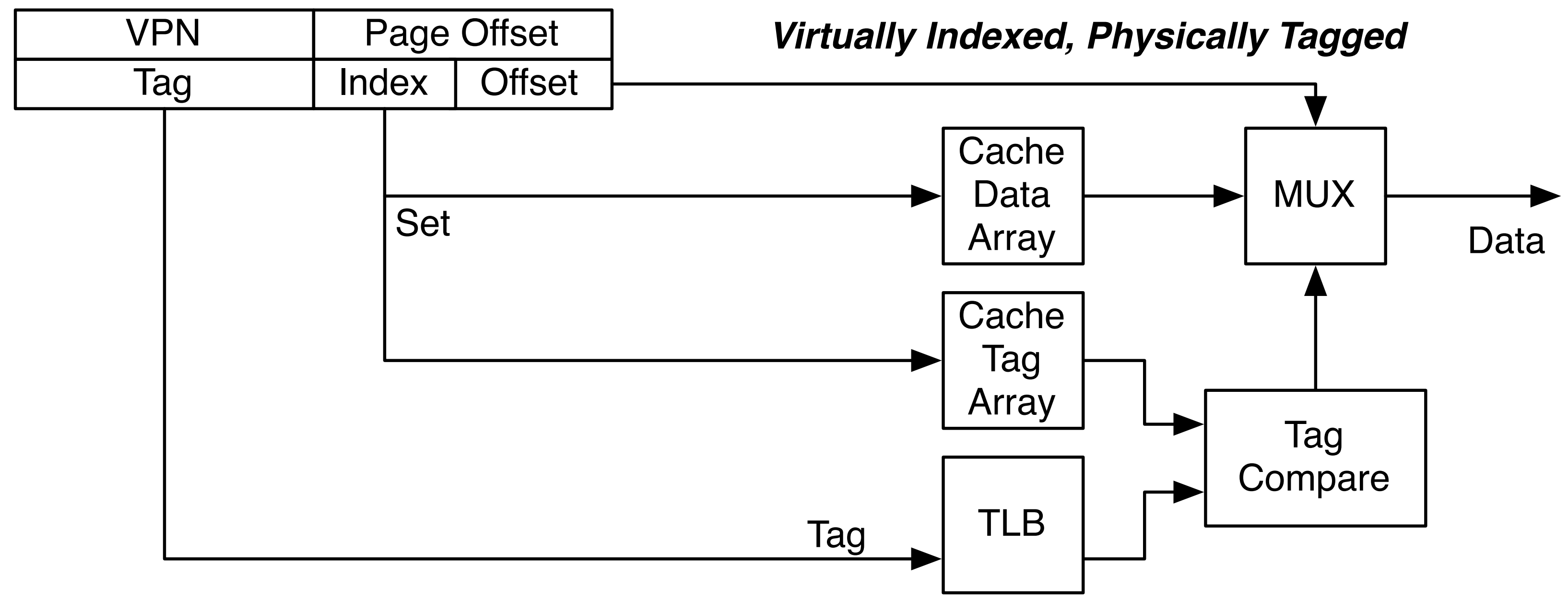
- Before cache access
  - “Physically tagged”
  - Serializes TLB and cache access
  - Implicitly avoids virtual address collision
- After (or during) cache access
  - “Virtually tagged”
  - TLB and cache access in parallel
  - Must disambiguate virtual address spaces (AS-id)

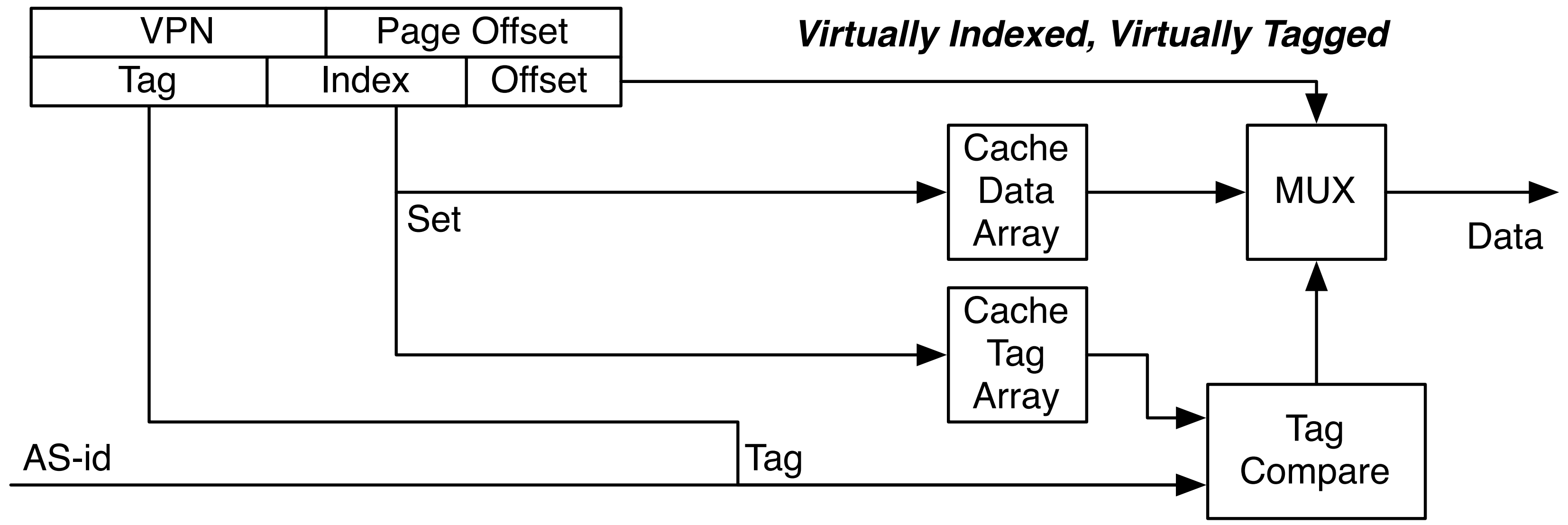
When to translate virtual page number?

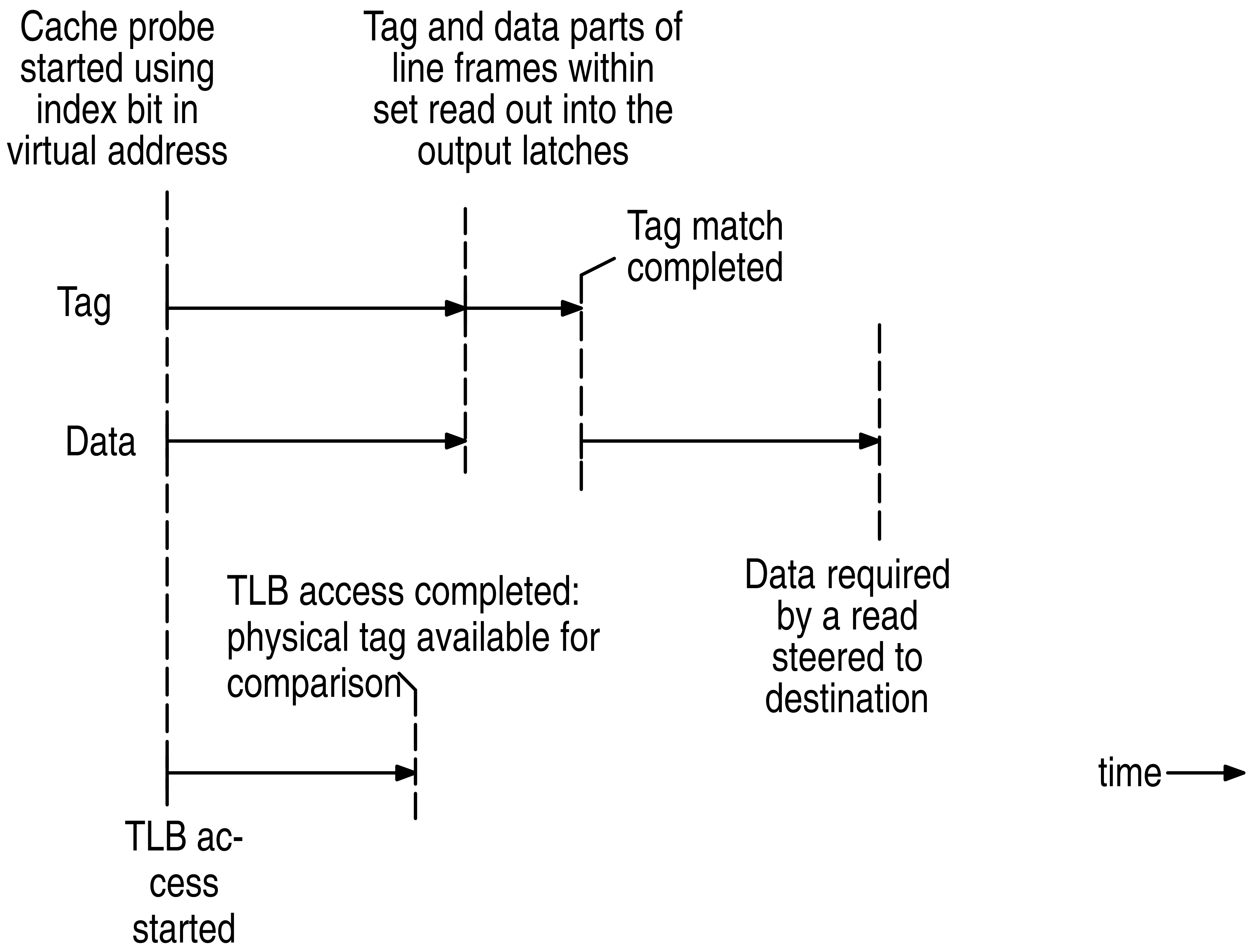
- Tag contained in page offset
  - Index both virtual and physical
  - Restricts cache size



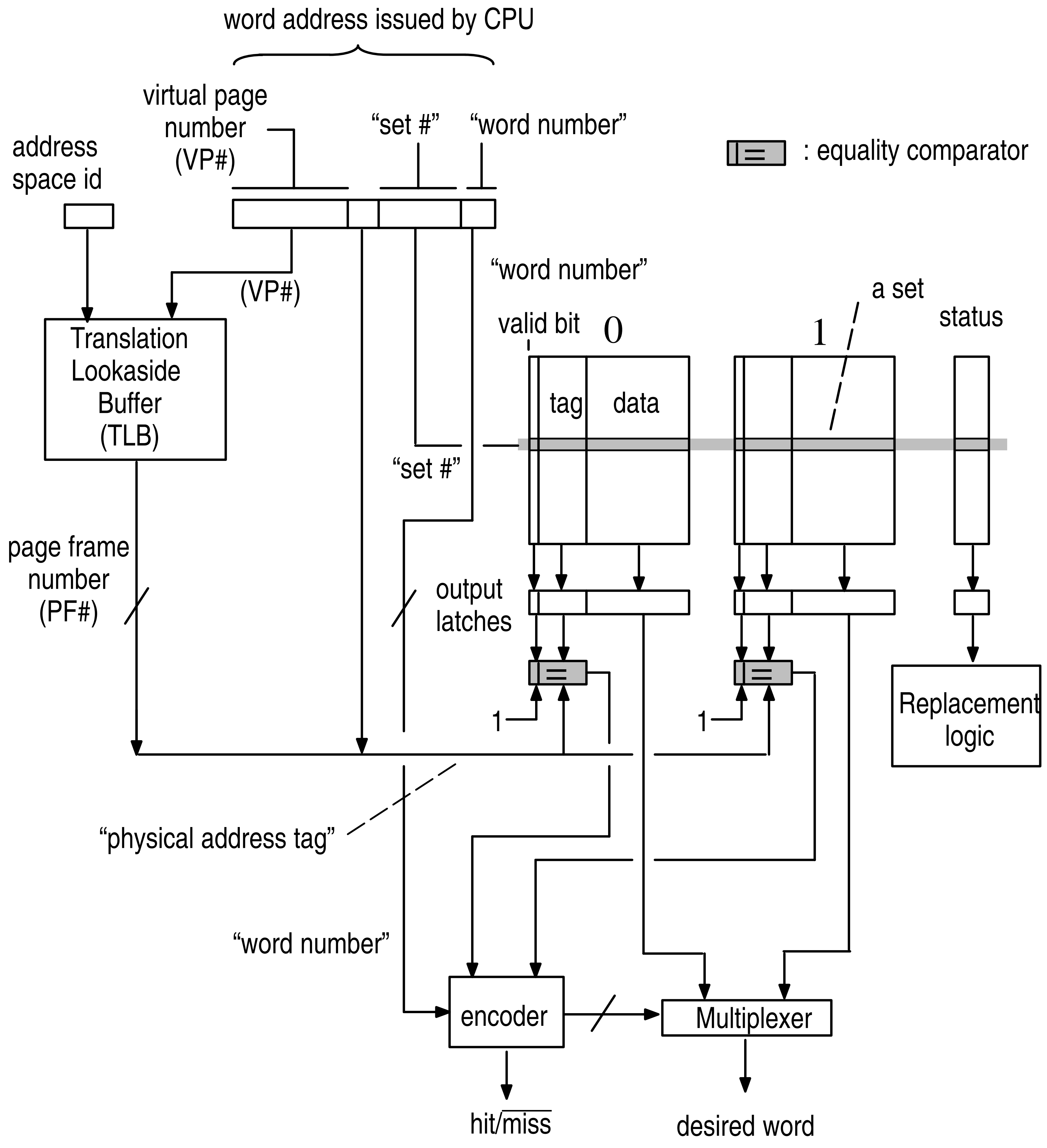

VPN
Page Offset
Tag
Index
Offset
Physically Indexed, Physically Tagged
TLB
Set
Cache Data Array
Cache Tag Array
MUX
Data
Tag
Tag Compare


VPN
Page Offset
Tag
Index
Offset
Virtually Indexed, Physically Tagged
Set
Cache Data Array
Cache Tag Array
MUX
Data
Tag Compare
Tag
TLB


VPN
Page Offset
Tag
Index
Offset
Virtually Indexed, Virtually Tagged
Set
Cache Data Array
Cache Tag Array
MUX
Data
Tag Compare
AS-id
Tag


Cache probe started using index bit in virtual address
Tag and data parts of line frames within set read out into the output latches
Tag match completed
Tag
Data
TLB access completed: physical tag available for comparison
Data required by a read steered to destination
time
TLB ac-cess started

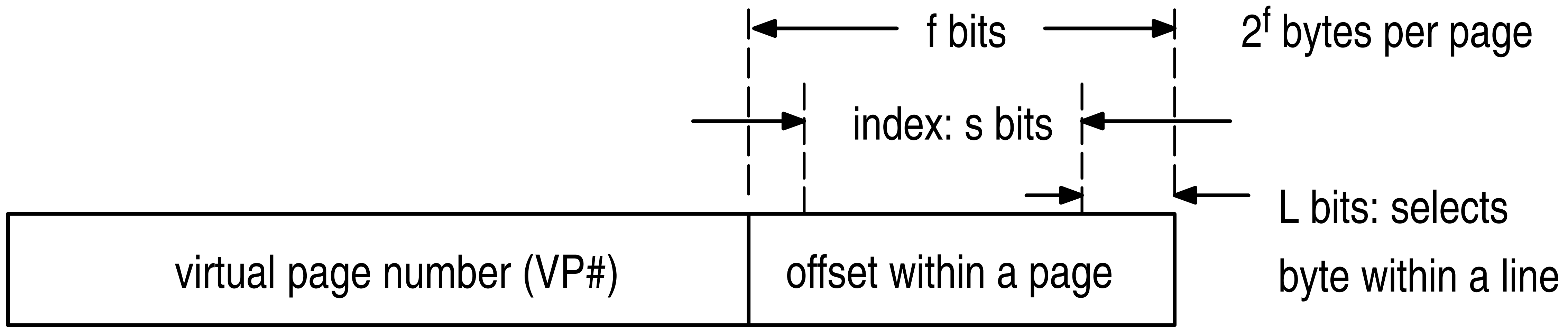
Overlapping the access of a physically addressed cache with TLB–based address translation



A 2–way set–associative cache with a TLB

- Guranteeing the constraints of overlapped translation for a physically addressed cache:

► Ensure that the index bits are confined entirely within the offset field:



- This requirement boils down to ensuring that $(s + L) \leq f$
- The maximum value of the number of index bits is thus:

$$s_{max} = (f - L)$$

– This has implications on the total capacity of the cache:

If the cache is p–way set–associative, the maximum cache capacity that allows address translation to be overlapped with tag and data array accesses is:

$$D_{max} = p * 2^{smax} * 2^{L} \text{ bytes} = p * 2^{(f-L)} * 2^{L} \text{ bytes}$$

– There are several ways to increase the cache capacity and still meet the constraints of overlapping address translation:

(a) Increase associativity (p) – not preferred beyond a limit

(b) Increase the line size (2^L) – not preferred beyond a limit, as cache miss handling time increases, unless **subblocking** is used.

Virtual Address Caches

- Used in the Univ. of Manchester MU5 prototype and the Berkeley SPUR prototype and the multi–chip implemented MIPS 6000 CPU (all are history now!)

- Main advantage: no address translation is needed before a cache access; no TLB is needed.

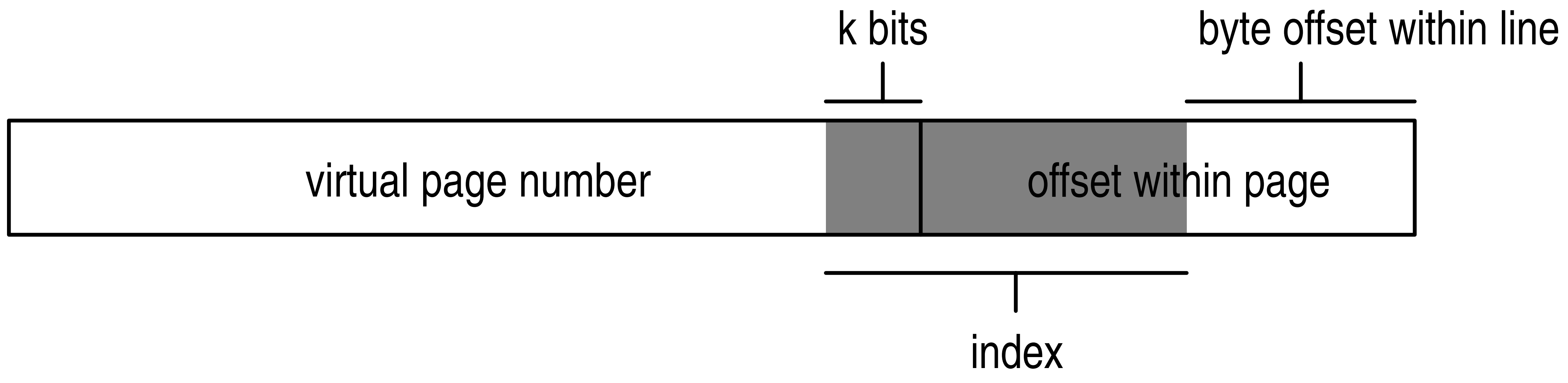
- Presents some serious problems:

Virtually-addressed Cache Challenges

- Shared memory
  - Line frame shared among processes has multiple virtual addresses
- Solutions:
  - Flush caches at context switches
  - Detect and evict synonyms
  - Common AS-ids for shared pages

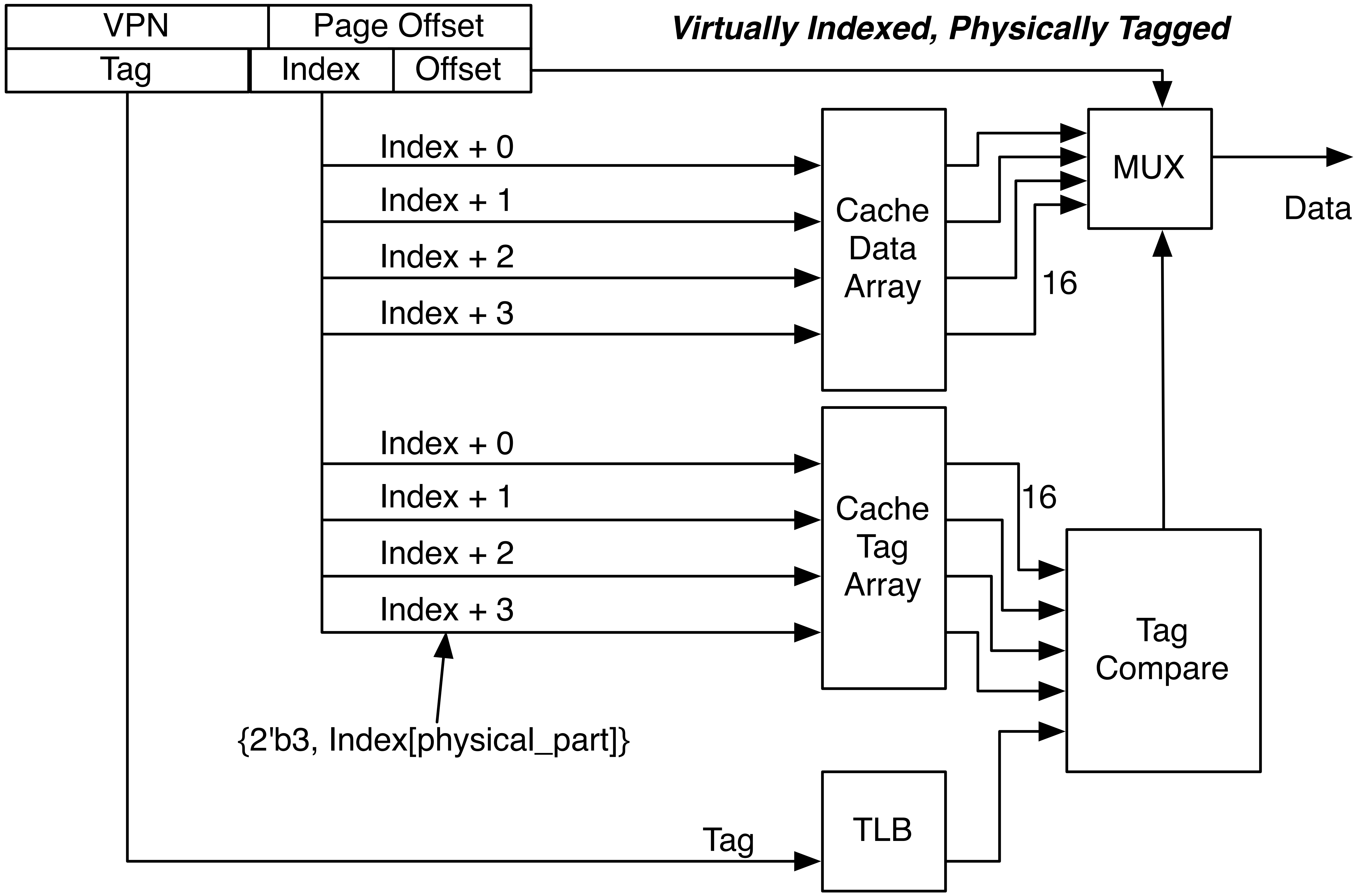
VIPT Caches in General

- Desire to avoid address translation (TLB) for cache data lookup.
- With K index bits in VPN, a shared line frame may be in up to 2^K different sets



Accessing VIPT Caches

- With K index bits in VPN, a shared line frame may be in up to 2^K different sets
- Therefore, provide 2^K ports on sub-cache SRAMs
- Read all 2^K sets simultaneously
  - Simultaneous with TLB lookup
- All synonyms will have the same physical tag.
- For writes, modify and save **all** synonyms.
- Remains consistent, no need for flush on context switch.
- Direct-mapped VIPT cache retains speed advantages


VPN
Page Offset
Tag
Index
Offset
Virtually Indexed, Physically Tagged
Index + 0
Index + 1
Index + 2
Index + 3
Cache Data Array
MUX
Data
16
Index + 0
Index + 1
Index + 2
Index + 3
Cache Tag Array
16
Tag Compare
{2'b3, Index[physical_part]}
Tag
TLB

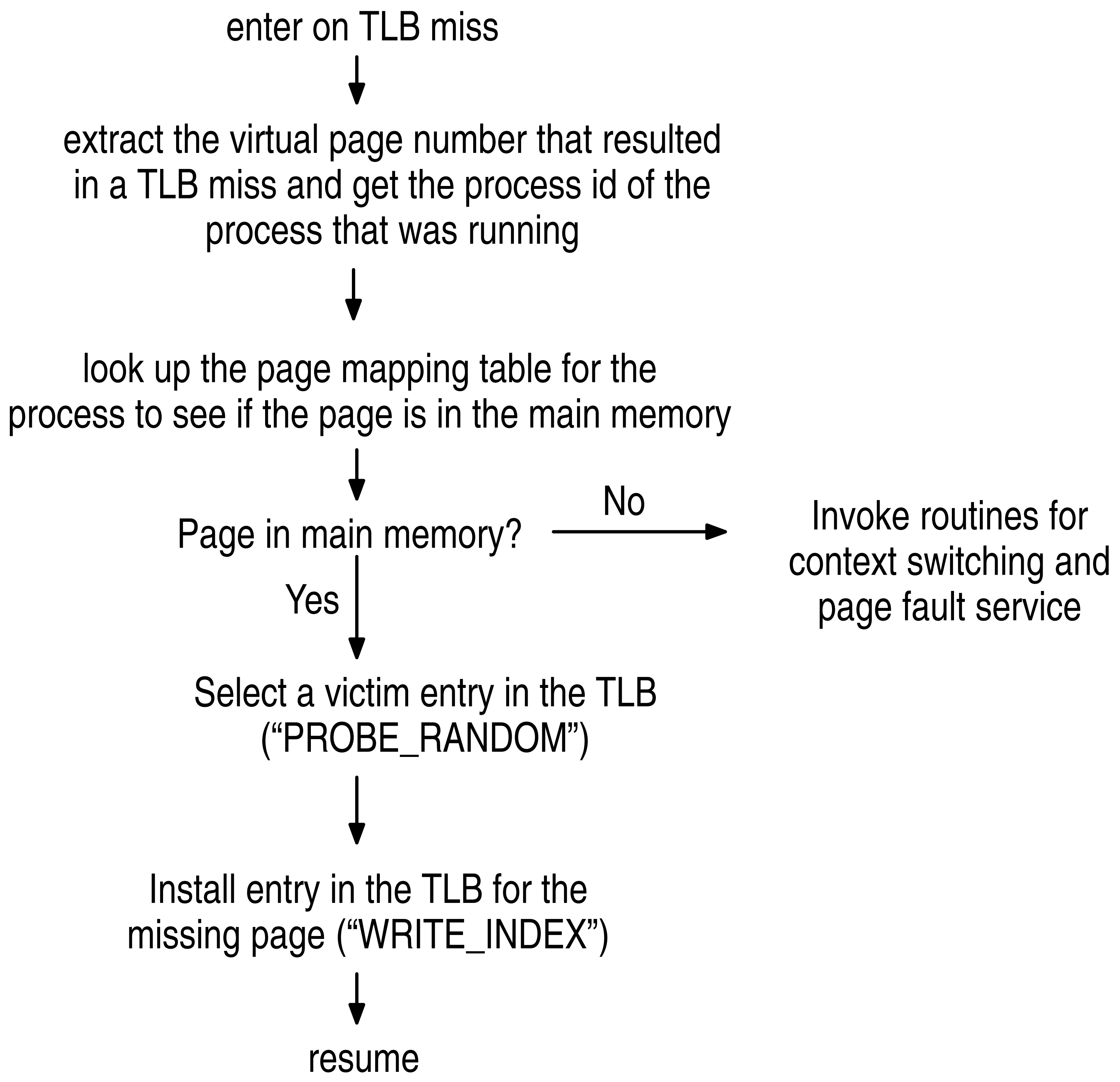
- Privileged instructions are provided to allow the OS to manipulate the TLB entries. Examples of TLB–related instructions:

Instruction	Meaning
PROBE_RANDOM <reg>	select an entry randomly from the TLB (for replacement and get its index)
WRITE_INDEX_hi|lo <index> <reg>	write contents of register into slot pointed to by index
INVALIDATE <vp#><pid>	invalidate entry for specified virtual page number for the specified process

  – The virtual address that resulted in a TLB miss is also available in a special TLB register. Other instructions and special registers are also used.

  – A special instruction may also be included to flush the entire TLB.

- TLB misses are usually handled in software (TLB misses trap to a kernel–level trap handler, which typically use dedicated trap handling registers; such traps can be serviced without the usual saving and restoration of registers on a context switch):


enter on TLB miss
extract the virtual page number that resulted in a TLB miss and get the process id of the process that was running
look up the page mapping table for the process to see if the page is in the main memory
Page in main memory?
No
Invoke routines for context switching and page fault service
Yes
Select a victim entry in the TLB ("PROBE_RANDOM")
Install entry in the TLB for the missing page ("WRITE_INDEX")
resume

Speeding Up TLB Miss Handling: Inverted Page Tables

- Dedicate registers for TLB miss handling – no need for full–fledged context switch.

- Also need to look up page table very quickly on a TLB miss.

- Simplest way of looking up a page table: use virtual page number as an index to directly locate the required page–mapping table entry (“direct lookup”). These page tables must be held in the RAM for best performance (= least TLB miss handling time).

- Problem with direct lookup: enormous amounts of RAM needed to hold page table entries in RAM:

  Example: For a system that supports 48–bit virtual addresses, with 12 bits of byte offset within a page, 2^{36} page table entries are needed. If each entry requires 8 Bytes, this translates to a storage requirement of 2^{39} Bytes. Not only is this huge but also beyond the physical addressing capability of all contemporary CPUs (32–bit physical addresses are common)!!

- Solution: store mappings (virtual page number to physical page frame number) in the RAM only for the pages that have been bought into the RAM – this is done using an **inverted page table**.

– The "inverted" qualifier alludes to the fact that there is (at most) one entry in this table for each page frame (as opposed to one entry per virtual page in the normal (disk–resident) page tables. "Inverted" does *not* imply that the table is searched using the page frame number – we still use the virtual page number to perform any lookup in the inverted page table.

Inverted Page Table

- Typical implementation:
  - hash virtual page number into an index into a table of pointers
  - the entry located in this table is a pointer into linked list of page table entries, which are searched linearly
  - linked list is required to handle collisions on hashing (different virtual page numbers mapping to the same pointer table index)

► Time to locate an entry = time to generate pointer table index through hashing + time needed to search linked list.

- ▶ Fast TLB miss handling implies linked list should be small.

- ▶ Tradeoff: size of pointer table vs. size of linked list; increasing pointer table size reduces collisions on hashing and reduces the size of the linked list.

- Allows entries for pages within memory to be updated easily on page swaps.

- Hash function: typically simple ex–or of bits fields in the virtual address (e.g., ex–or of the bits of the segment address (see next foil!) and the virtual page number within this segment (with zero padding to equalize the bit lengths; the necessary number of lower order bits of the ex–or–ed value is used as an index into the pointer table).

Hiding Memory Latency: Multi–threaded Datapaths

- Also called “hyperthreading”. Thread = a process (incorrect usage of the term “thread”, but the processor industry is kind of stuck with this!).
- Same as “instruction level” multiprocessing, discussed earlier.
- Here, instructions from different threads are injected into the pipeline. In a superscalar multithreaded design, instructions from different threads can be co–dispatched. Hence the alternative name, simultaneous multithreading (SMT).
- Threads have their own *contexts*: context = state information needed to run a thread = registers, rename tables, PCs etc.

- Common implementation:

- ► Some resources, such as L1 I–cache and L1 D–cache, caches beyond L1, IQ, fetch logic, parts of the predictor, execution units, etc. can be shared among threads – some restrictions may be imposed on the sharing to prevent the hogging of resources by a thread.

- ► Other resources (rename table, physical registers, ROB, LSQs etc) can be dedicated to threads.

- ► Note that partitioning the ROB among threads can indirectly limit use of shared resources among threads. Additional mechanisms may be needed to ensure that all threads are making progress – e.g., round robin instruction fetching, dispatching.

- Such multithreading provides a way of hiding memory latency –when a thread is blocked due to a cache miss, instructions from other threads can continue to be processed.

- Multithreaded datapaths thus improve average throughput on a set of ready–to–run threads; individual execution time of a thread can go up!

- Real designs: DEC Alpha EV8 (21464) – never put into production, HT versions of Intel P4 ("Northwood", "Prescott"), IBM POWER 4 and beyond, Sun's Niagara.

- There is a practical limit on the number of threads supported simultaneously – shared resources increase in size and become slower, putting a limit on the pipeline clock.

► A limit of 2 to 4 threads is common; limit can be higher with simpler datapaths (e.g., 8 or more threads in Sun's in–order Niagara implementations).

- Most OSs typically have a few "threads" (really, processes) that can be run simultaneously multithreaded. Typical server code is heavily multithreaded in this sense – as soon as a server request comes in, a process (i.e., thread in the jargon of SMT) is spawned to serve the request.

Prefetching in Contemporary Processors

Basics

- Prefetch request: anticipated memory read request.
- Demand request: actual request (triggered on a miss) – different from prefetch request.
- Hardware–initiated prefetching anticipates the next address from which a load instruction will access data and performs this access in advance to move the requested data into the L1 cache well before it is needed.

- Streaming instructions (typically found in the multimedia instruction sets such as SSE) do this based on user supplied information, as seen earlier.

- For non–explicit accesses, the hardware needs to anticipate the address of the memory locations to be accessed in advance.

- Many ISAs support explicit prefetching commands that can be inserted by the compiler in strategic places within the code.
  - In many cases, these instruction specify what is to be done with the pre-fetched data (place in cache, consume in buffers only without being moved into the cache, get data for potential update and writeback with no intent to cache for future use etc.)

- The term **prefetch distance** is used to refer to how far in advance (in terms of number of data items) the actual prefetching starts.

► This distance has to be chosen to account for delays in the access path and the number of misses that would occur before any data is prefetched.

– Keeping this distance small would reduce potential near–term misses but may not prefetch data far enough in advance to avoid future cache misses.

– Keeping this distance large may trigger too many near–term demand misses.

- **Prefetch burst size** refers to the number of items prefetched back–to–back. A small burst size can cause future misses; a large burst size may bring in a lot of data in vain (prefetched data does not get used). ***Unutilized prefetches*** can waste precious memory bandwidth and latency.

<u>Hardware–initiated Prefetching</u>

- Anticipating the address of the next memory location to be accessed: two broad techniques are used in current hardware implementation:

► Next line prefetch: prefetch next cache line – works well for instruction access as well as data access. hardware has to detect sequential access pattern before starting prefetching. For data accesses, next line prefetching amounts to prefetching at stride distances of unity (see below).

- ► Stride–based prefetching – detects *stride–based accesses* and initiates prefetching based on detected stride. Usually triggered by repeated executions of the *same* LOAD instruction in a loop, as exemplified in the following code:

```
for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            sum = sum + A[i, j] * B [j, i];
```

– If the arrays A and B are allocated in column–order, the loop that implements this code will have two LOAD instructions – one accessing A [i, j] and another accessing B [j, i]. The strides used by the LOAD for the A and the B elements are 1 and N, respectively.

Example 1: Adaptive next line prefetching in the AMD Barcelona Quad–Core Design

- Prefetch next line is triggered by two consecutive accesses: access of lines L, L+1 will trigger the prefetching of the next N consecutive lines starting with L+2.

- Burst size(N) is programmable.

- Prefetch depth is adaptive – this distance is increased dynamically if the demand stream catches up with the prefetch stream.

– The IBM POWER family uses a similar next line prefetching scheme.

Example 2: Prefetching from the L1 D–cache in the Intel Core 2 Duo

- Prefetchers exist between core and L1, between L1 and L2, between L2 and memory. A prefetch request from the upper level is treated as a demand request at the next level.

- *Instruction–pointer (IP) based prefetching* as used in this design stores information related to prefetching and related to automatically detecting the state of the prefetching and the setup of prefetching conditions in a 256–entry prefetch history table (PHT).

- Detects constant stride–based accesses. These are a series of accesses that target uniformly–spaced memory addresses. Often, the same LOAD instruction is used within a loop to generate a series of accesses. The IP prefetcher used in Intel's Core 2 duo automatically enables prefetching from the L1 D–cache for such LOAD instructions.

- The format of an entry in the PHT is as follows:

  <instruction address >: serves as a tag for locating a LOAD instruction's entry in the history table

  <12 bits of the last virtual memory address targeted by the load>

  <13–bit signed computed stride>: difference between the targeted memory addresses between the two past accesses for the past two executions of the LOAD.

  <2–bit history/state>: relates to the state and usage of this entry for pre-fetching – not disclosed, but potentially: entry just set up on first execution of load, second execution – stride between addresses used in first and second execution computed, stride confirmed on third execution, prefetching disabled.

  <6–bits of address targeted for prefetching>: used to avoid duplicate prefetches.

- The prefetching sequence is as follows:

1. The PHT is looked up for LOAD instructions at the head of the load queue.

2. On detecting a stride based access, a prefetch request is generated for the L1–D cache and queued up in a FIFO queue. As mentioned, the prefetch is triggered on the 3rd execution of a LOAD at the earliest. When this FIFO queue is full, new requests overwrite earlier requests.

3. Prefetch requests compete with streaming requests and demand misses for accessing the L1 D–cache. When the cache port is free and when appropriate number of cache fill buffers and external bus request queue entries are available, the prefetch request is processed.

– If a L1 D–cache hit occurs, the request data is fetched into the fill buffer.

– If a L1 D–cache miss is triggered, the request moves down to the L2 cache as a normal "demand" request.

4. The prefetched data is not necessarily placed into the L1 D–cache – an option specifies if the line is to be placed into the L1 D–cache or not. The option may specify that the prefetched line be consumed directly off the fill buffer and not cached. Under that option, the prefetched line is cached only when a later demand request uses it.

- The Intel design uses multiple prefetchers: mechanisms are in place to avoid any adverse impact of aggressive prefetching. A prefetch monitoring logic can throttle prefetches or momentarily suspend prefetching as needed.

- Implementation–dependent options can specify the prefetch distance (how much in advance to prefetch) and the amount of prefetched data (how many data items are prefetched at a time back–to–back as a burst).