Chapter 9: Caching

9.1 Concept of Cache

- SRAM doesn't scale well with large sizes but is much faster
- DRAM scales well with large sizes but is slow

- Hidden storage to stash information brought from memory
- Much smaller than main memory and much faster
- Look up in cache before resorting to main memory

9.2 Principle of Locality

Spatial locality - high probability of a program accessing adjacent memory locations `i-3, i- 2, i-1, i+1, i+2, i+3, ...`, if it accesses a location `i`Temporal locality - high probability of a program accessing in the near future, the same memory location `i` that it is accessing currently

9.3 Basic Terminology

Term	Definition
Hit	CPU finding the contents of the memory address in the cache
Miss	CPU fails to find contents in the cache
Miss Penalty	Time penalty associated with servicing a miss at any particular level of the memory hierarchy
Effective Memory Access Time (EMAT)	effective access time experienced by the CPU

9.4 Multilevel Memory Hierarchy

- We want to optimize two things: fast access and low miss rate
- Small cache = fast as possible
- Bigger cache = lower miss rate

This makes our EMAT calculation recursive (read lab slide 10 for more details)

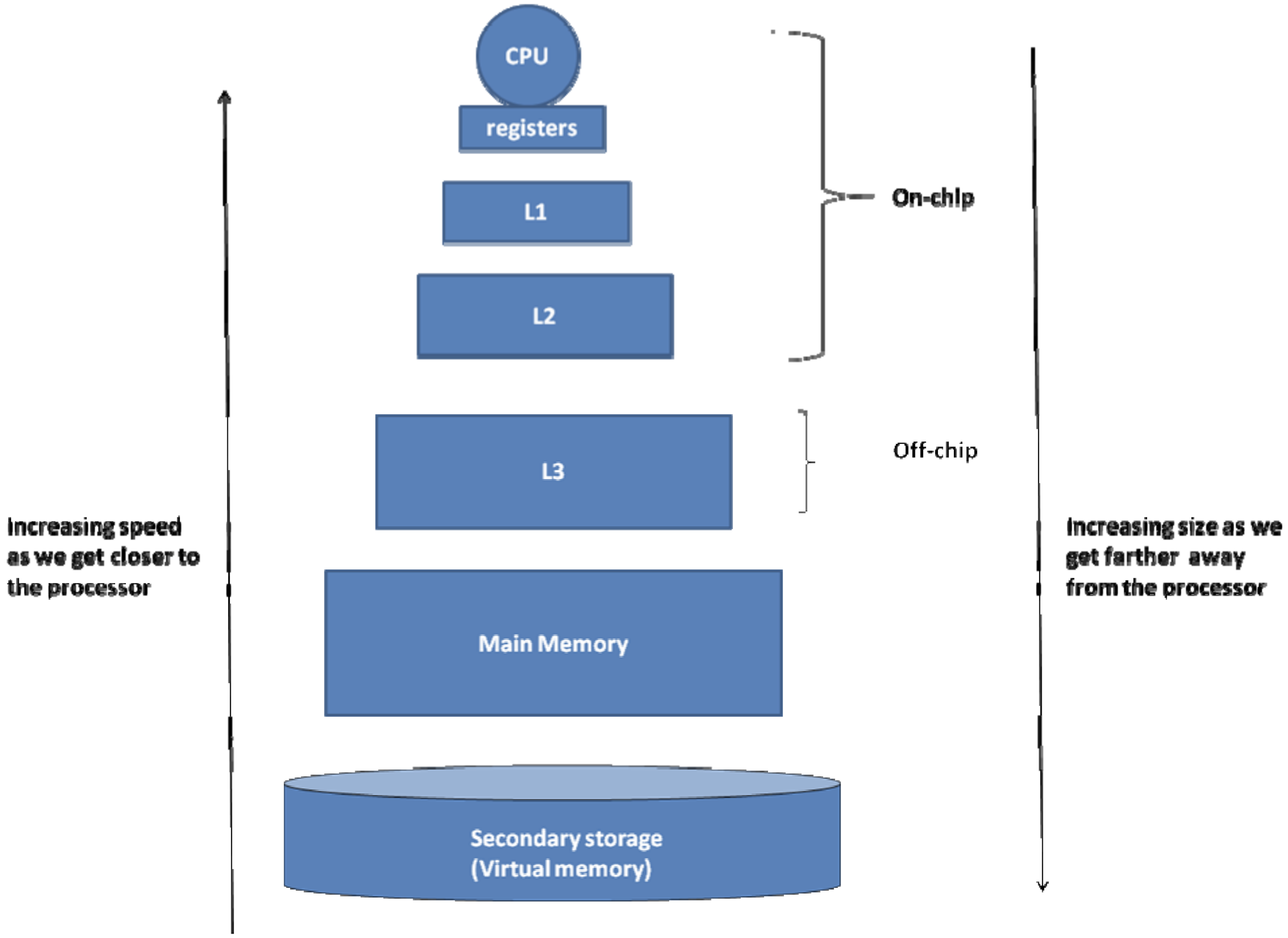

CPU
registers
L1
L2
L3
Main Memory
Secondary storage
(Virtual memory)
On-chip
Off-chip
Increasing speed as we get closer to the processor
Increasing size as we get farther away from the processor

Figure 9.1: Memory Hierarchy

9.6 Direct-mapped cache organization

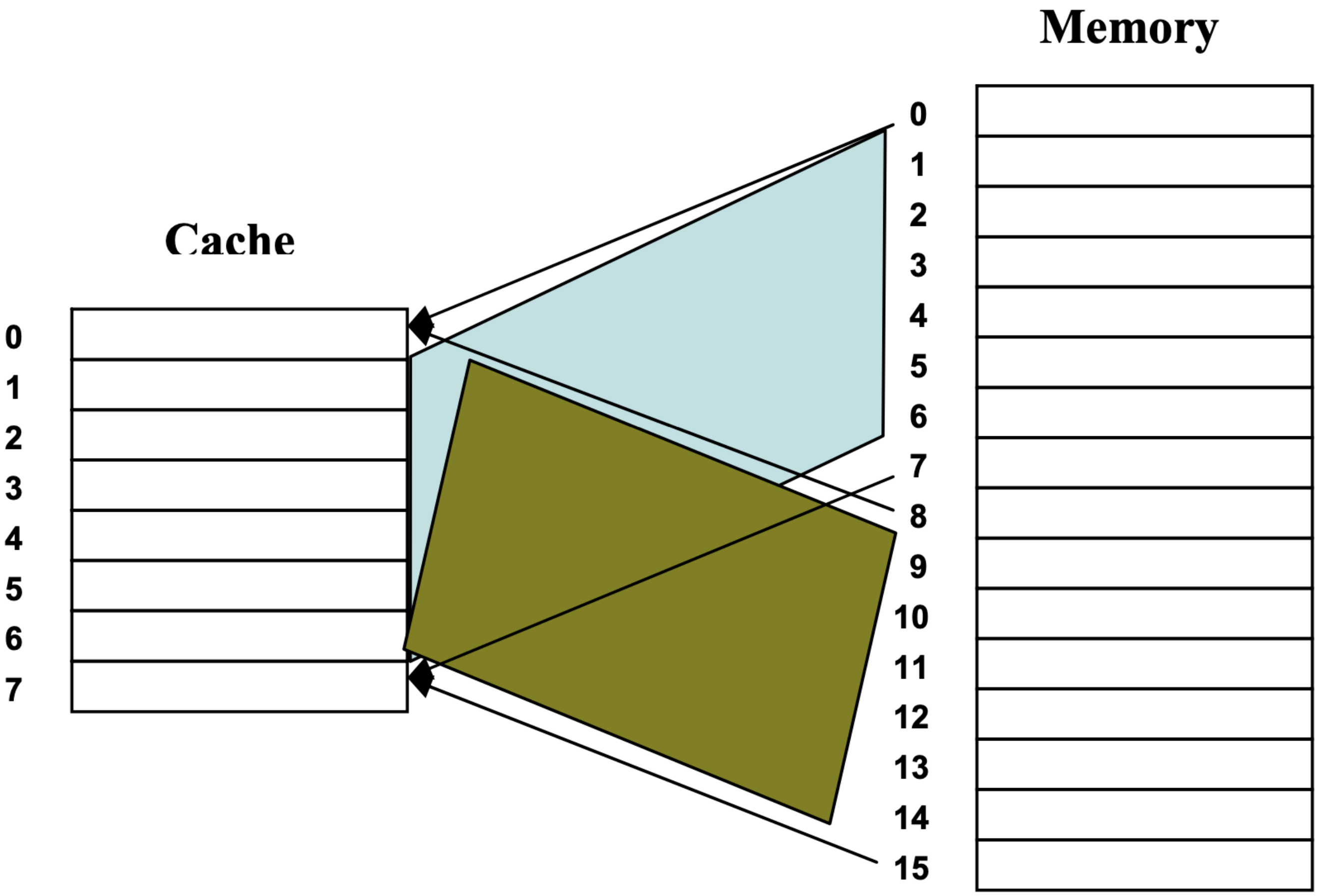


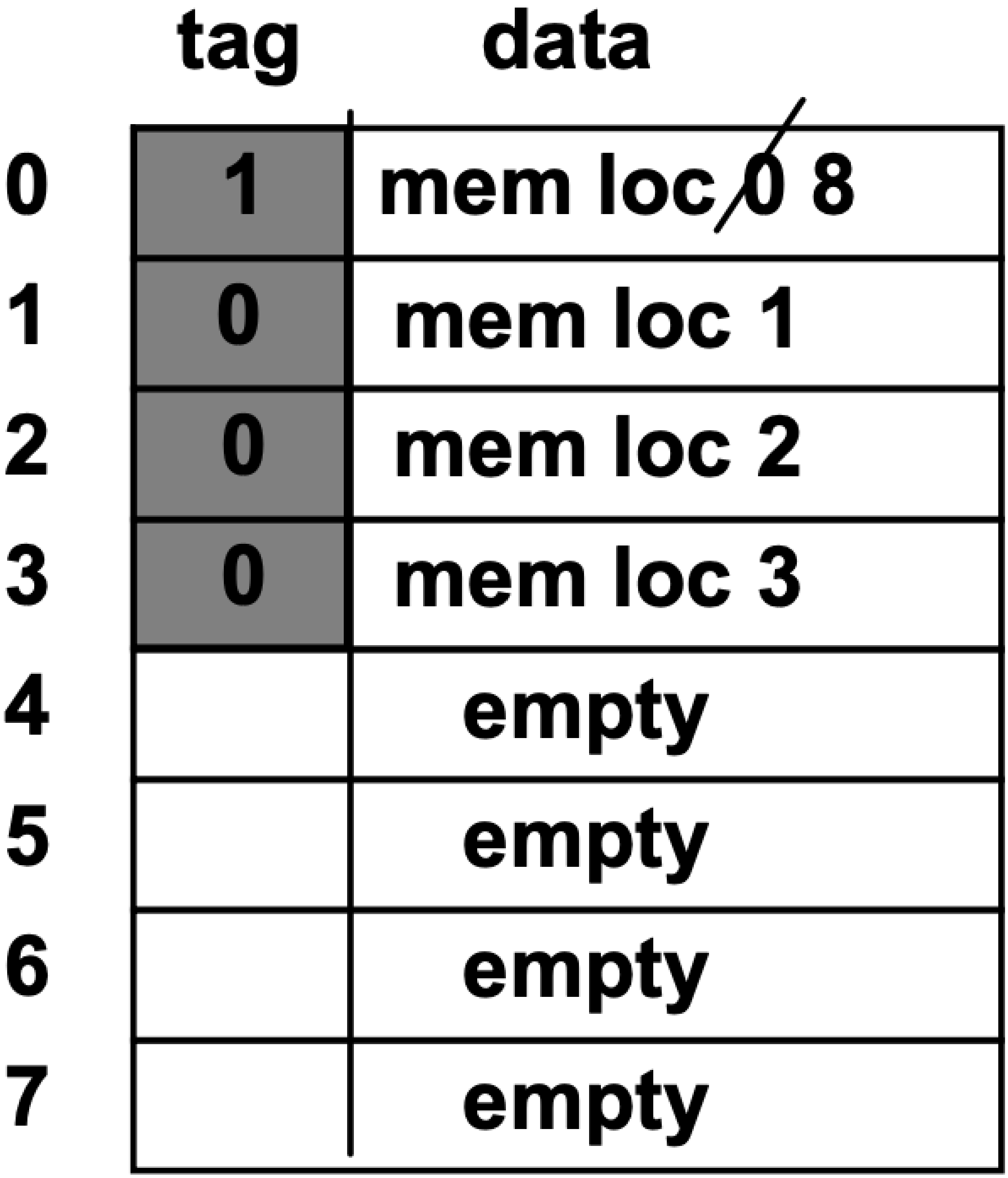
*Figure 9.2: Direct-Mapped Cache With Kishore Glitch On Cache*Types of Misses

- Compulsory Misses
  - Misses because the data has never loaded into the cache before
  - Trying to retrieve memory location 8 for the very first time
- Conflict Misses
  - Misses because the cache entry has been overridden by another memory location
  - There might be unused spots in the cache
  - Example: 15 and 7 map to the same location, so if 7 is put in the cache, and 15 overrides 7 in the cache, we will have a conflict miss when trying to get 15
- Capacity Misses
  - All spots in the cache are taken
  - Unable to find data without a conflict

Cache Lookup

$\text{cache-index} = \text{memory-address} \mod \text{cache-size}$Suppose the CPU gets data from memory address 8; `1000` is the memory address in binary, cache index is `000`. We need some way of knowing whether the contents of this cache entry are actually from memory location `8` or if its from memory location `0`.

- This additional information is called the **tag**



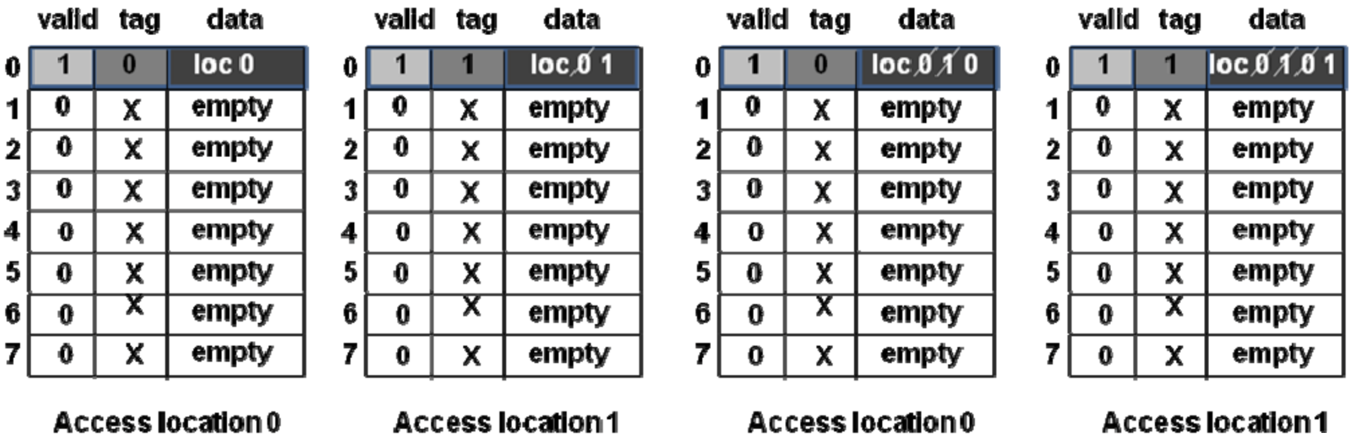
	tag	data
0	1	mem loc ~~0~~ 8
1	0	mem loc 1
2	0	mem loc 2
3	0	mem loc 3
4		empty
5		empty
6		empty
7		empty

Figure 9.3: Direct-mapped cache with valid field, tag field, and data field in each entry. A value of "X" in the tag field indicates a "don't care" condition

Fields of a Cache Entry

Valid	Tag	Data

Cache Tag	Cache Index



	valid	tag	data
0	1	0	loc 0
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 0

	valid	tag	data
0	1	1	loc ~~0~~ 1
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 1

	valid	tag	data
0	1	0	loc ~~0~~ ~~1~~ 0
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 0

	valid	tag	data
0	1	1	loc ~~0~~ ~~1~~ ~~0~~ 1
1	0	X	empty
2	0	X	empty
3	0	X	empty
4	0	X	empty
5	0	X	empty
6	0	X	empty
7	0	X	empty

Access location 1

Figure 9.4: Sequence of memory accesses if the cache index is chosen as the most significant bits of the memory address

Hardware for Direct Mapped Cache

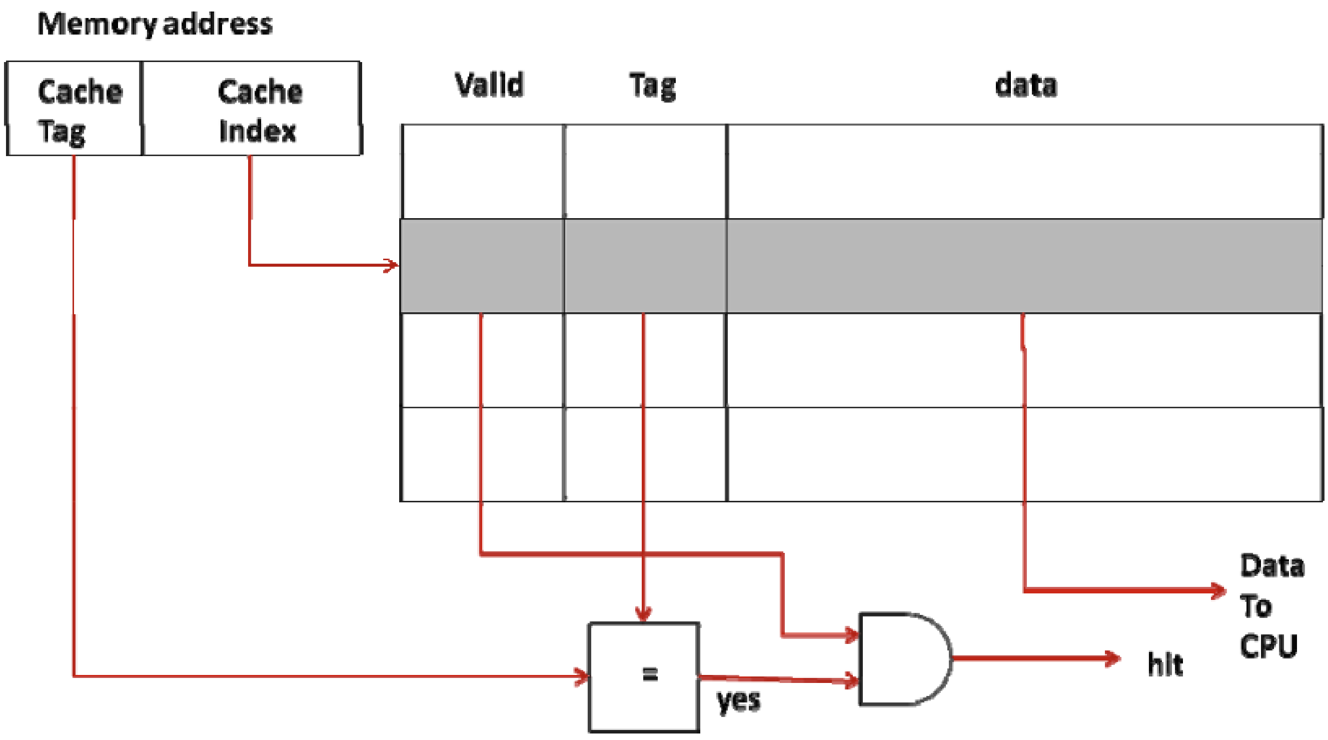


*Figure 9.5: Hardware for Direct Mapped Cache*The index part of the memory address picks out a unique entry in the cache (the shaded part in the diagram above)The

comparator compares the tag field of this entry against the tag part of the memory addressIf there is a match and the entry is valid, then it signals a hit

- Cache supplies the data field of the selected entry on the *cache line* (a.k.a. *cache block* and *cache entry*) to the CPU

The *valid bits*, and *tag fields*, called *meta-data*, are for managing the actual data contained in the cache, and represent a *space overhead*

Cache Tag	Cache Index	Byte Offset

- If the word-width is 32-bits and the architecture is byte addressable then the bottom 2-bits of the address form the byte offset

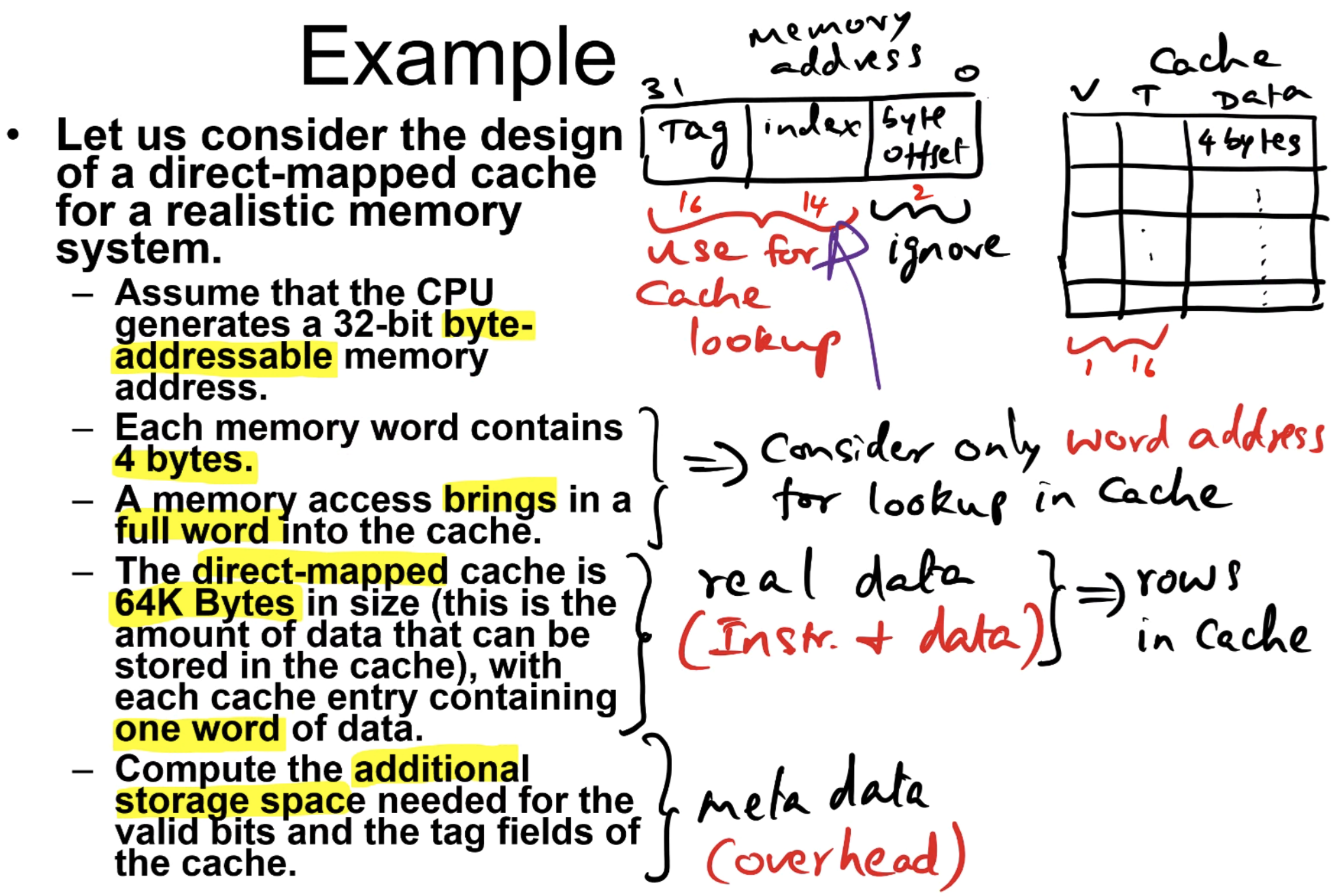
Example

- Assume that the CPU generates a 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- A memory access brings in a full word into the cache
- The direct-mapped cache is 64K bytes in size (amount of data that can be stored in the cache), with each cache entry containing one word of data
    - (real data = instr + data) ⇒ rows in cache
- Compute the additional storage space needed for the valid bits and the tag fields of the cache (a.k.a. the meta data or overhead)

$64K = 2^{16}$

- There are 16K entries which is 14 bits for the index

There are 64K Bytes and each row has 4 bytes, so $64 \div 4 = 16$



Example

- Let us consider the design of a direct-mapped cache for a realistic memory system.
  - Assume that the CPU generates a 32-bit byte-addressable memory address.
  - Each memory word contains 4 bytes.
  - A memory access brings in a full word into the cache.
  - The direct-mapped cache is 64K Bytes in size (this is the amount of data that can be stored in the cache), with each cache entry containing one word of data.
  - Compute the additional storage space needed for the valid bits and the tag fields of the cache.

⇒ Consider only word address for lookup in Cache

real data (Instr. + data) ⇒ rows in Cache

meta data (overhead)

Turningpoint A direct mapped cache has a one-to-one mapping between a memory location and a cache locationIn a direct-mapped cache with a t-bit tag there is a 1-bit tag comparator for the entire cache

9.7 Repercussion on Pipelined Processor

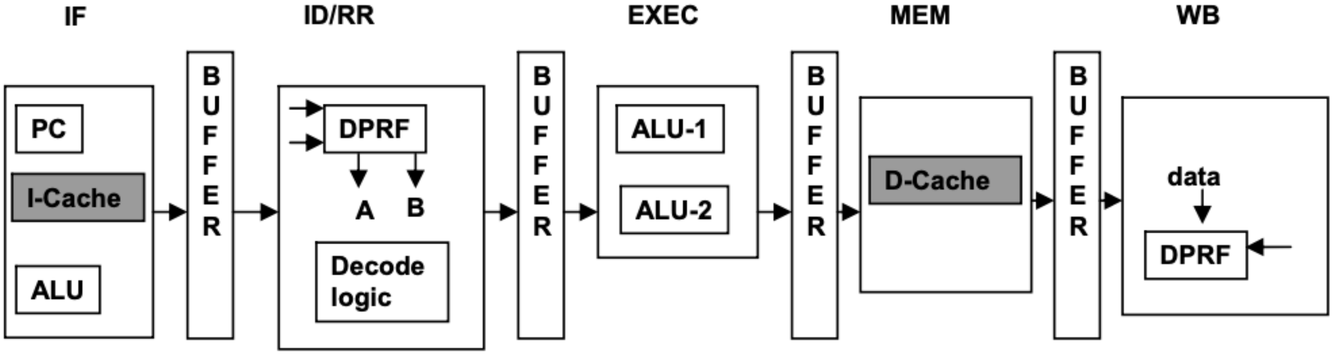


Figure 9.6: Pipelined Processor With Caches

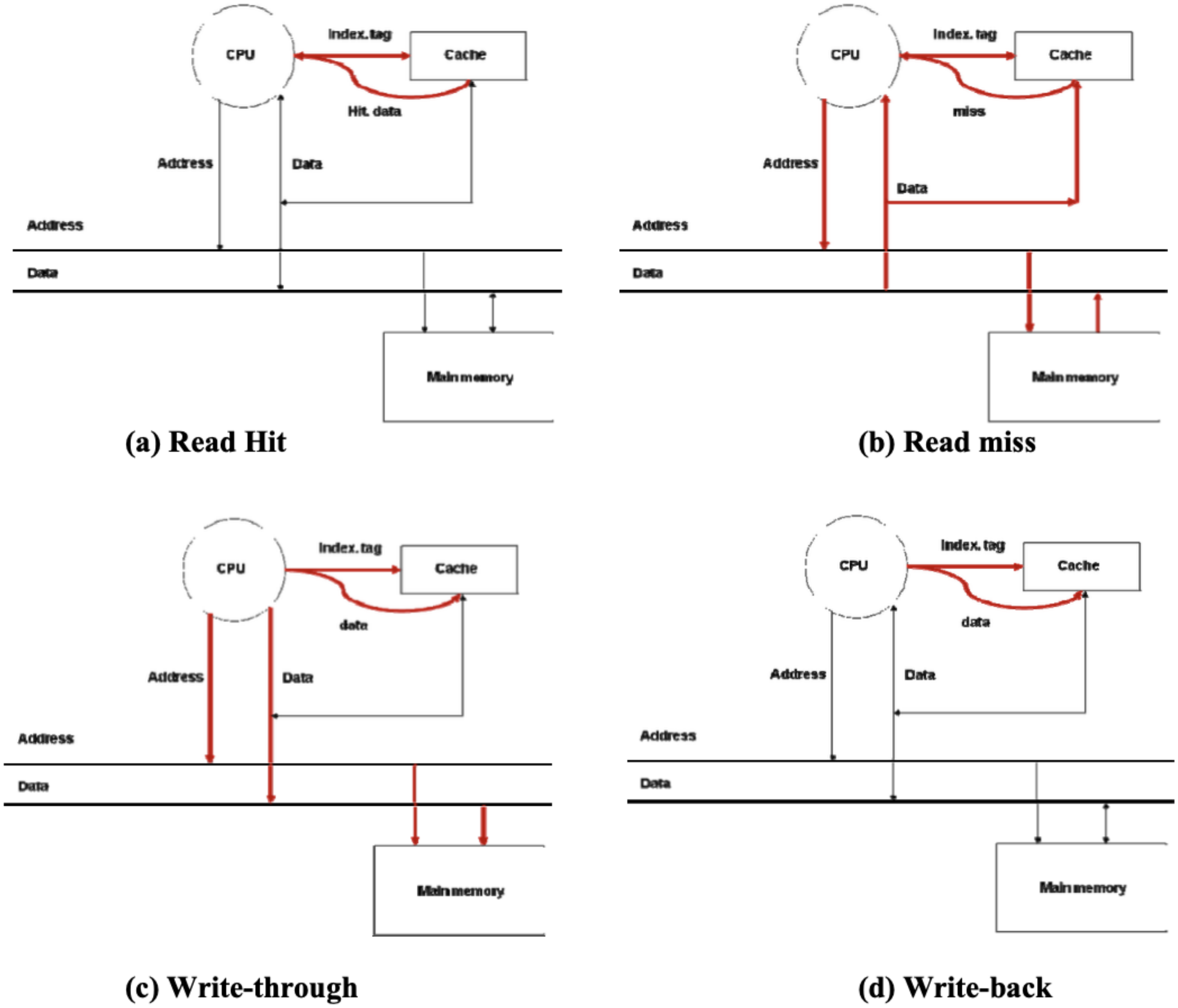
- IF and MEM stages to have comparable cycle times to the other stages of the pipeline

1. Miss in the IF stage: IF stage sends the reference to the memory to retrieve the instruction. As we know, the memory access time may be several 10’s of CPU cycles. Until the instruction arrives from the memory, the IF stage sends NOPs (bubbles) to the next stage
2. Miss in the MEM stage: Misses in the D-Cache are relevant only for ***memory reference instructions (load/store)***. Similar to the IF stage, a miss in the MEM stage results in NOPs to the WB stage until the memory reference completes. It also ***freezes*** the preceding stages from advancing past the instructions they are currently working on

Memory Stall - processor cycles wasted due to waiting for a memory operation to complete

- Read stall
- Write stall

9.8 Cache Read/Write Algorithms



(a) Read Hit

(b) Read miss

(c) Write-through

(d) Write-back

Figure 9.7: CPU, Cache, Memory interactions for reads and writes

Read Access to the Cache from CPU

1. Step 1: CPU sends the index part of the memory address to the cache. Cache does a lookup, and if successful (hit), it supplies data to CPU. If miss, CPU sends the address on the memory bus to the main memory. In principle, all of these actions happen in the same cycle
2. Step 2: Upon sending the address to the memory, the CPU sends NOPs down to the subsequent stage until it receives the data from the memory. ***Read stall*** is defined as the number of processor clock cycles wasted to service a read-miss. This could take several CPU cycles depending on the memory speed. The cache allocates a cache block to receive the memory block. Eventually, the main memory delivers the data to the CPU and simultaneously updates the allocated cache block with the

data. The cache modifies the tag field of this cache entry appropriately and sets the valid bit

Write Access to The Cache From CPU

Write Through Policy

1. Step 1: On every write (store instruction in LC-2200), the CPU simply writes to the cache. No need to check the valid bit or the cache tag – the cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline.
2. Step 2: Simultaneously, the CPU sends the address and data to the main memory. This of course is problematic in terms of performance since memory access takes several CPU cycles to complete. To alleviate this performance bottleneck, it is customary to include a write-buffer in the datapath between the CPU and the memory bus. The write-buffer is a small hardware store (similar to a register file) to smoothen out the speed disparity between the CPU and memory. As far as the CPU is concerned, the write operation is complete as soon as it places the address and data in the write buffer. Thus, this action also happens in the MEM stage of the pipeline without stalling the pipeline.
3. Step 3: The write-buffer completes the write to the main memory independent of the CPU. Note that, if the write buffer is full at the time the processor attempts to write to it, then the pipeline will stall until one of the entries from the write buffer has been sent to the memory. Write stall is defined as the number of processor clock cycles wasted due to a write operation (regardless of hit or a miss in the cache).

1. Write Allocate - intuition is that the data being written will be needed by the program in the near future, so we put it in the cache (even if it means allocating a cache block and bringing in the missing memory block)
2. No Write Allocate - argument is to have the write stage finish quickly, so processor simply places the write in the write buffer to complete the operation needed in the MEM stage of the pipeline

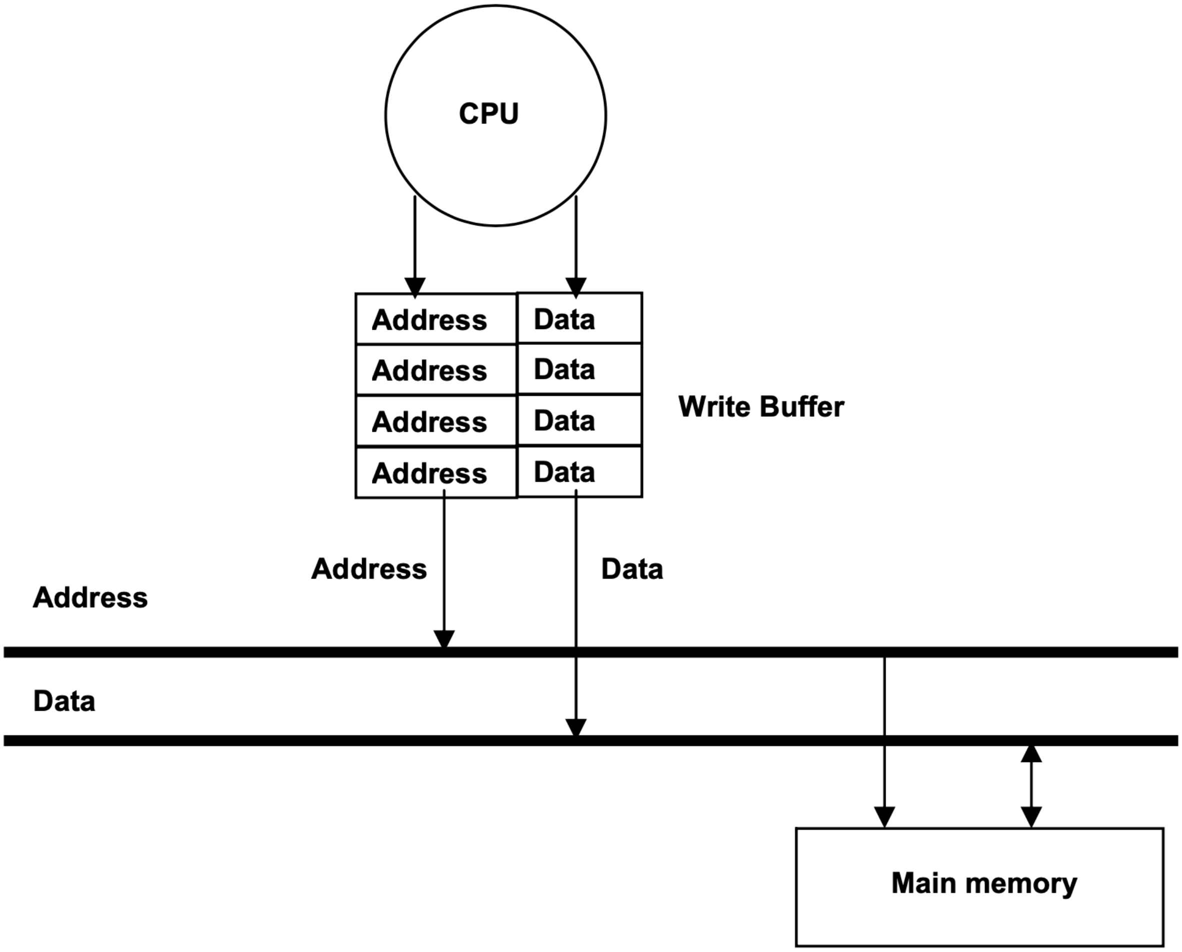


Figure 9.8: 4-Element Write-Buffer for Write Through Policy

Write Back Policy

Step 1: CPU writes to the cache. The cache updates the tag field of the corresponding entry and sets the valid bit. These actions happen in the MEM stage of the pipeline**Step 2:** The contents of this chosen cache entry and the corresponding memory location are inconsistent with each other (doesn't matter though because CPU just gets latest value from cache)**Step 3:** May become necessary to replace an existing cache entry to make room for a memory location not currently present in the cacheWe introduce the *dirty bit* to each cache entry

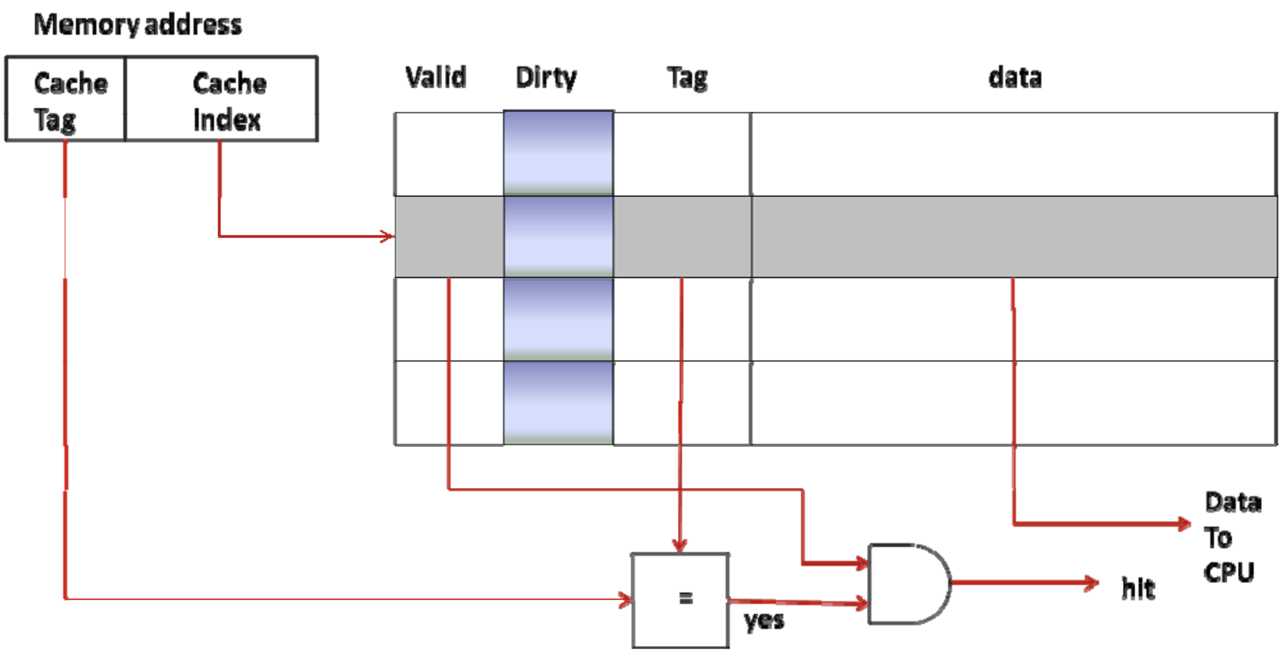


Figure 9.9: Direct-Mapped Cache With Write-Back Policy

- clears the bit upon processing a miss that brings in a memory location into this cache entry
- sets the bit upon a CPU write operation
- writes back the data in the cache into the corresponding memory location upon replacement. Note that this action is similar to writing back a physical page frame to the disk in the virtual memory system

9.9 Dealing With Cache Misses In The Pipeline

Read miss in MEM Stage

- Introduce busy bit

Write miss in MEM Stage

- If write-allocate, handle as read-miss
- If write-back, write stalls are inevitable

Effect of memory stalls due to cache misses on pipeline performance

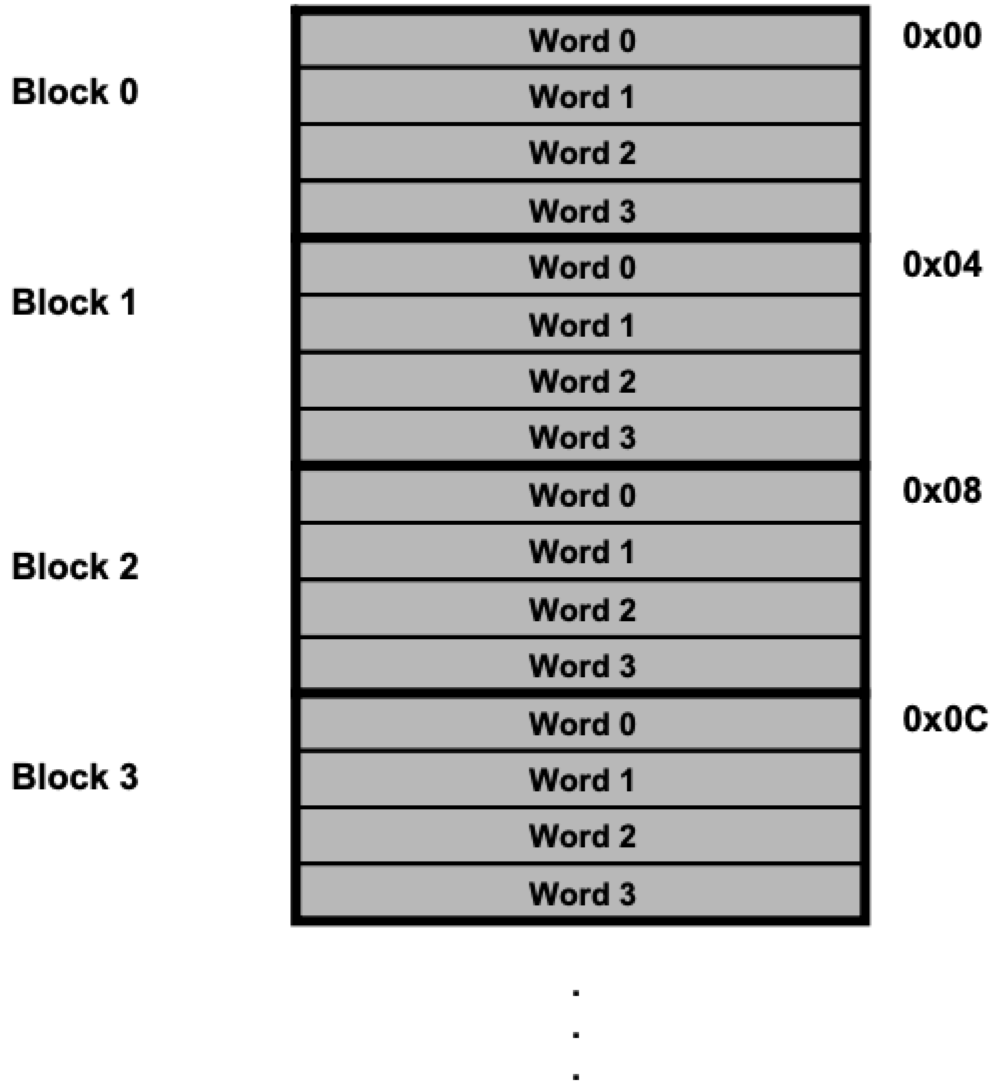
Updated Equations Execution time $=$ (Number of instructions executed $\cdot$ (CPIAvg $+$ Memory $-$ stallsAvg)) $\cdot$ clock cycle timeEffective CPI $=$ CPIAvg $+$ Memory $-$ stallsAvgTotal memory stalls $= \text{Number of instructions} \cdot \text{Memory} - stalls_{Avg}$

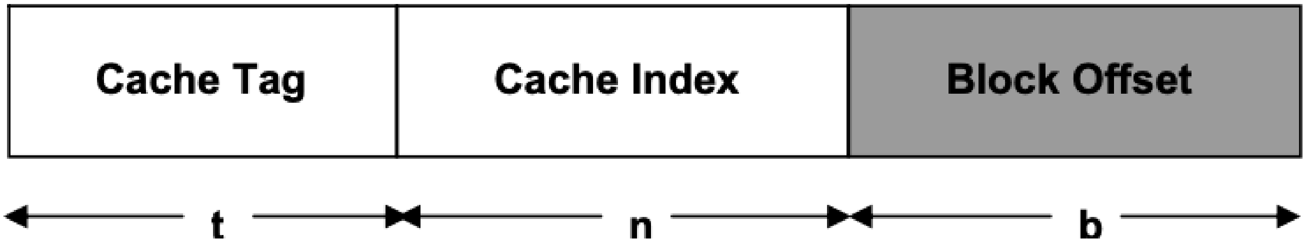
$\text{Memory} - stalls_{Avg} = \text{misses per } instruction_{Avg} * \text{miss} - penalty_{Avg}$

9.10 Exploiting Spatial Locality to Improve Performance

The basic idea is to bring adjacent memory locations into the cache upon a miss for a memory location i

- To exploit spatial locality in the cache design, we decouple the *unit of memory access* by an instruction from the *unit of memory transfer* between the memory and the cache
- We refer to the unit of transfer between the cache and the memory as the ***block size***
- Upon a miss, the cache brings an entire *block of block size* bytes that contains the missing memory reference





- $2^{16} = 65,536$

If L represents the number of lines in a direct-mapped cache of size S and B bytes per block, with b representing the number of least significant bits of the memory field, t representing the number of most significant bits of the memory field, n representing the middle bits, and a representing the total number of bits, then$b = log_2 B L = S \div B n = log_2 L$
$t = a - (b + n)$

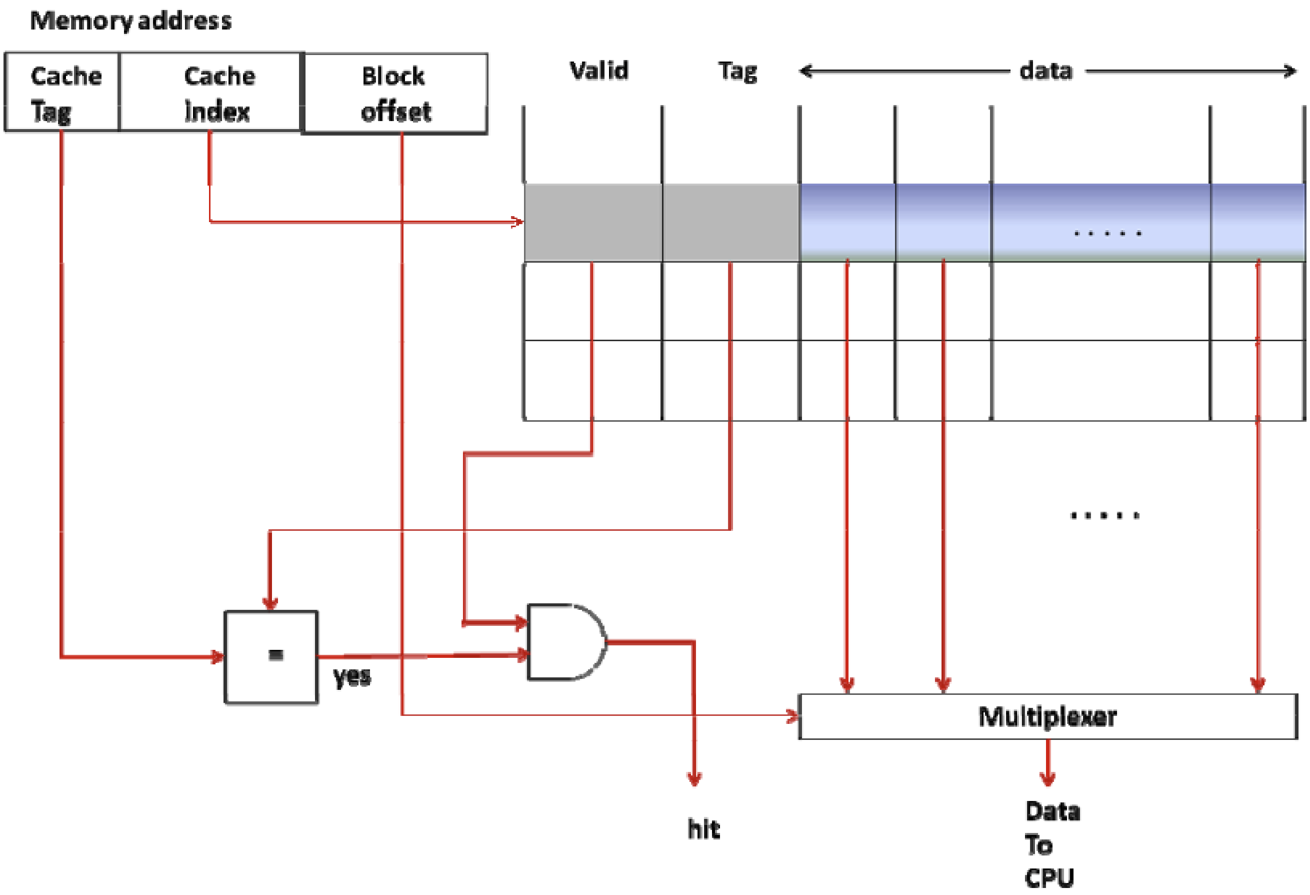


Figure 9.12: Multi-word Direct-mapped Cache Organization

1. Lookup: The index for cache lookup comes from the middle part of the memory address. The entry contains an entire block (if it is a hit as determined by the cache tag in the address and the tag stored in the specific entry). The least significant b bits of the address specify the specific word (or byte) within that block requested by the processor. A multiplexer selects the specific word (or byte) from the block using these b bits and sends it to the CPU.
2. Read: Cache brings out the entire block corresponding to the cache index. If the tag comparison results in a hit, then the multiplexer selects the specific word (or byte) within the block and sends it to the CPU. If miss then the CPU initiates a block transfer from the memory
3. Write: Modify the write algorithm since there is only 1 valid bit for the entire cache line. Similar to the read-miss, the CPU initiates a block transfer from the memory upon a write-miss

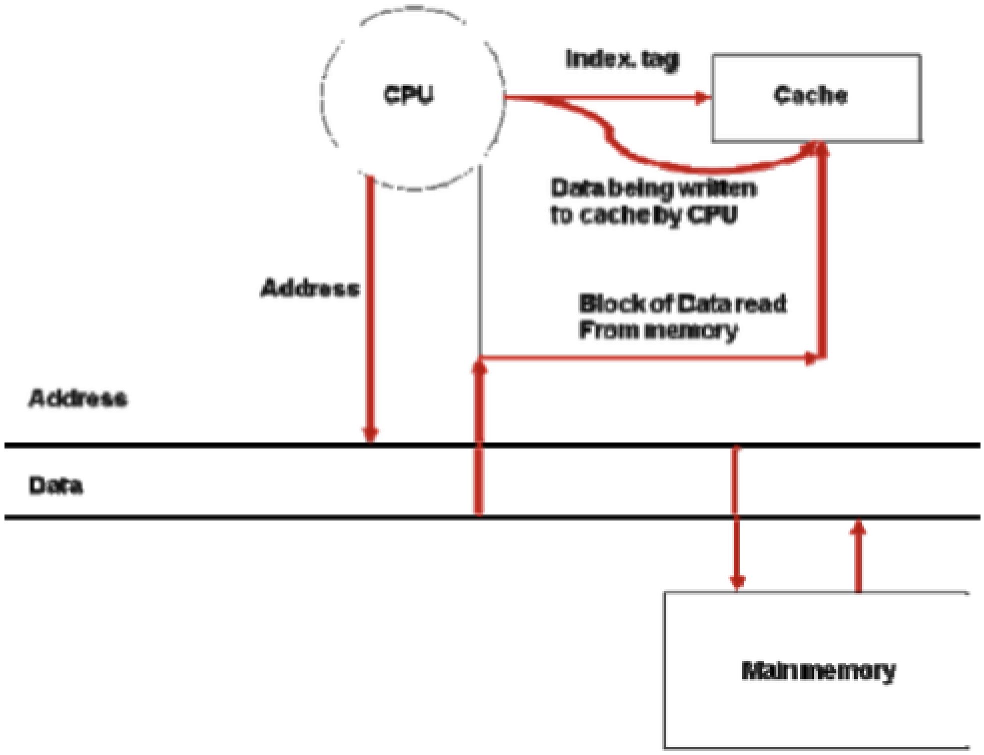


Figure 9.13: CPU, Cache, and Memory Interactions For Handling a Write-miss

Example Consider a multi-word direct-mapped cache with a data size of 64 Kbytes . The CPU generates a 32-bit byte-addressable memory address. Each memory word contains 4 bytes . The block size is 16 bytes . The cache uses a write-back policy with 1 dirty bit per word . The cache has one valid bit per data block .

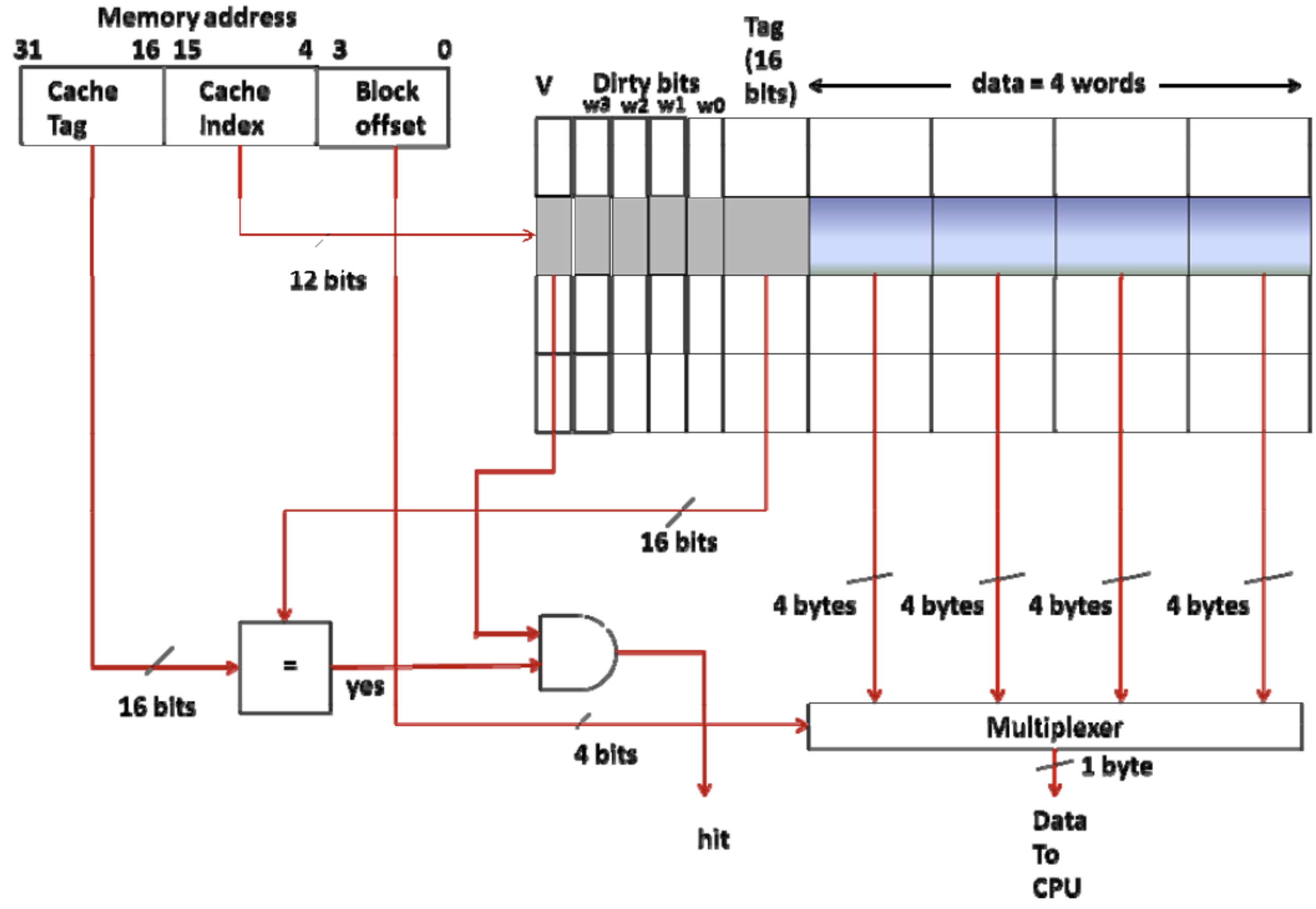


*Figure 9.14: Multi-word Direct Mapped Cache With Write-Back*a). How does the CPU interpret the memory address?Answer Block size $B = 16$ bytes, therefore $b = log_2 16 = 4$. We need four bits for the block offset.Number of cache lines $L = 64KBytes/16bytes = 64 * 1000/16 = 4096$

- Data size / block size

Number of index bits $n = log_2 L = log_2 4096 = 12$Number of tag bits $t = a - (n + b) = 32 - (12 + 4) = 16$b). Compute the total amount of storage for implementing the cache (i.e. actual data plus the meta-data).Answer Each cache line will have

- 16 bits x 8 bits/byte =128 bits for data
- 1 Valid bit
- 4 dirty bits (1 for each word)
- 16 bits for tag

Total amount of space for the cache = 149 × 4096 cache lines = 610,304 bitsThe space requirement for the meta-data = total space – actual data = 610,304 – 64Kbytes =

610,304 – 524,288 = 86,016The space overhead = meta-data/total space = 86,016/610,304 = 14%

Performance Implications of Increased Blocksize

- Will miss rate keep decreasing forever? ***NO***
- Will the overall performance of the processor go up as we increase the block size?

  > ***DEPENDS***

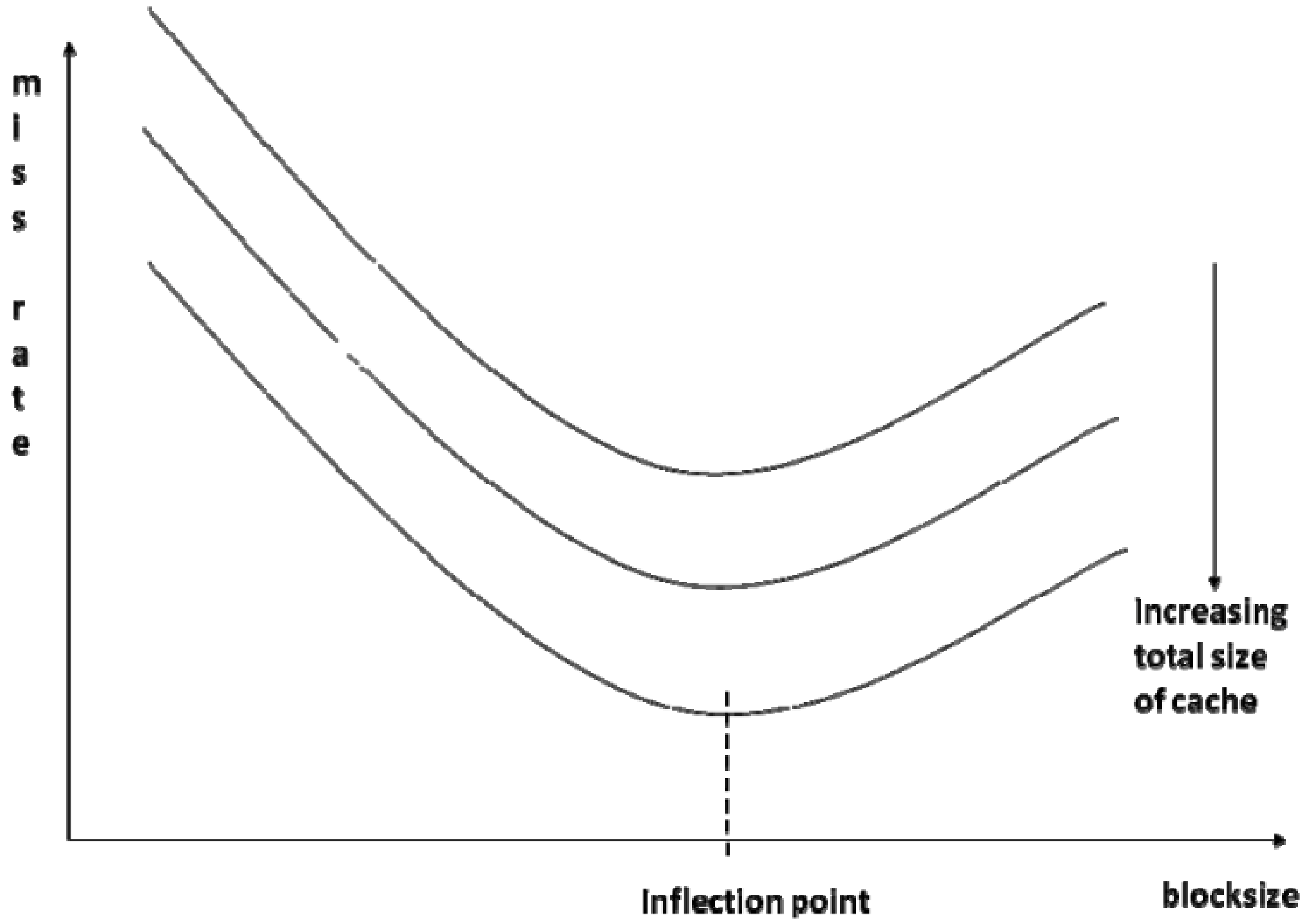


Figure 9.15: Expected behavior of miss-rate as a function of blocksize

- Increased block size could negatively interact with the miss penalty
- The larger the block size, the larger the time penalty for the transfer from the

  > memory to the cache on misses, thus increasing the memory stalls

Choice of block size affects the balance between latency for a *single instruction* (that incurred the miss) and throughput for the program as a whole, by reducing the potential misses for other *later instructions* that might benefit from the exploitation of spatial locality

9.11 Flexible Placement

- unable to place a new memory location in a currently unoccupied slot in the cache

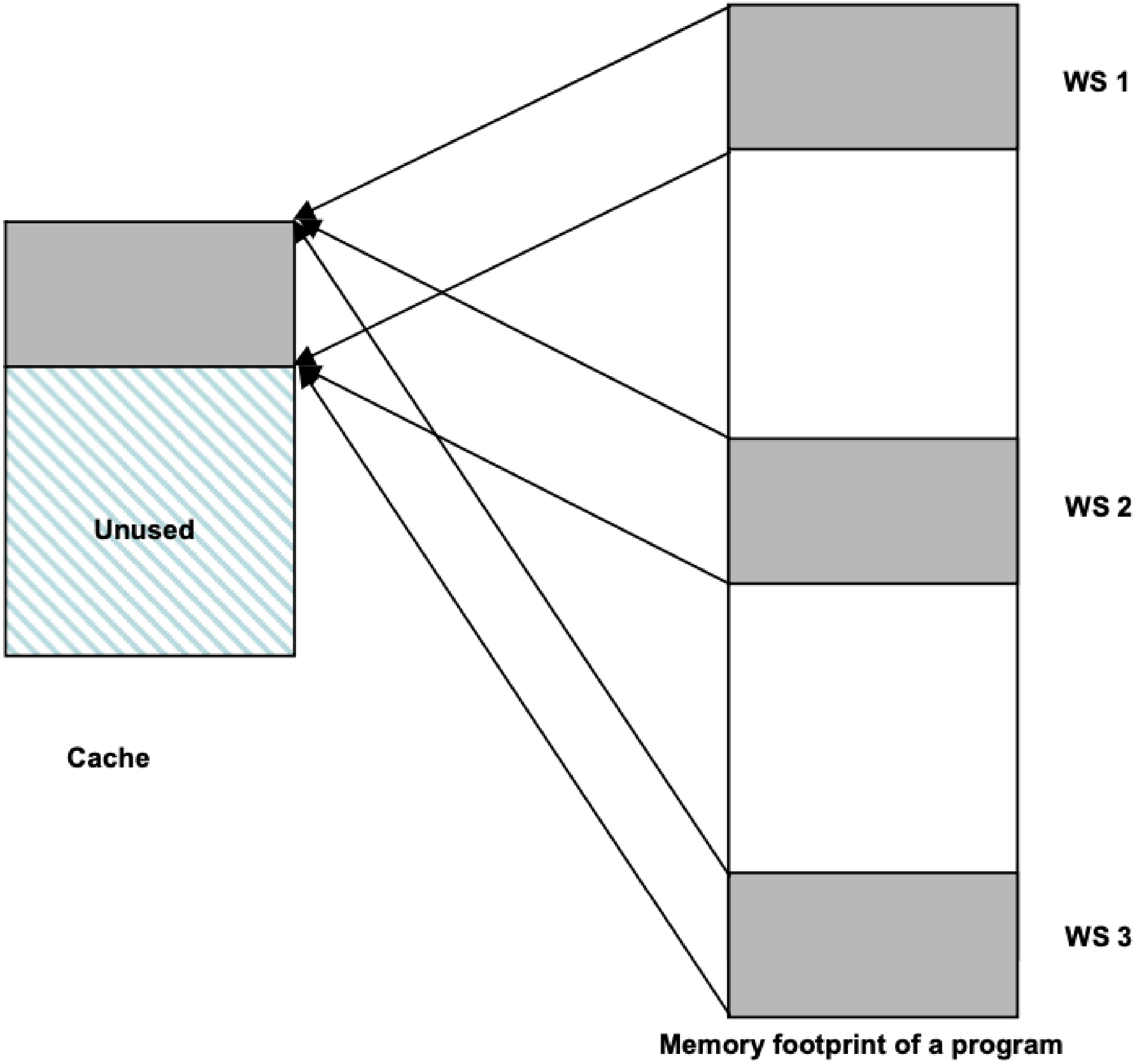


Figure 9.16: Different working sets of a program occupying the same portion of a direct-mapped cache

- Due to the rigid mapping, the working sets displace one another from the cache resulting in poor performance
- We would want all three working sets of the program to reside in the cache so that there will be no more misses beyond the compulsory ones

Fully Associative Cache

- A cache block can host any memory block
- ***Compulsory*** and ***capacity*** misses are the only kind of misses encountered with this organization

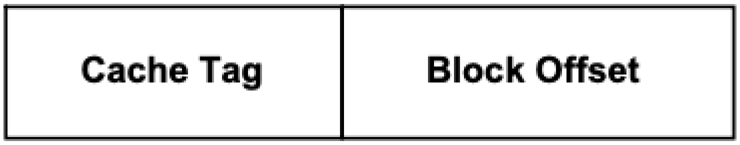
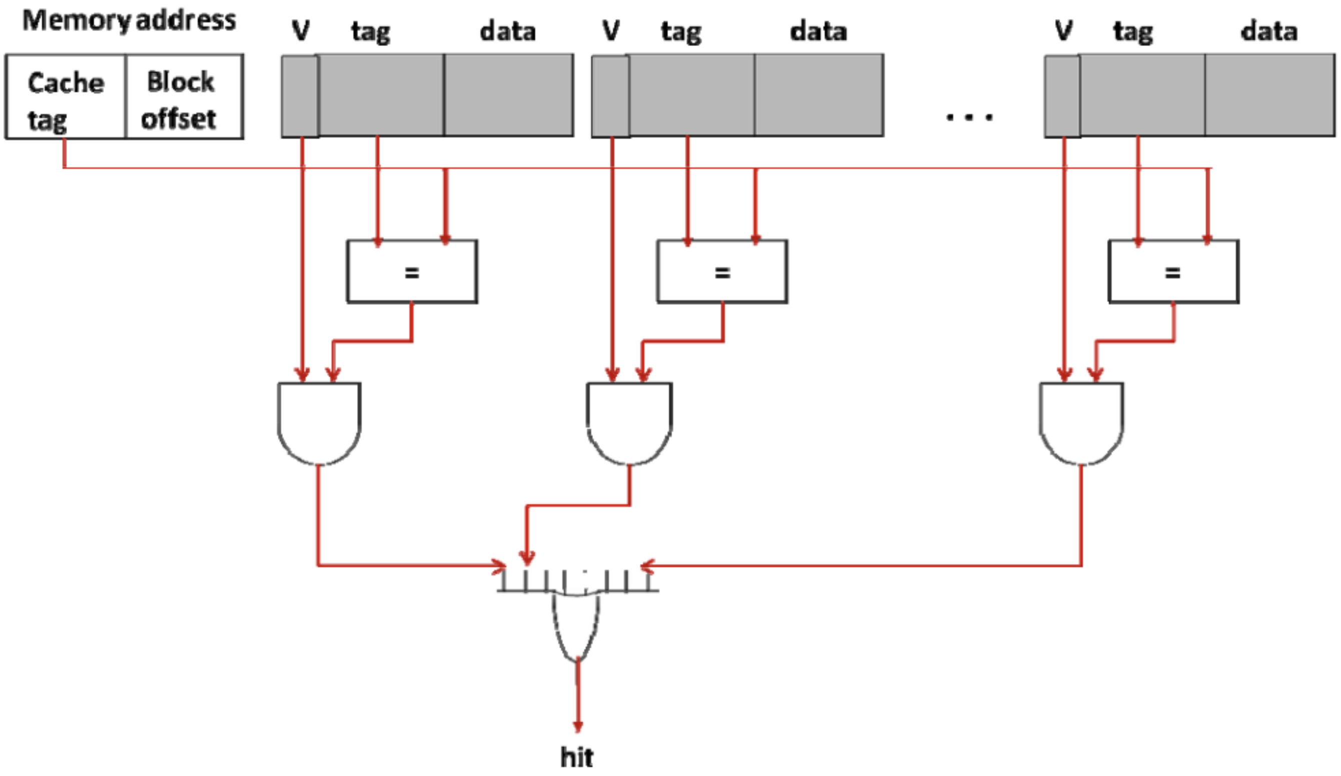


*Figure 9.17: Memory address interpretation for a fully associative cache*To perform a look up, the cache has to search through all the entries to see if there is a match between the cache tag in the memory address and the tags in any of the valid entries

- Search through sequentially? Too slow
- Add comparator for everything instead



Turningpoint In a fully associative cache with 64K bytes of data, 64 bytes per block and with a t-bit tag, there are 1K t-bit tag comparators

- 64K bytes of data / 64 bytes per block = 1K

Downsides

- Very expensive hardware wise
- Misses in the cache will inevitably result in replacement of something that is already present in the cache
    - Miss overshadows flexibility advantage of fully associative

Set Associative Cache

Memory block can be associated with a *set of cache blocks*

- Ex: a 4-way set associative cache gives a memory block one of four possible homes in the cache
- The ***degree of associativity*** is defined as the number of homes that a given memory block has in the cache

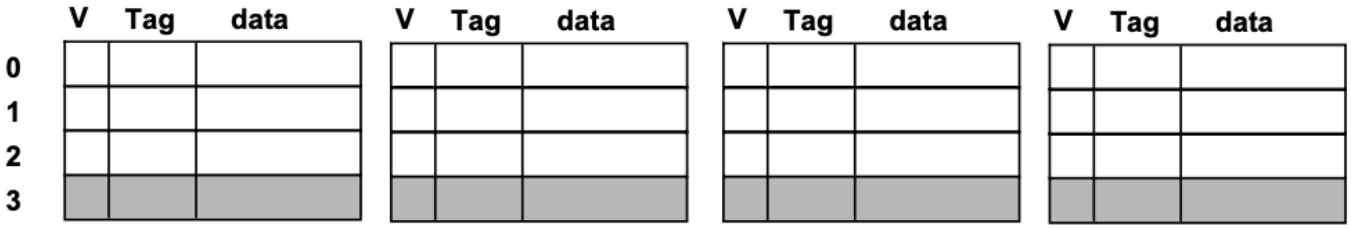


Figure 9.19: 4-way associative cache

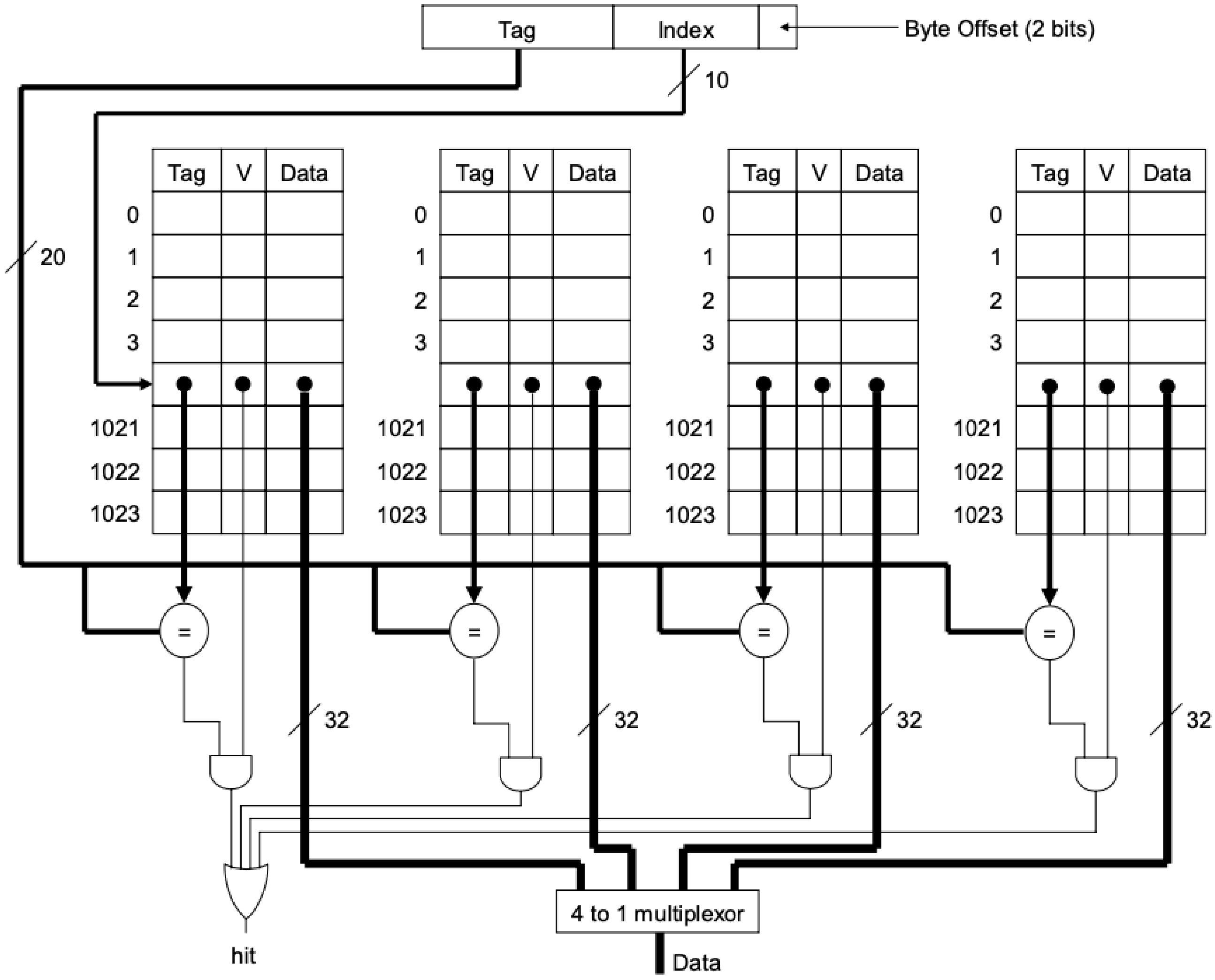
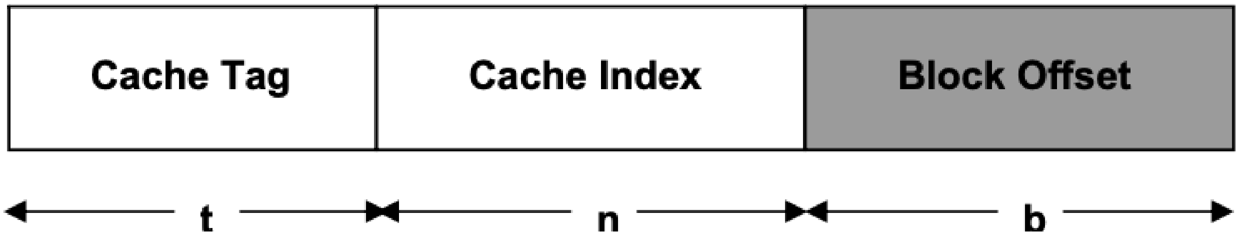


Figure 9.20: 4-way set associative cache organization



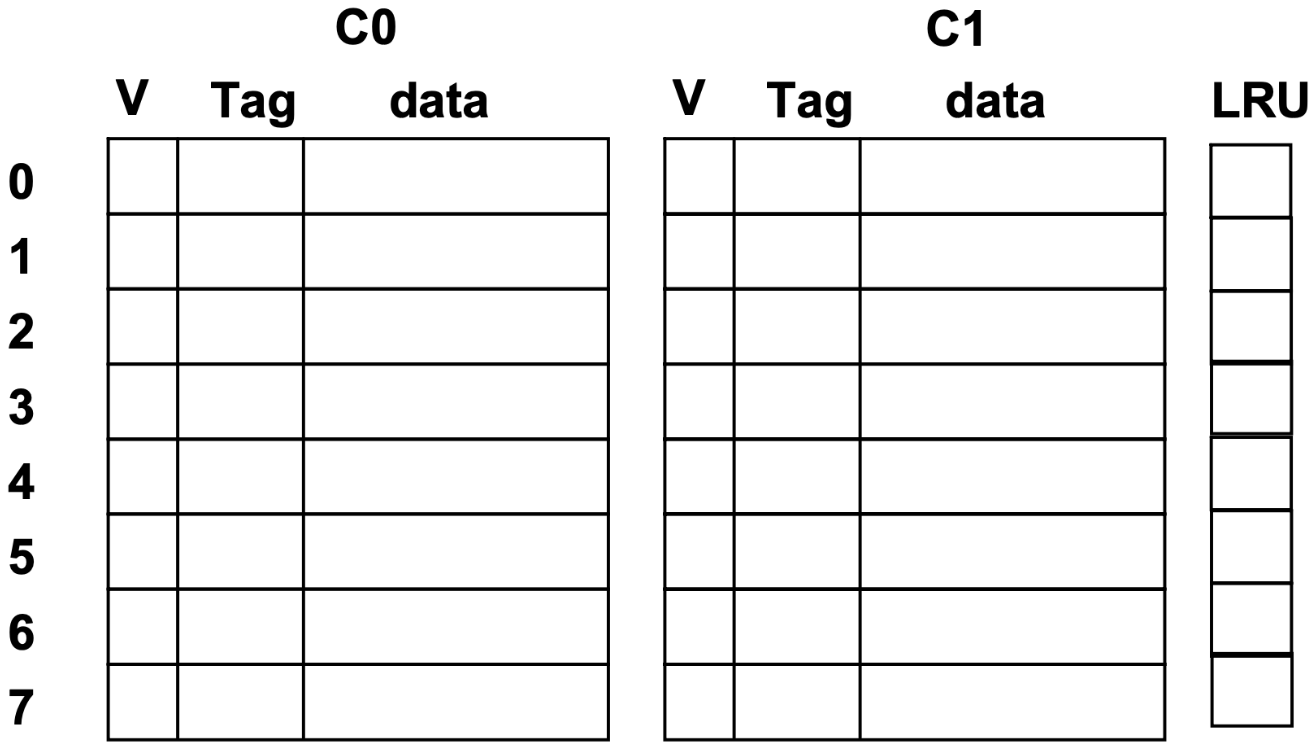
Turningpoint In a 4-way set associative cache with 64K bytes of data, 64 bytes per block and with a t-bit tag, there are four t-bit tag comparators

9.12 Instruction and Data Caches

- Combine means increased hit rate
- Combine also means we have a structural hazard

9.14 Cache Replacement Policy

- Fully associative cache: LRU policy across all the blocks
- Set associative cache: limited to the set that will house the currently missing memory reference



The candidate for replacement is the block in C0 if the LRU bit is 1; it is the block in C1 if the LRU bit is 0 - Small time penalty setting the LRU bit - Hardware is simple for two way, gets more complex with more and more sets

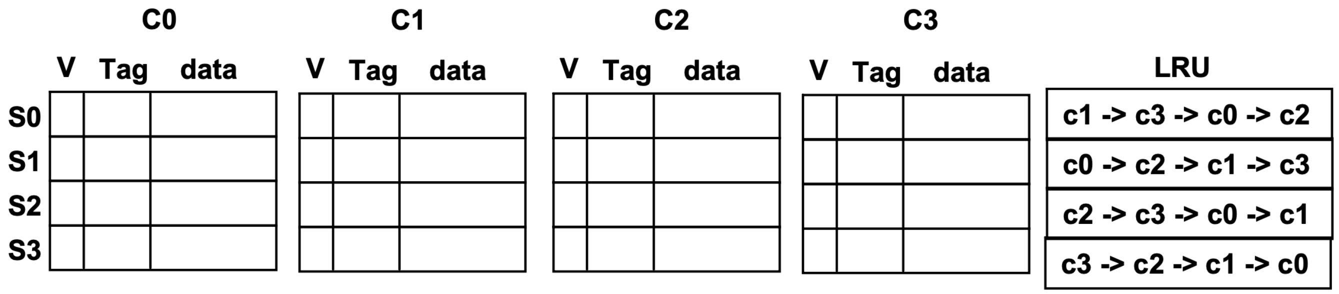


Figure 9.31: LRU information for a 4-way set associative cache

	LRU for set S0	
Access to C1	c1 -> c3 -> c0 -> c2	Replacement candidate: current block in C2
Access to C2	c2 -> c1 -> c3 -> c0	Replacement candidate: current block in C0
Access to C2	c2 -> c1 -> c3 -> c0	Replacement candidate: current block in C0
Access to C3	c3 -> c2 -> c1 -> c0	Replacement candidate: current block in C0
Access to C0	c0 -> c3 -> c2 -> c1	Replacement candidate: current block in C1

In general, for a n-way associative set, we need $n!$ states, so the counter gets big very fast.

9.15 Recapping Types of Misses

Compulsory/Cold Misses

- Similar to automobile engine being cold at the start
- Misses because it is the first time the program is being accessed

Capacity Miss

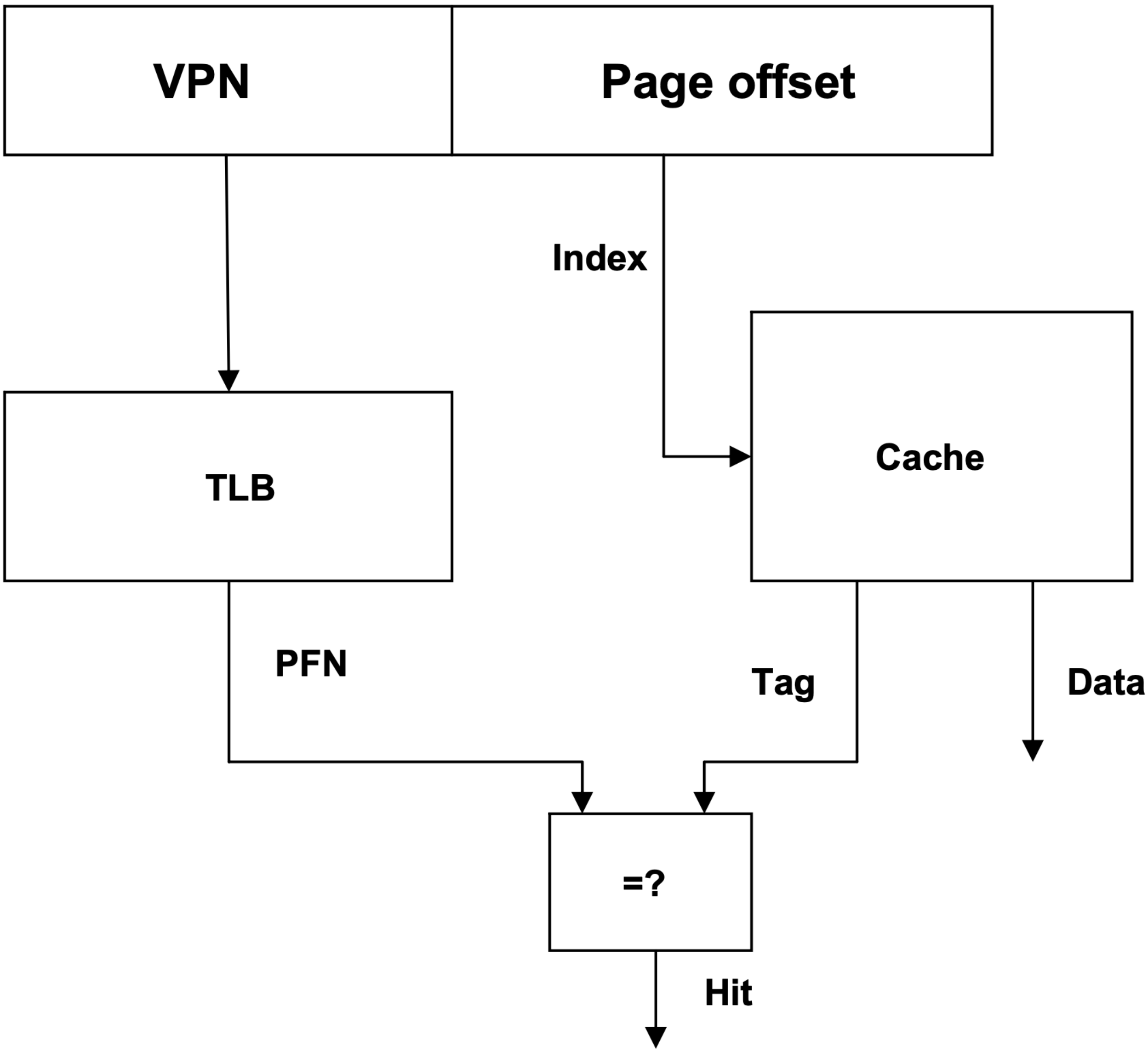
- CPU incurs a miss on a memory location X that used to be in the cache previously
- Might happen because cache is full and so we had to evict it

Conflict Miss

- CPU incurs a miss on a memory location X that used to be in the cache previously
- Cache is not full but the mapping strategy forces X to be brought into a cache line that is currently occupied by some other memory location
- Doesn't exist in fully associative cache

9.18 Virtually Indexed Physically Tagged Cache

- Observation: Offset is the same in the virtual and physical address
- The name comes from the fact that the tag is comes from the physical address
- Allows the processor to have a larger virtually indexed physically tagged cache independent of the page size



9.20 Main Memory

Simple Main Memory

- CPU simply sends the block address to the physical memory
- Physical memory internally computes the successive addresses of the block, retrieves the corresponding words from the DRAM, and sends them one after the other back to the CPU

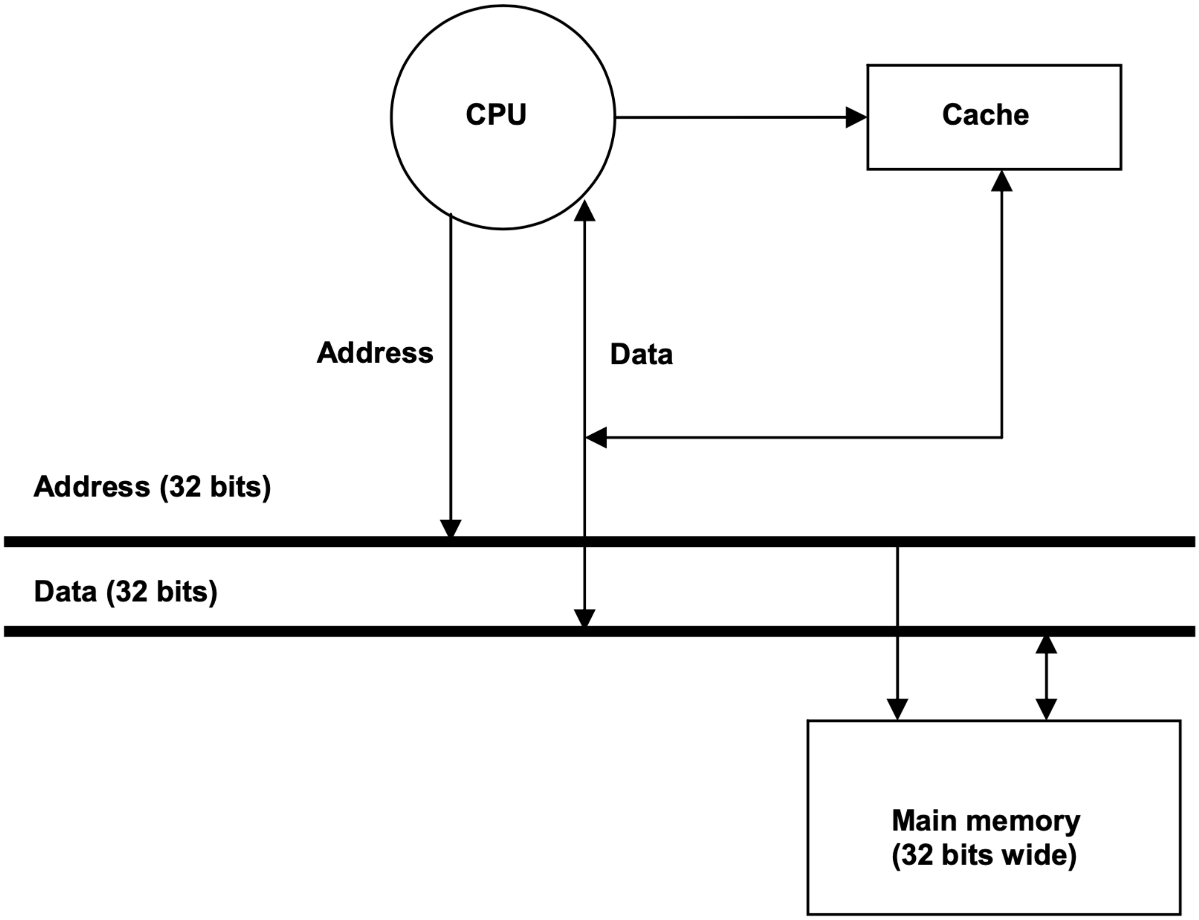


Figure 9.25: A simple memory system

Main Memory and Bus To Match Cache Block Size

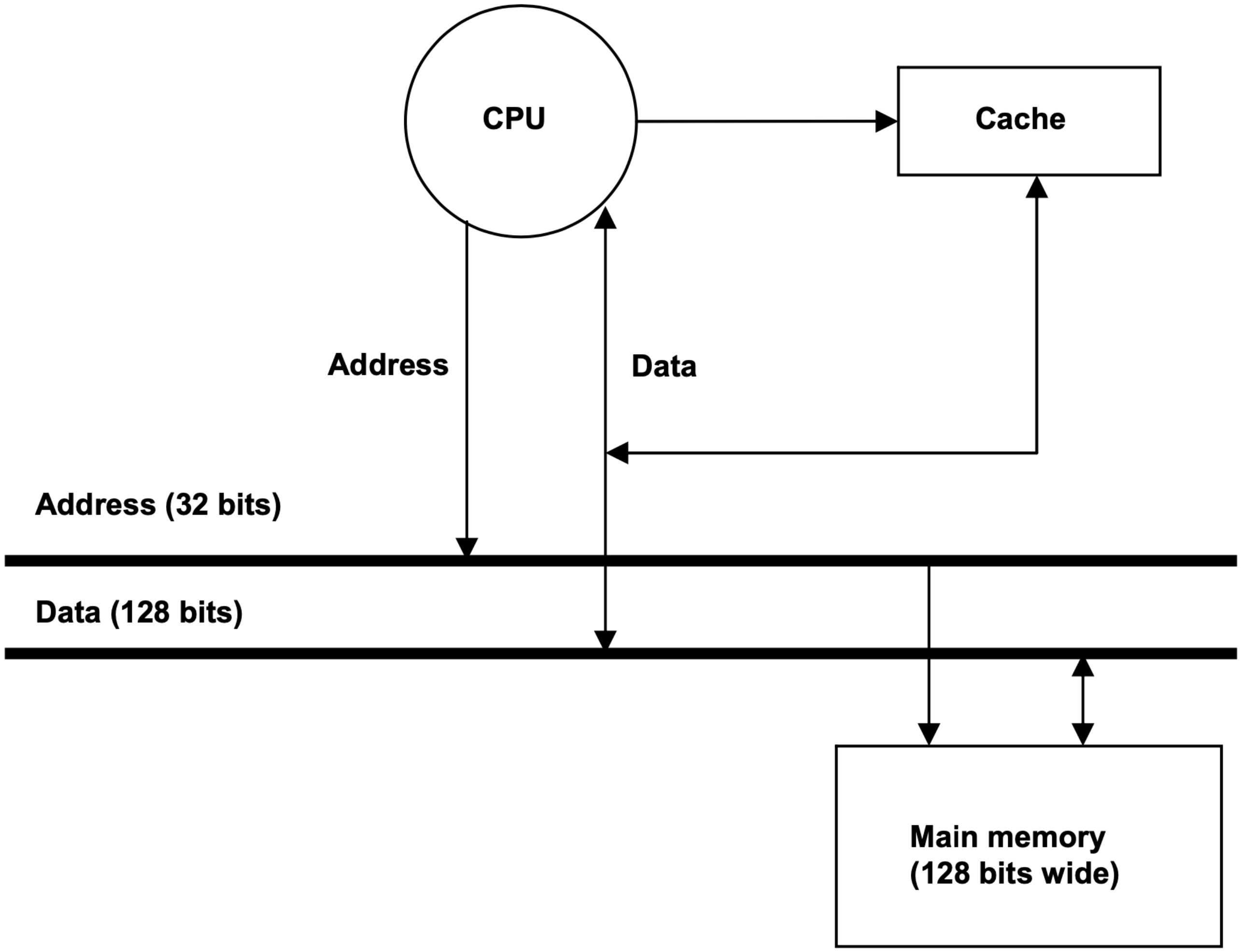


Figure 9.26: Main memory organization matching cache block size

Interleaved Memory

- Idea is to have multiple banks of memory
- Each bank is responsible for providing a specific word of the cache block

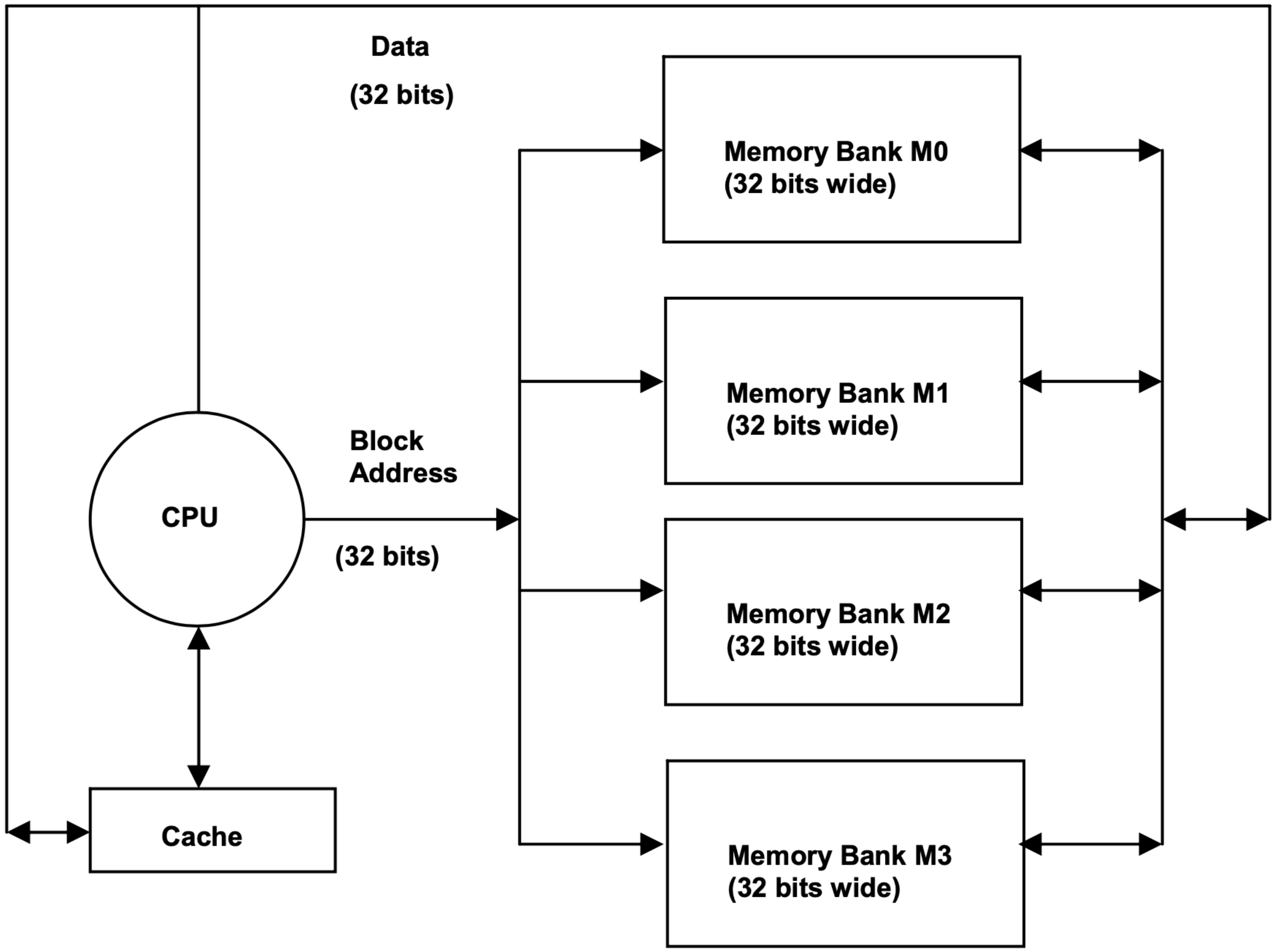


Figure 9.27: An interleaved main memory