CO101
Principle of Computer Organization

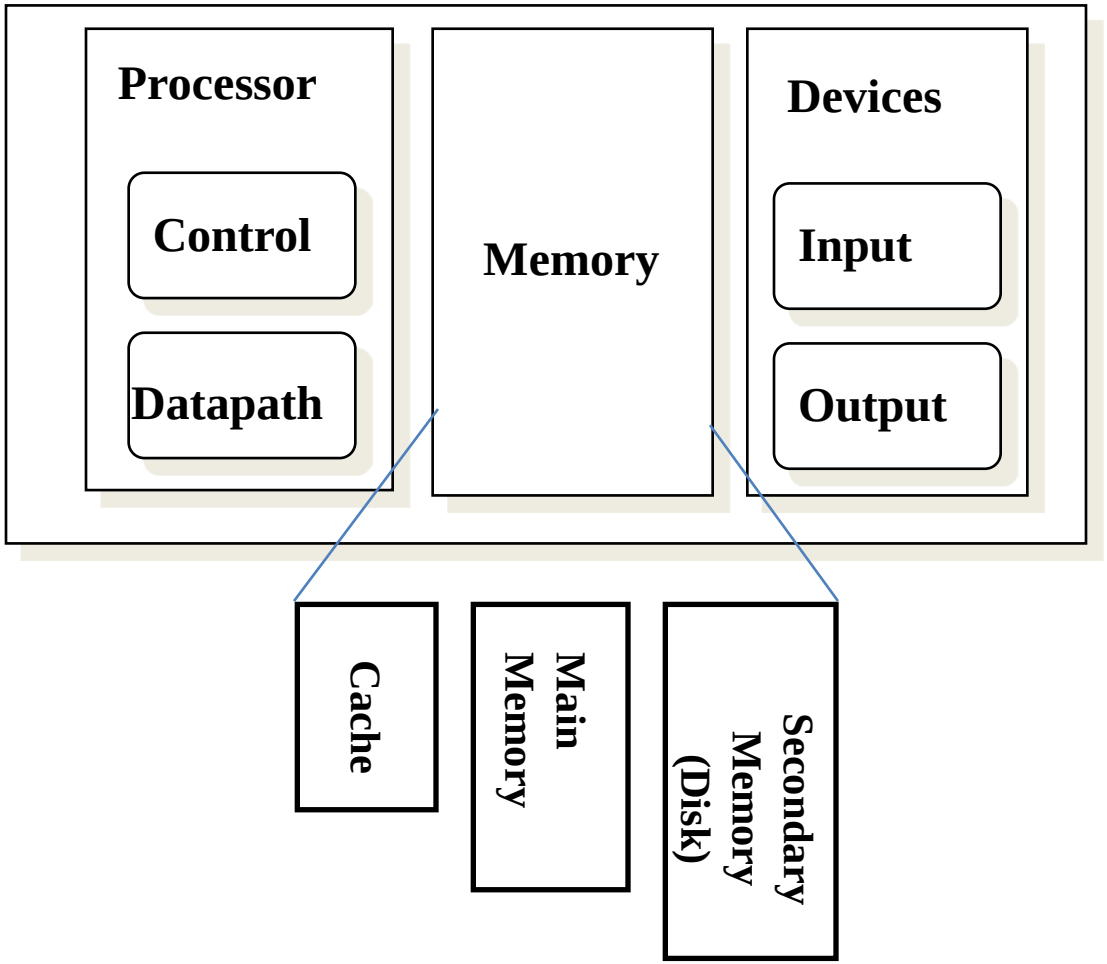
Lecture 07: Memory Architecture

Liang Yanyan

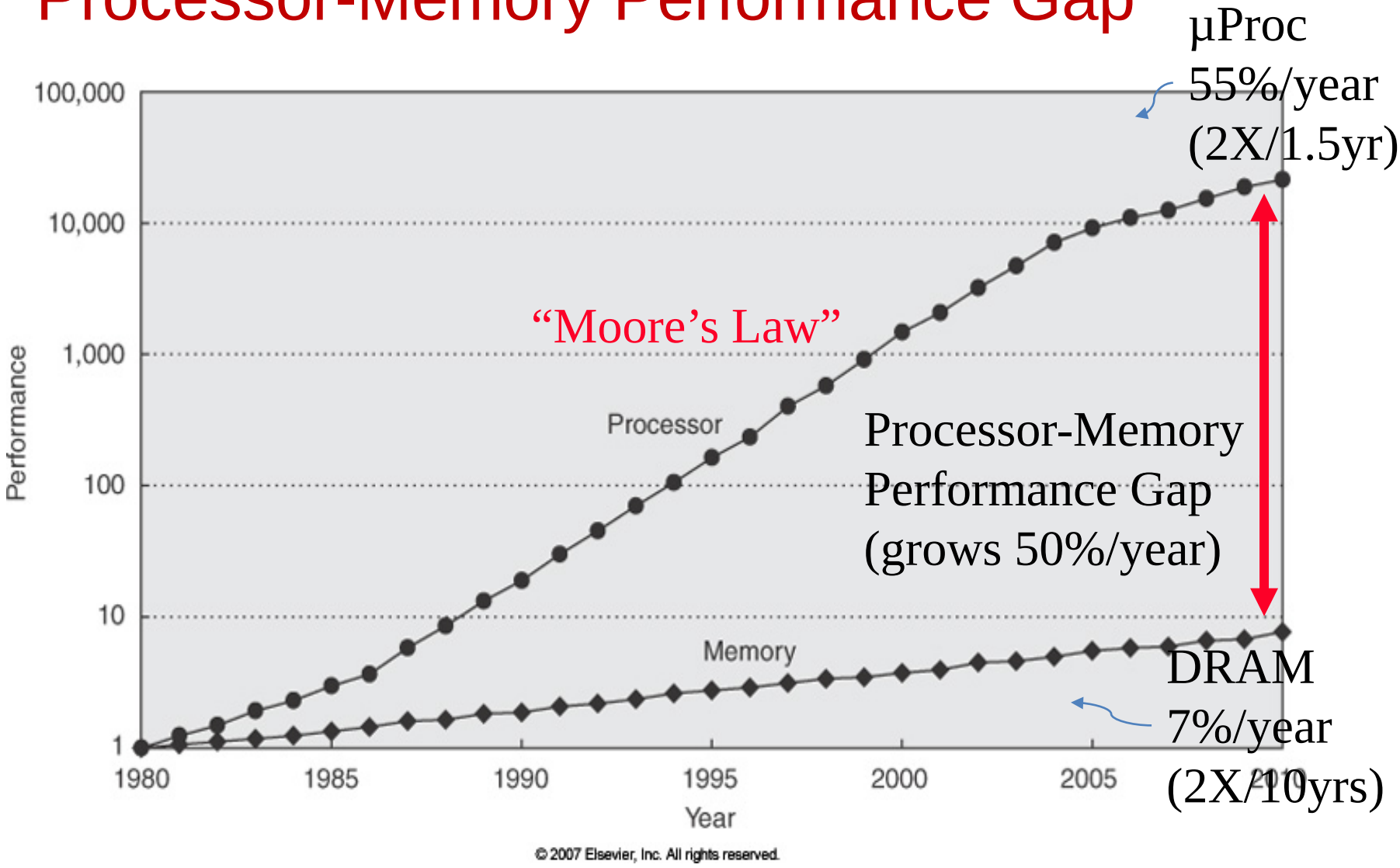
澳門科技大學
Macau of University of Science and Technology

Computer Organization

- CPU clock rate is much faster than memory.
- Slow memory can dramatically reduce the performance.



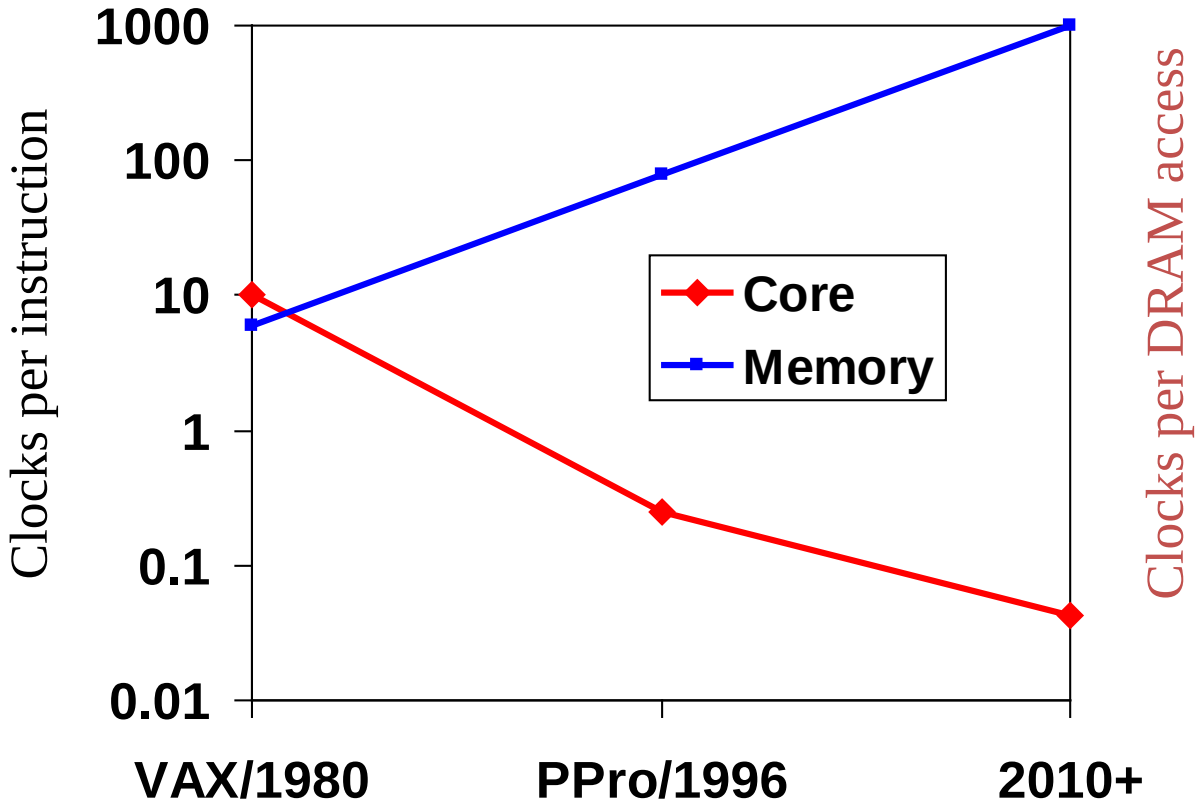
Processor-Memory Performance Gap



Slow memory can reduce the performance of computer systems.

The “Memory Wall”

- Processor vs DRAM speed disparity continues to grow



- Good memory hierarchy (cache) design is increasingly important to overall performance

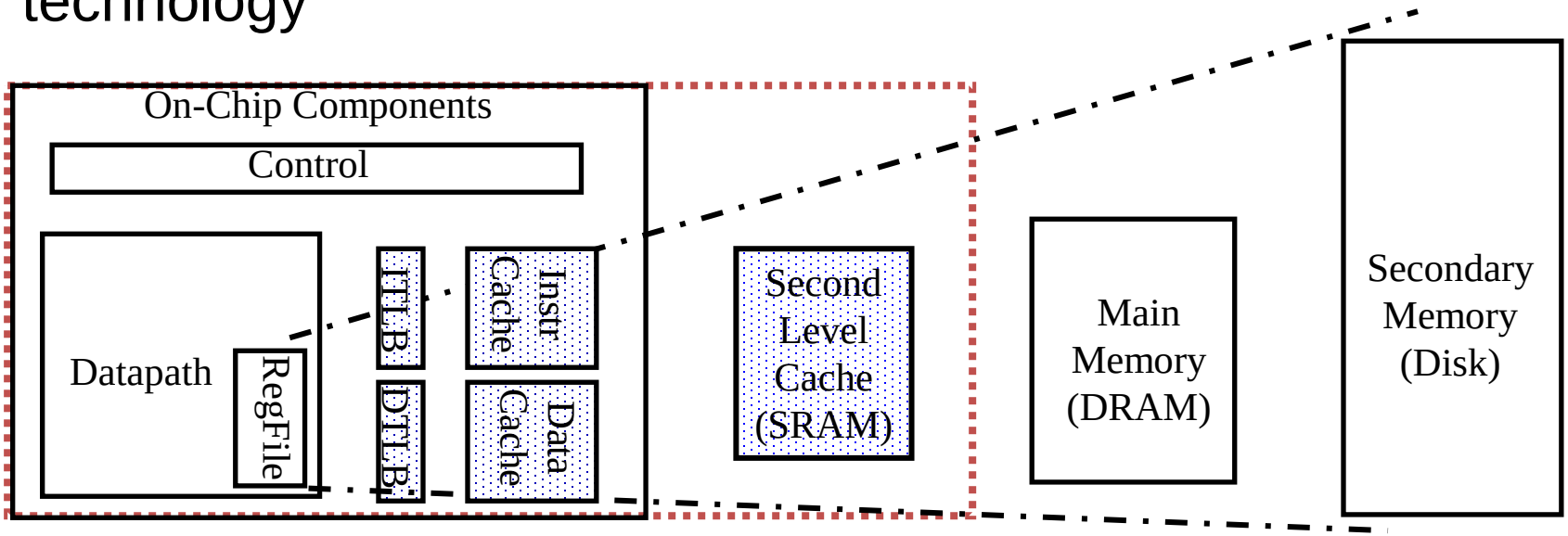
The Memory Hierarchy Goal

- Fact: Large memories are slow and fast memories are small

- How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?
  - With hierarchy

A Typical Memory Hierarchy

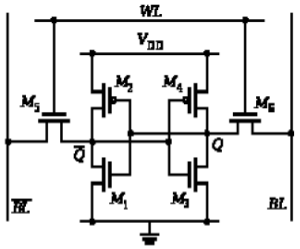
- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology

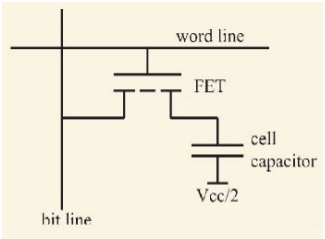


Speed (%cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

Memory Hierarchy Technologies

- Caches use *SRAM* for speed and technology compatibility
  - Fast (typical access times of 0.5 to 2.5 nsec)
  - Low density (6 transistor per bit), higher power, expensive ($2000 to $5000 per GB in 2008)
  - Static: content will last “forever” (as long as power is left on)
- Main memory uses *DRAM* for size (density)
  - Slower (typical access times of 50 to 70 nsec)
  - High density (1 transistor per bit), lower power, cheaper ($20 to $75 per GB in 2008)
  - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
    - consumes1% to 2% of the active cycles of the DRAM





SRAM: 6 transistors DRAM: 1 transistor

The Memory Hierarchy: Why Does it Work?

- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon.
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon.

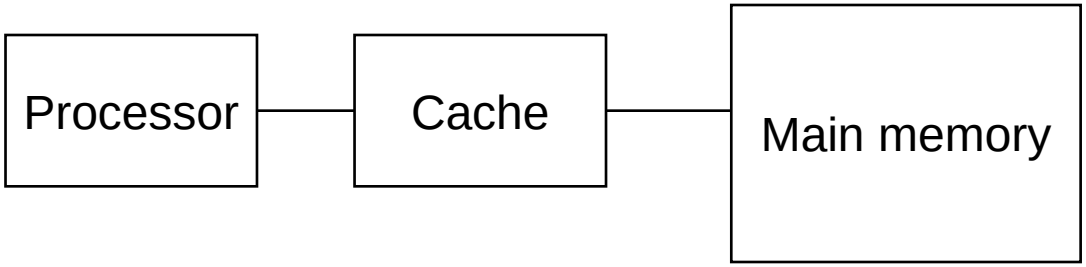
```
for (i=0; i <, 100; i++)
{
    a[i] = b[i] + c[i];
}
```

```
a = b + c;
d = c;
e = b + a;
```

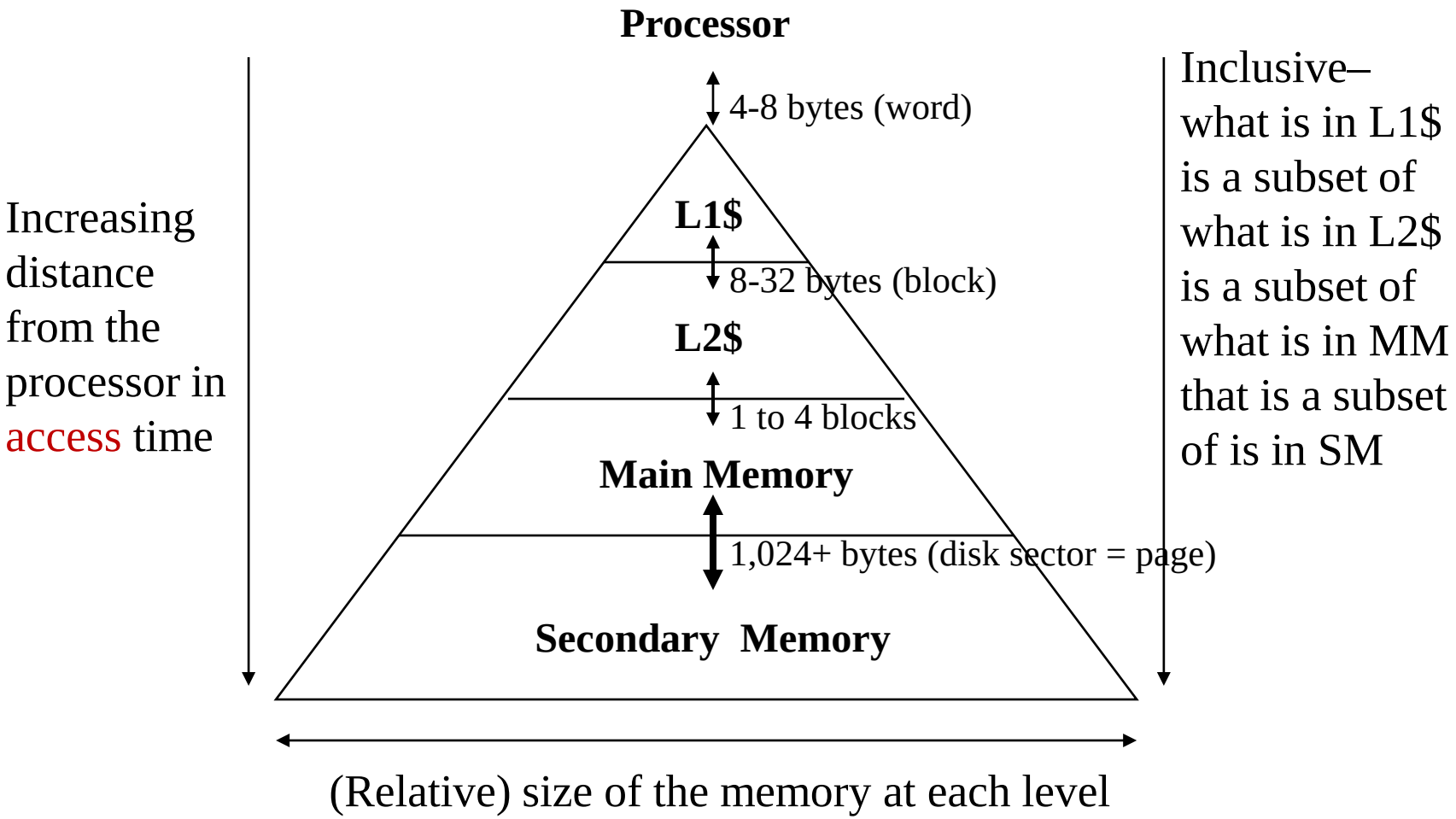
- How can we take advantage of the principles of locality to improve the performance of computer systems? At the same time:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

Make use of locality: add a cache

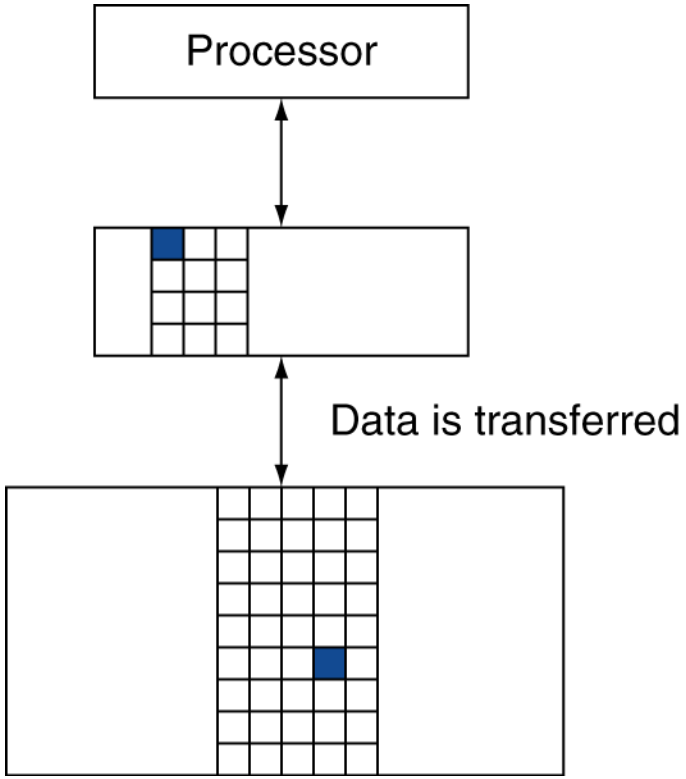
- Temporal locality
  - The first time CPU read a data *D* from main memory, *D* is also loaded into cache.
  - The next time CPU requests *D*, *D* is loaded from cache instead of main memory → reduce read latency.
- Spatial locality
  - Assume an array *D[10]*, the first time CPU reads *D[0]*, it is likely that the CPU will read subsequent data later, e.g. *D[1], D[2]*, …
  - The first time CPU read a data *D[0]* from main memory, data close to *D* are also loaded into cache, e.g. address *D+4*, *D+8*, etc.
  - The next time CPU requests data close to *D*, e.g. data at address *D+4*, it will be loaded from cache instead of main memory → reduce read latency.



Characteristics of the Memory Hierarchy



Memory Hierarchy Levels



- Block (or line): the minimum unit of information that is present (or not) in a cache.
  - May be multiple words of copying.
- If accessed data is present in upper level
  - A cache hit occurs if the cache contains the data that the processor is looking for. Hits are good, because the cache can return the data to the processor much faster than main memory.
- If accessed data is absent
  - A cache miss occurs if the cache does not contain the requested data. This is bad, since the processor has to load data form the slow main memory.

Some Definitions

- There are two basic measurements of cache performance.
  - The hit rate is the percentage of memory access that are handled by the cache.
    - Hit Time: Time to access that level which consists of

      Time to access the block + Time to determine hit/miss
  - The miss rate (1 - hit rate) is the percentage of access that must be handled by the slow main memory.
    - Miss Penalty: Time to replace a block in that level with the corresponding block from a lower level which consists of

      Time to access the block in the lower level + Time to transmit that block to the level that experienced the miss + Time to insert the block in that level + Time to pass the block to the requestor
  - Hit Time << Miss Penalty
- Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and results in higher performance.

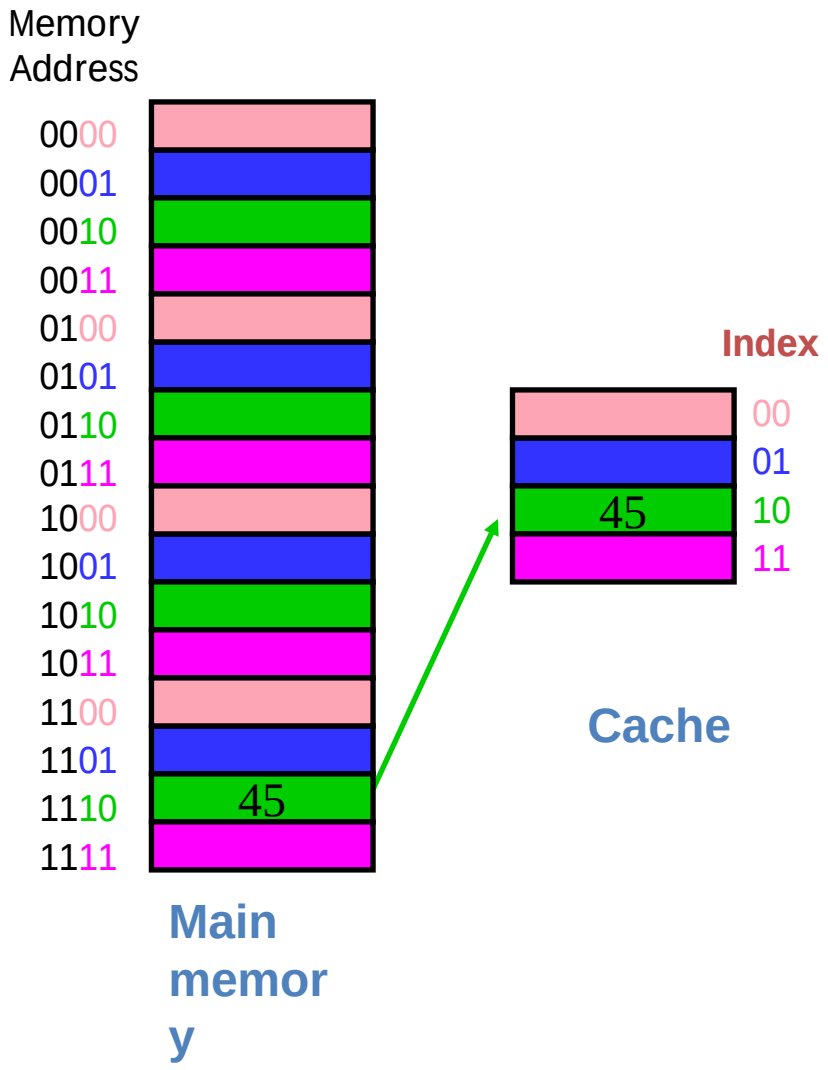
Important Issues

- Where can a data be placed in cache when we copy the data from main memory to cache?
  - As we can move data from memory to cache to make use of temporal and spatial locality.
- How to locate a data in cache?
  - This is related to the first issue.
- If cache is full, how to replace an existing data?

Where can a data be placed in cache?

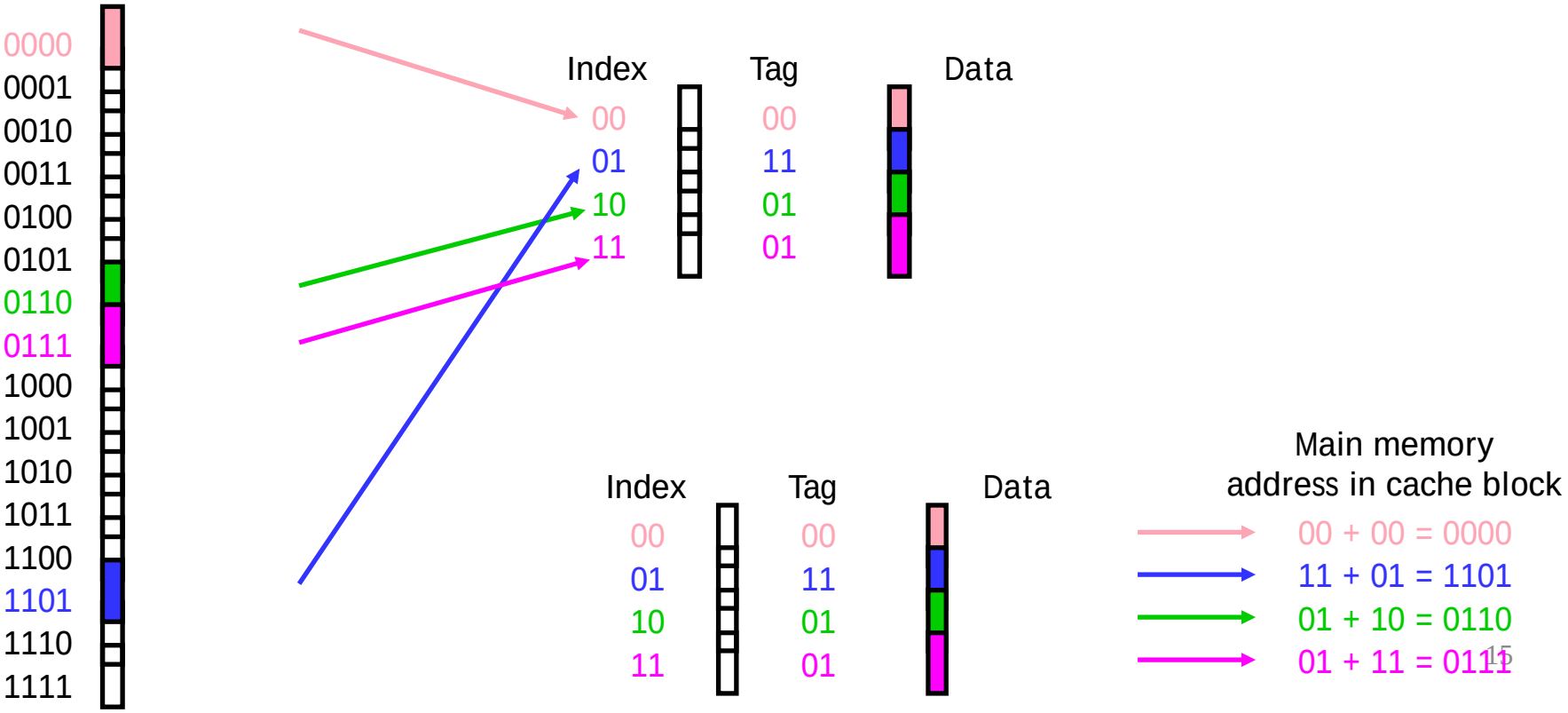
- Each memory location is mapped to a location in the cache using the least significant k bits of the address → direct mapped cache
  - Least significant 2 bits are used in this example.
  - E.g. data in memory address 1110 will be stored in cache address (index) 10, data in memory address 0001 will be stored in cache address 01.
- Given a memory address *A*, the corresponding cache index *C* is:

  $C = A \bmod 2^K$
- Multiple memory locations can be mapped to one cache location.



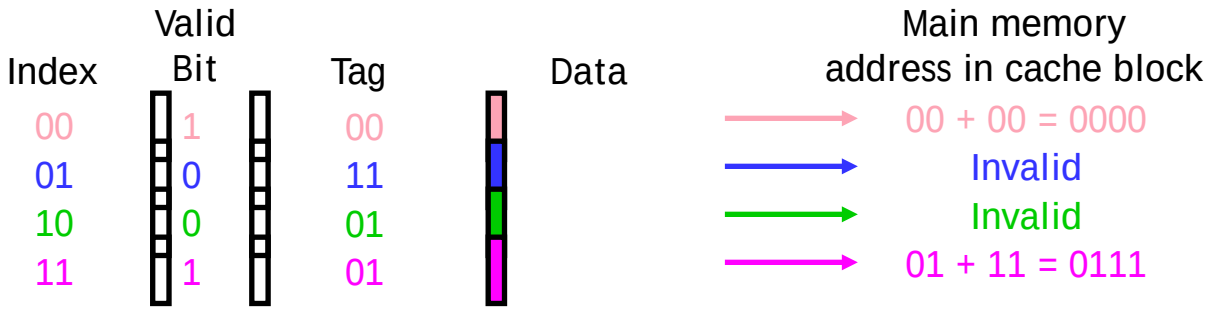
Load data from cache

- Since multiple memory locations can be mapped to one cache location
  - If we want to read data from memory address 0010, how can we know that the cache index 10 is storing data for memory 0010, 0110, 1010, or 1110?
- Each cache entry is associated with a tag to specify the upper memory address of the memory data that the current cache location is storing.



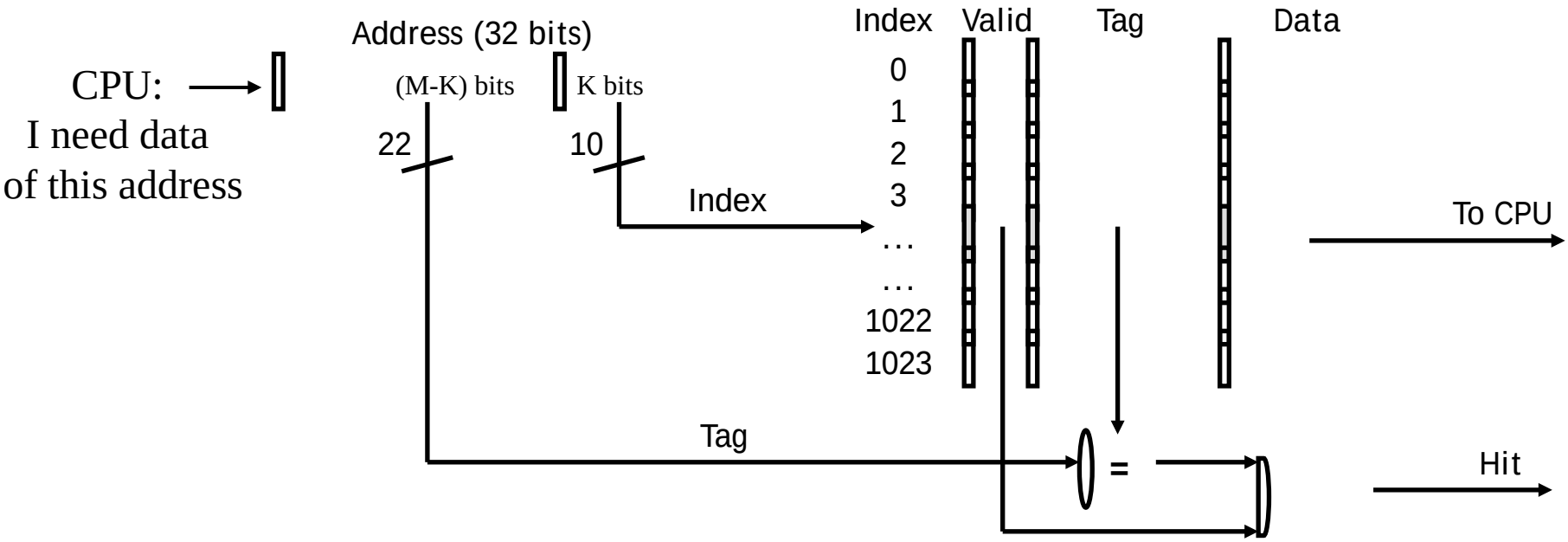
Is data in cache?

- Even the Tag and Index match when reading data from cache, how can we know the data is update (valid) or not?
  - Startup?
  - CPU loads another program to execute?
- Add a valid bit
  - 0: Data is invalid for reading.
  - 1: data is valid and ready for reading.



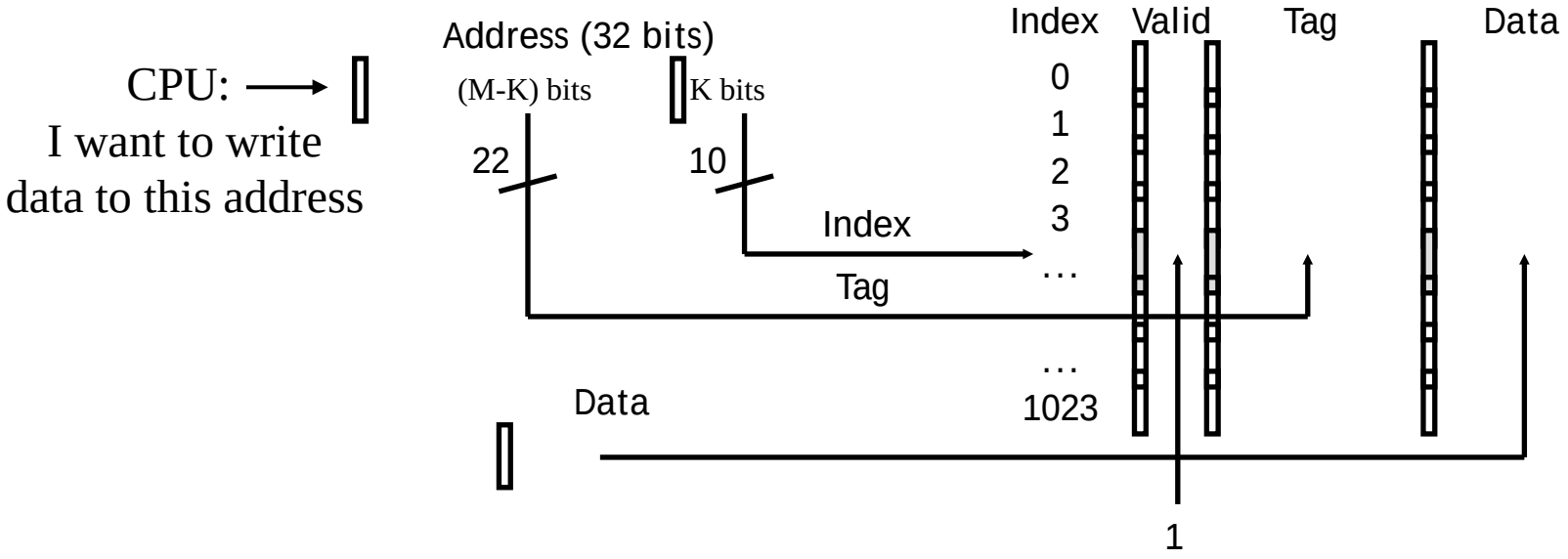
Processor read data from memory

- Assume memory address is M-bit width
- Search the cache
  - Use the lower K bits of the memory address as index to locate the corresponding cache entry.
  - Check if the tag equals to the upper (M-K) bits of the memory address.
  - Check if the valid bit is 1.
  - If all these are true → cache hit, return data from cache to processor.



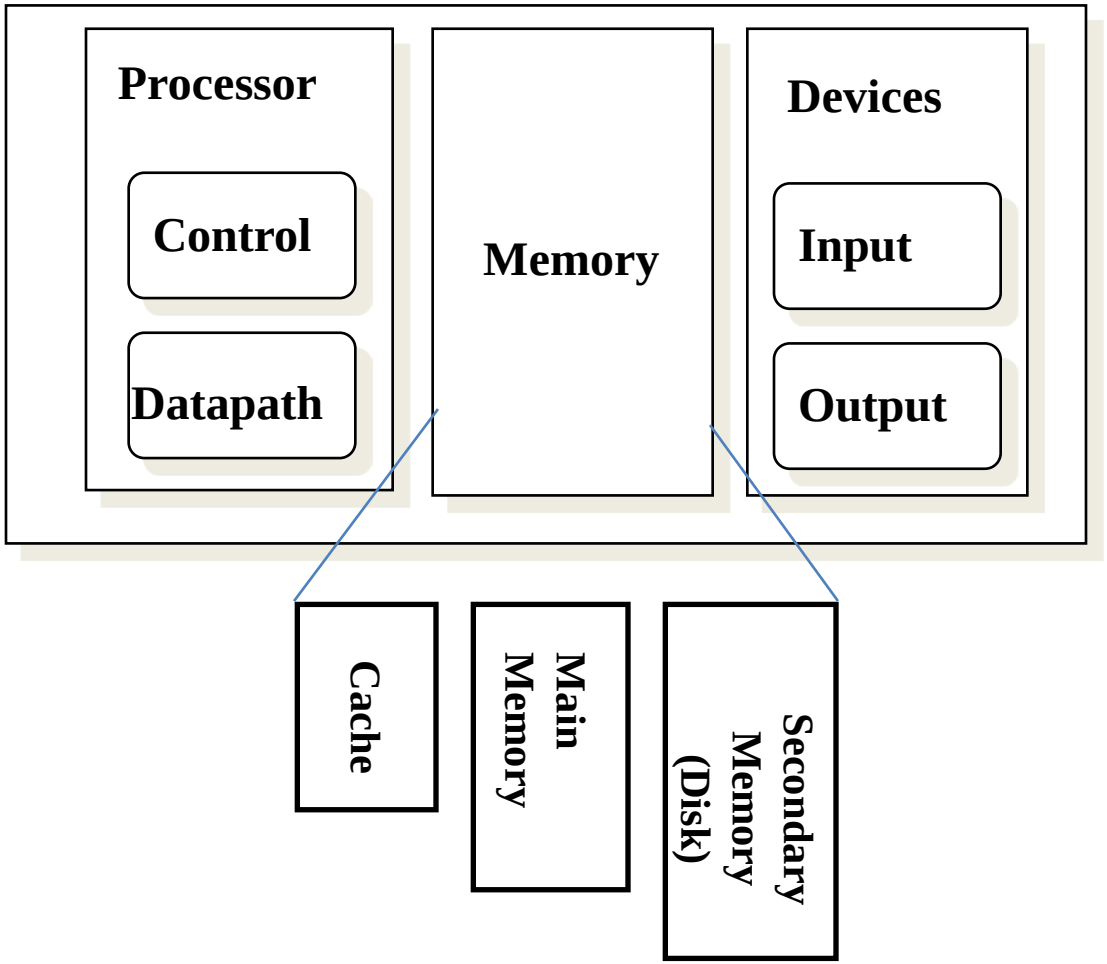
Write data to cache (from processor to memory)

- To write a data to cache
  - Use the lower K bits of the memory address as index to locate the corresponding cache entry.
  - Store the upper (M-K) bits of the memory address into tag field.
  - Store data into cache's data field.
  - Set the valid bit to 1.
  - Different write policies will be discussed later.



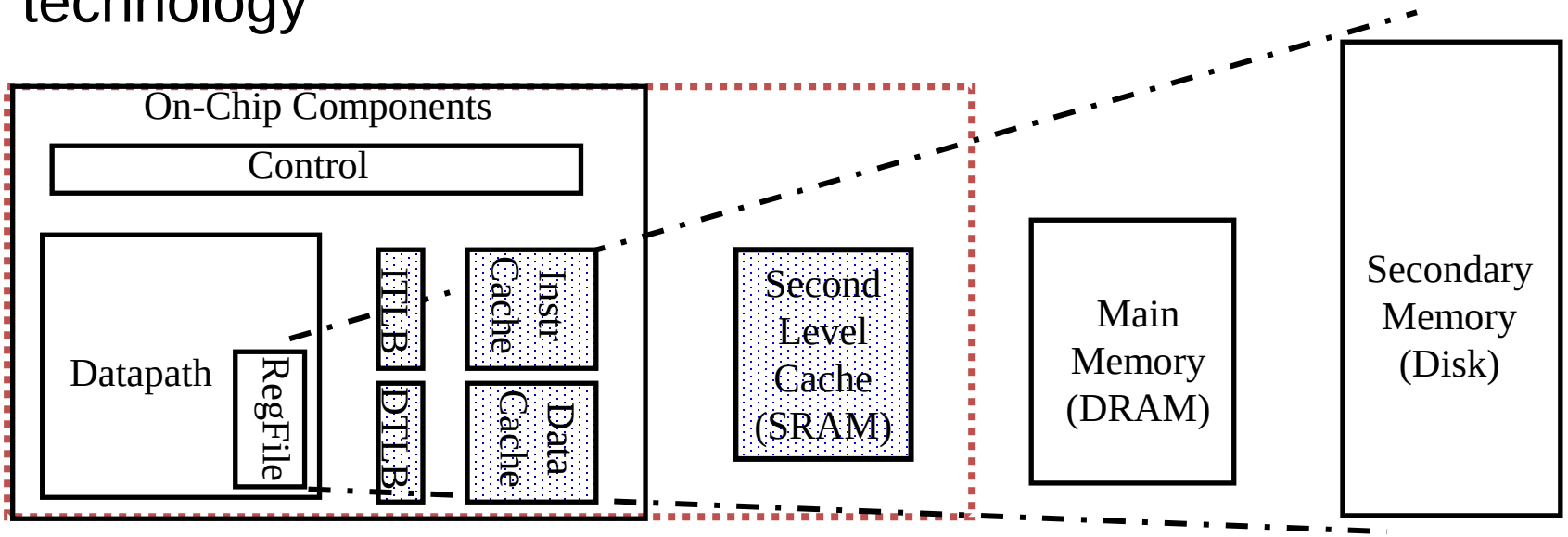
Computer Organization

- CPU clock rate is much faster than memory.
- Slow memory can dramatically reduce the performance.



A Typical Memory Hierarchy

- Take advantage of the principle of locality to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



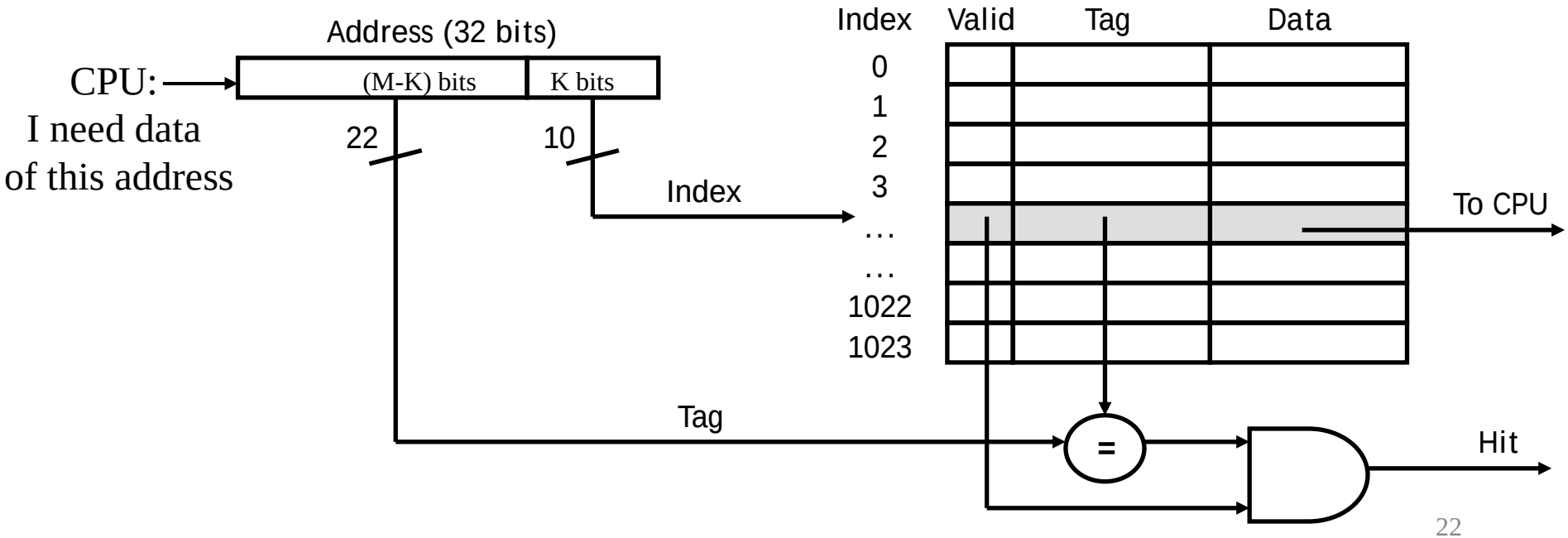
Speed (%cycles):	½'s	1's	10's	100's	10,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost:	highest				lowest

Important issues

- Where can a data be placed in cache when we copy the data from main memory to cache?
  - As we can move data from memory to cache to make use of temporal and spatial locality.
- How to locate a data in cache?
  - This is related to the first issue.
- If cache is full, how to replace an existing data?

Processor read data from memory

- Assume memory address is M-bit width
- Search the cache
  - Use the lower K bits of the memory address as index to locate the corresponding cache entry.
  - Check if the tag equals to the upper (M-K) bits of the memory address.
  - Check if the valid bit is 1.
  - If all these are true → cache hit, return data from cache to processor.



Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	**Y**	**10**	**Mem[10110]**
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
26	11 010	Miss	010

Index	V	Tag	Data
000	N		
001	N		
010	**Y**	**11**	**Mem[11010]**
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Hit	110
26	11 010	Hit	010

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem[11010]
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

Index	V	Tag	Data
000	**Y**	**10**	**Mem[10000]**
001	N		
010	Y	11	Mem[11010]
011	**Y**	**00**	**Mem[00011]**
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cache Example

Word addr	Binary addr	Hit/miss	Cache block
18	10 010	Miss	010

Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	**Y**	**10**	**Mem[10010]**
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

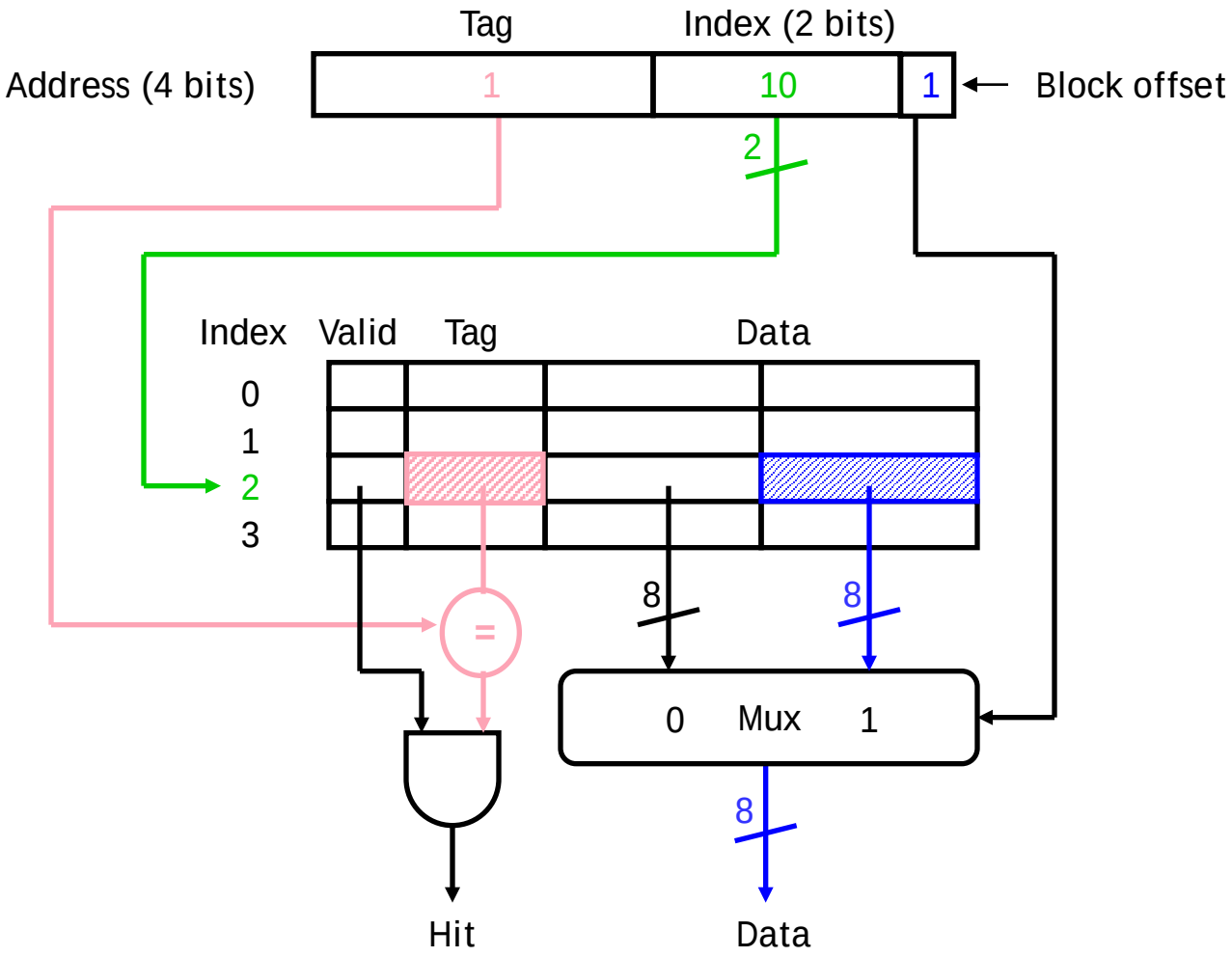
Cache Example

Word addr	Binary addr	Hit/miss	Cache block
16	10 000	Hit	000

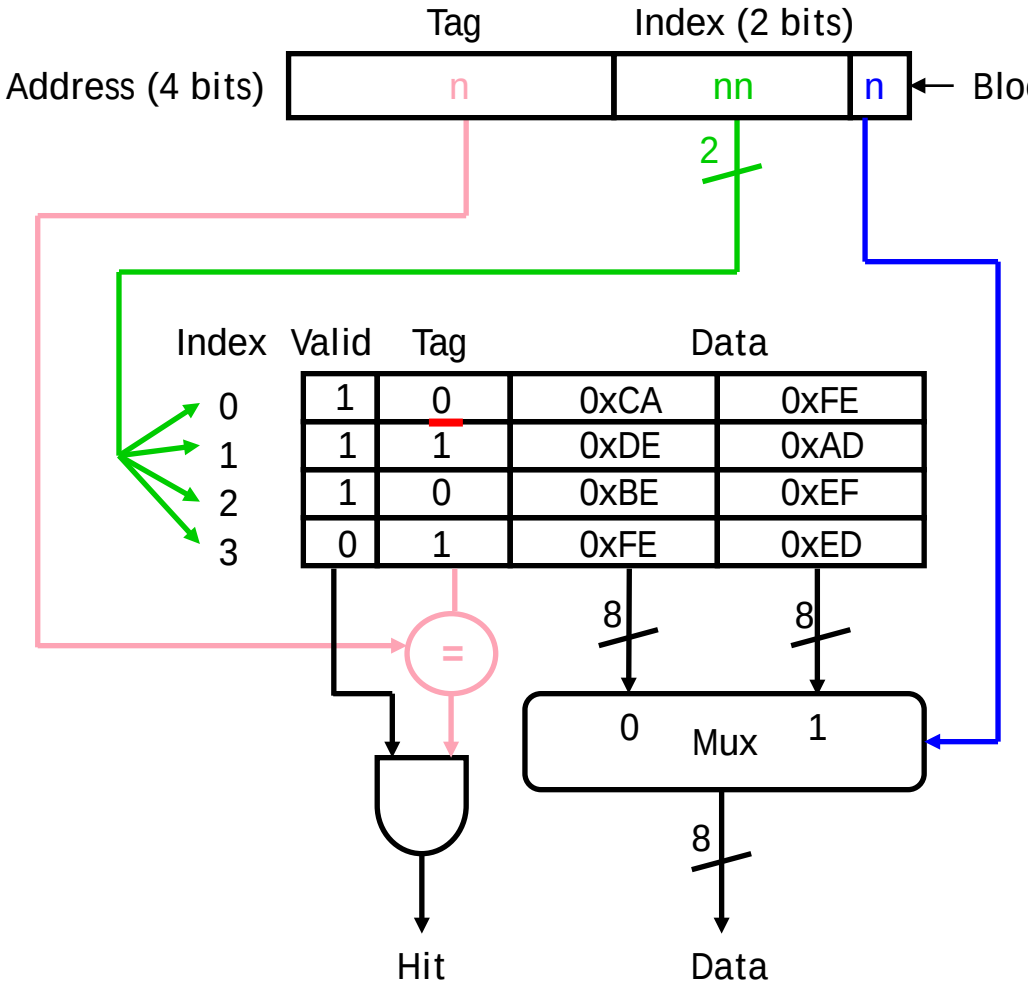
Index	V	Tag	Data
000	Y	10	Mem[10000]
001	N		
010	Y	10	Mem[10010]
011	Y	00	Mem[00011]
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Example:

- 4 bits memory address



Example:

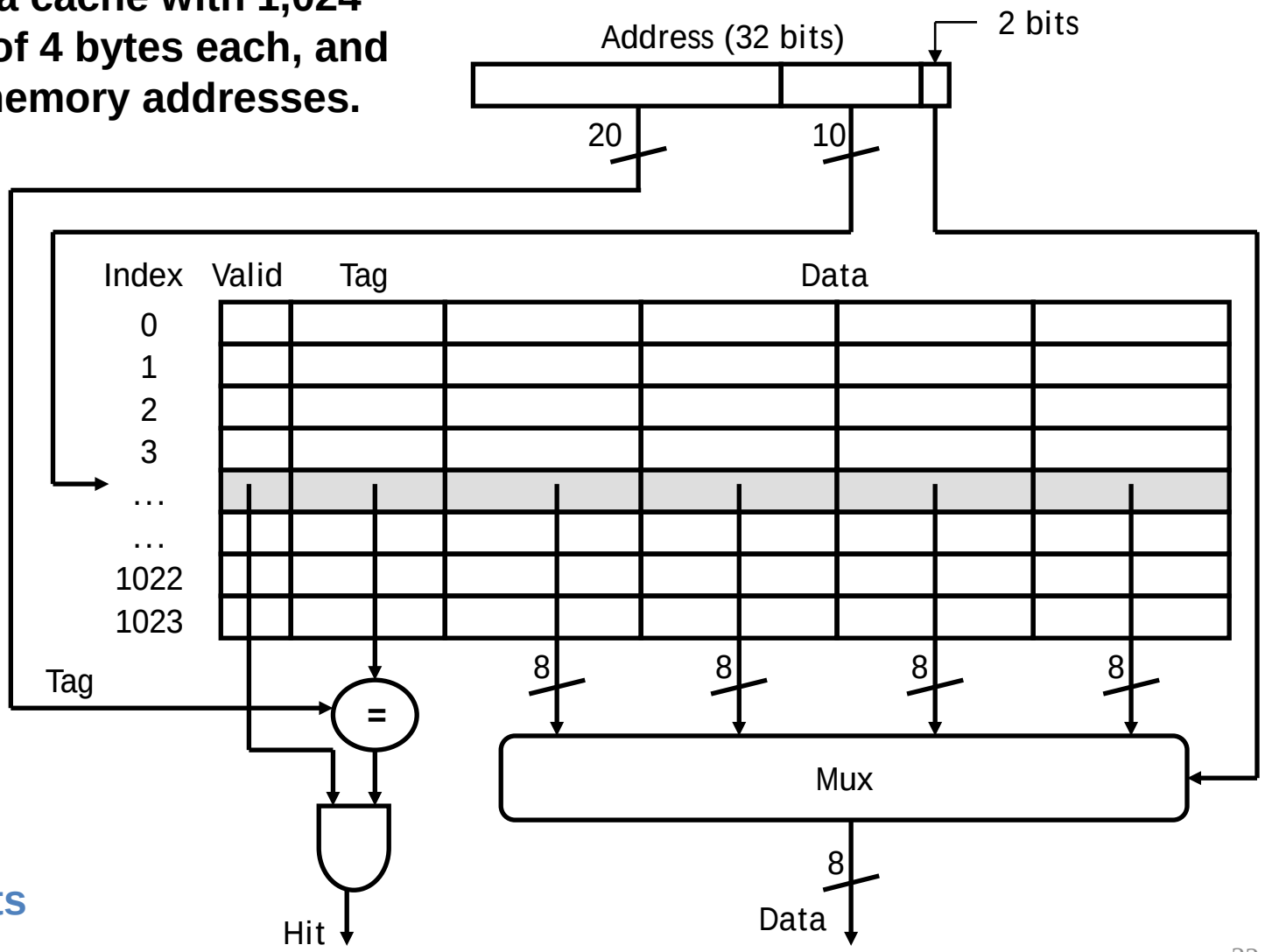


For the addresses below, what byte is read from the cache (or is there a miss)?

- **1010**
- **1110**
- **0001**
- **1101**

A larger case

- **Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses.**



Tag = 20 bits

Index = 10 bits

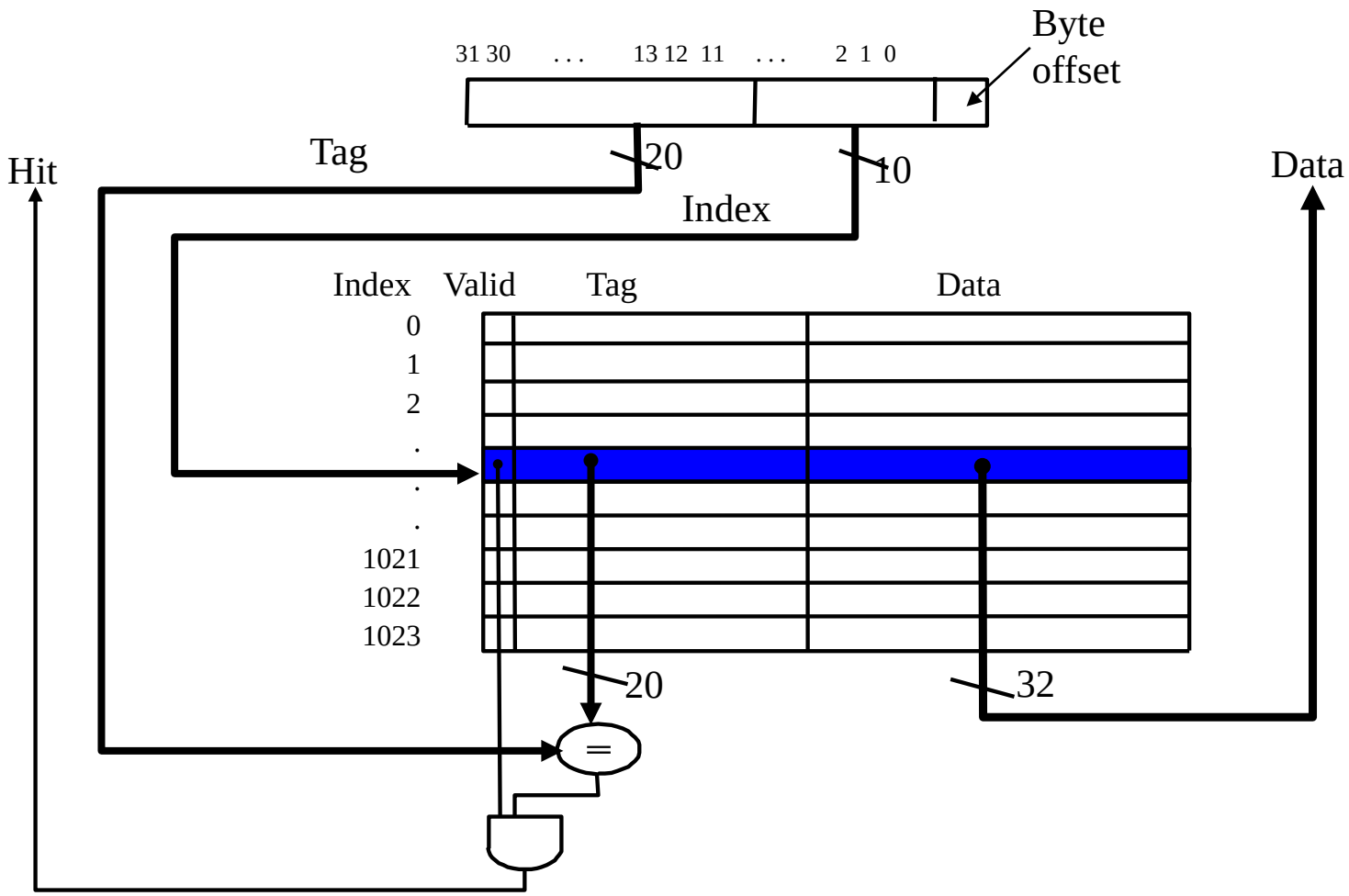
Block offset = 2 bits

Cache Field Sizes

- The number of bits in a cache includes both the storage for data and for the tags.
  - 32-bit byte address
  - For a direct mapped cache with 2^n blocks, *n* bits are used for the index
  - For a block size of 2^m words (2^{m+2} bytes), *m* bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the tag field?
  - 32 – (n + m +2)
- The total number of bits in a direct-mapped cache is then
  - 2^n x (block size + tag field size + valid field size)

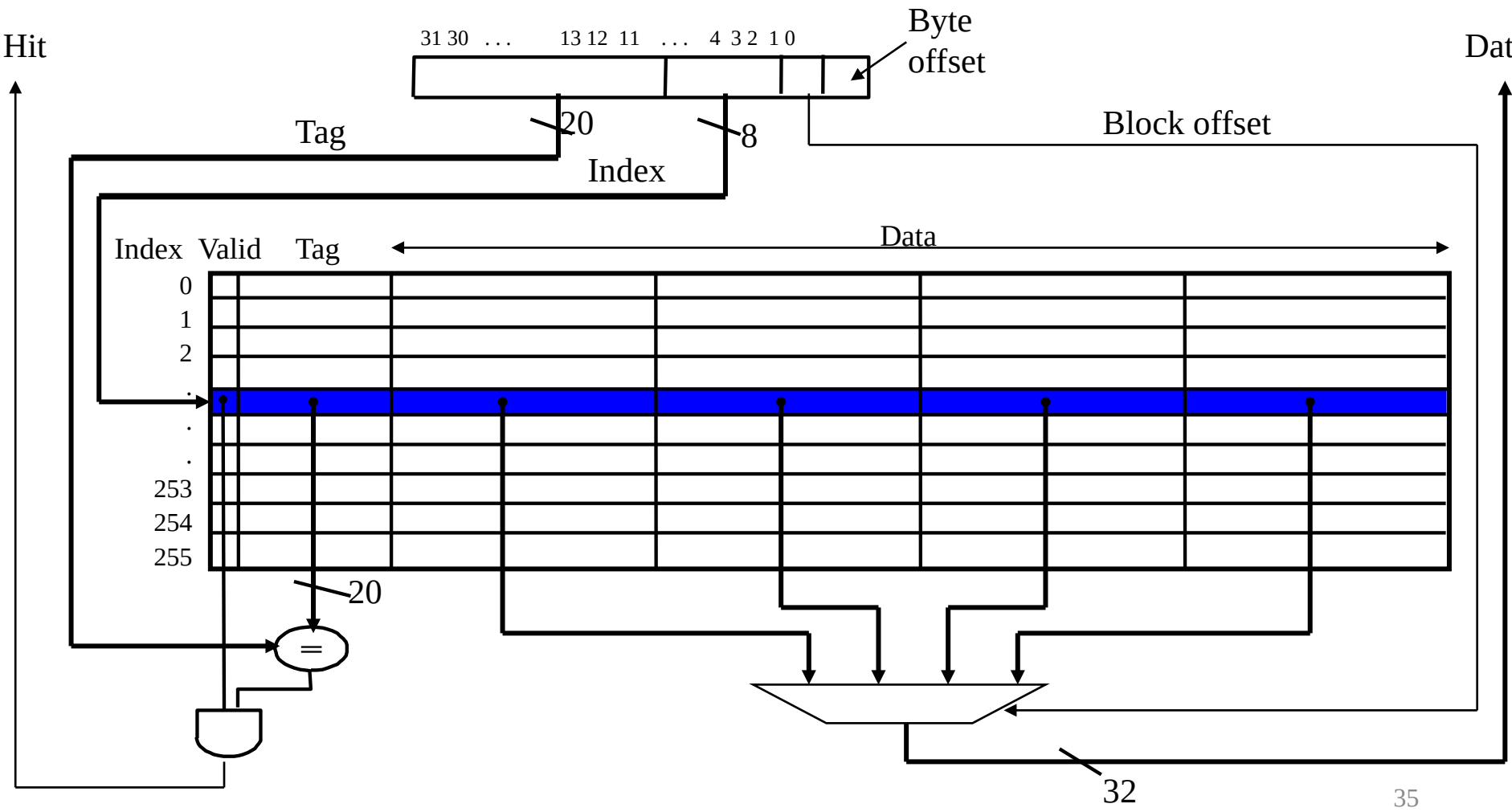
MIPS Direct Mapped Cache Example

- One word blocks, cache size = 1K words (or 4KB)

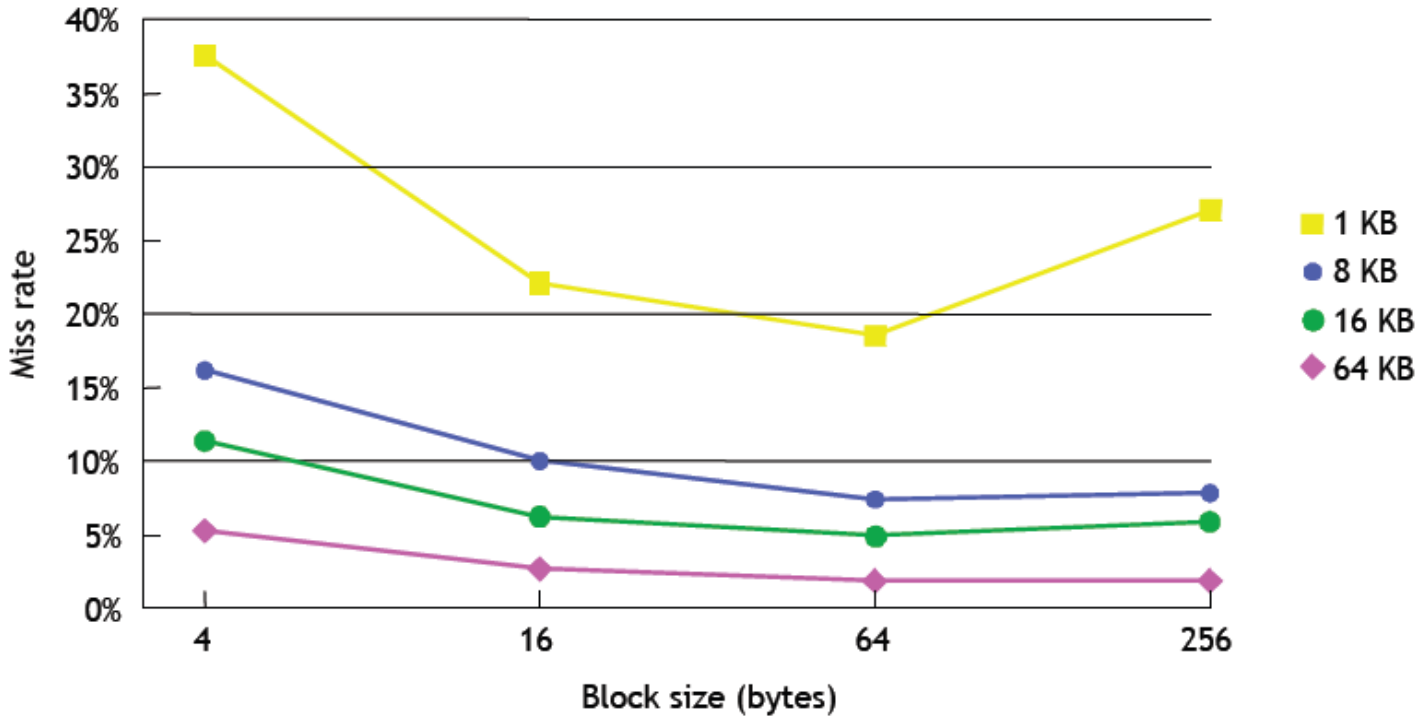


Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



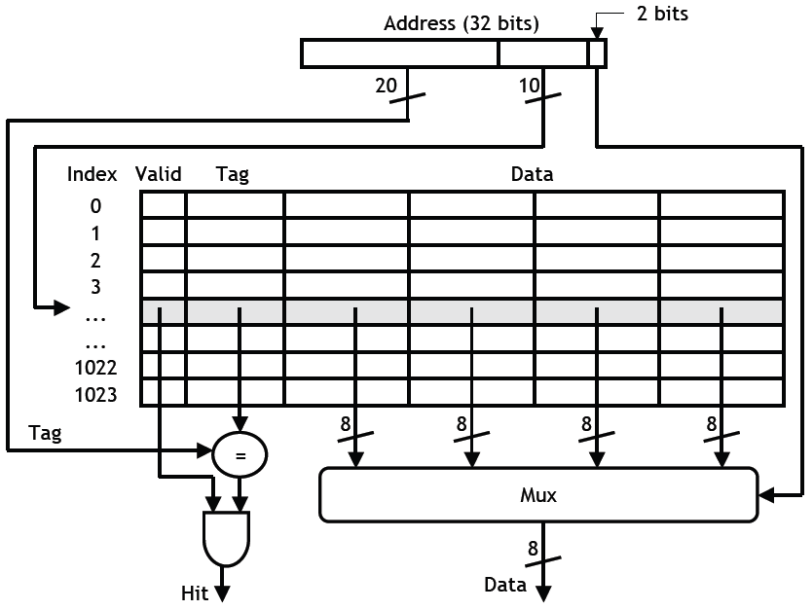
Miss Rate vs Block Size vs Cache Size



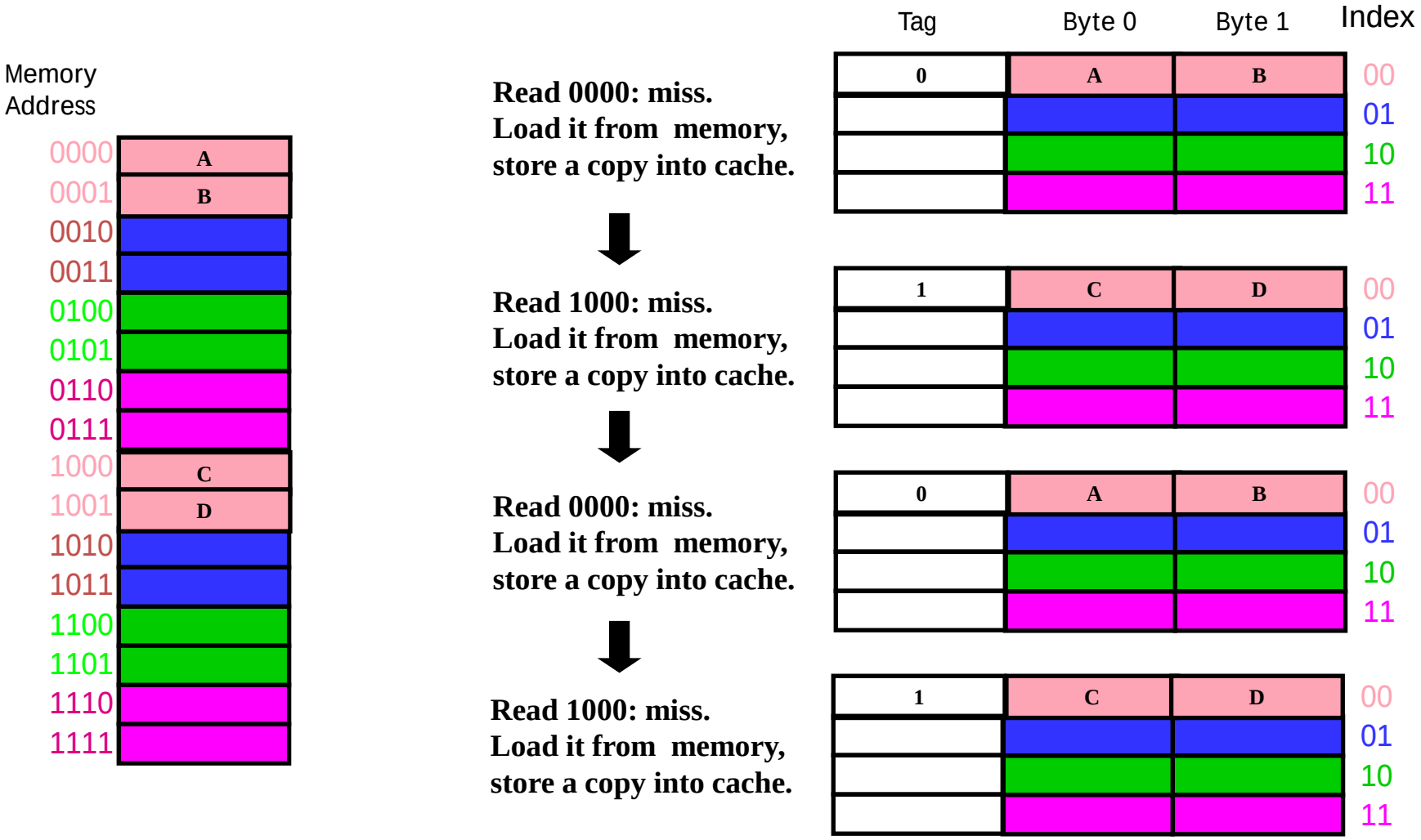
- Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing capacity misses)

Disadvantages of direct-mapped cache

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs to exactly one block.
- However, this isn't really flexible. Consider a program keep reading two addresses A1, A2, A1, A2, A1, A2 ,… where A1 and A2 have the same cache index and block offset.
  What will happen?



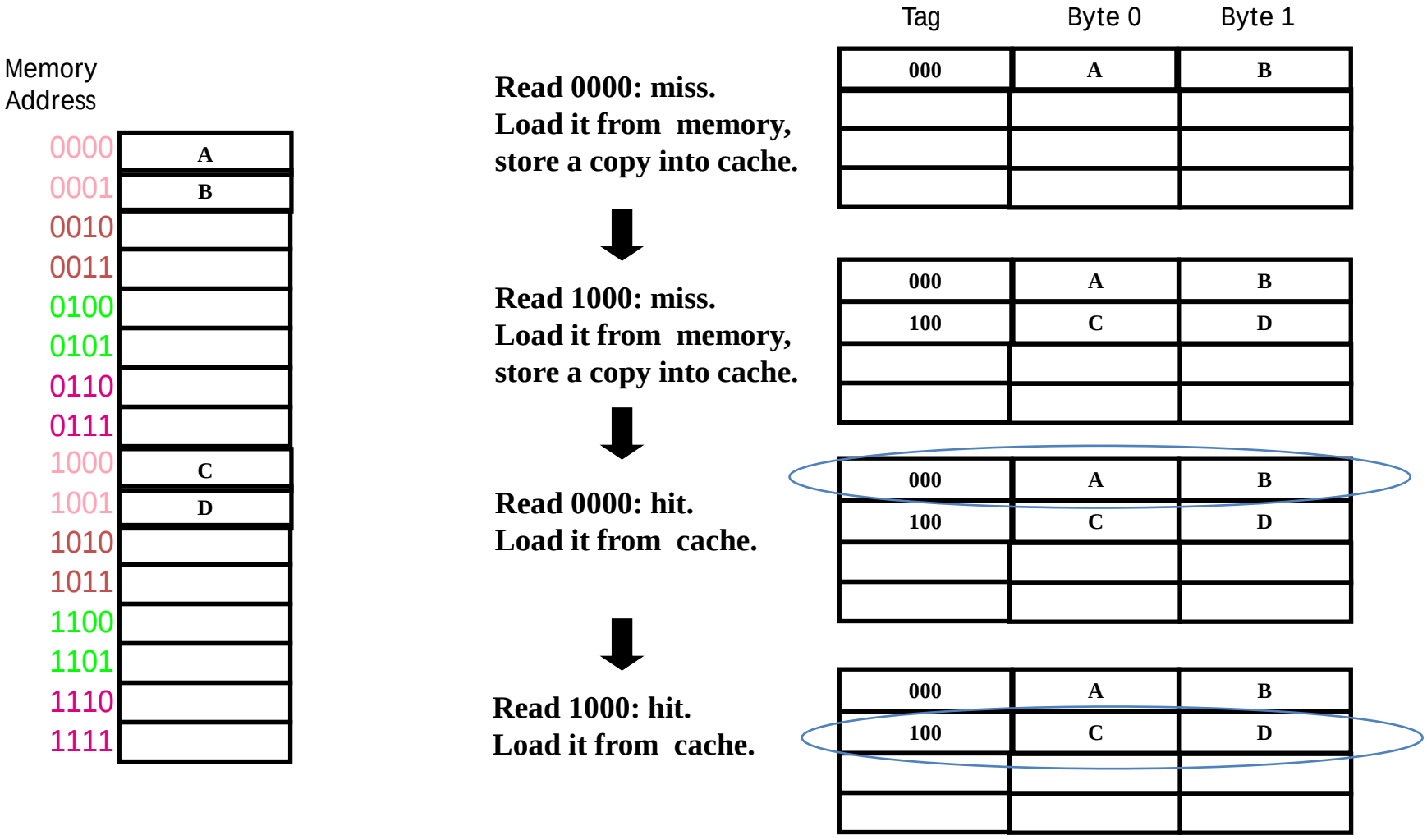
Directed mapped cache: keep reading 0000, 1000



A fully associative cache

- A fully associative cache permits data to be stored in *any* cache block, instead of enforcing each memory address into one particular block.
  - When data is stored to cache, it can be placed in *any* unused block of the cache. There is no cache index field to locate the cache entry.
  - In this way we can reduce the conflict between two or more memory addresses which map to a single cache block.
- In the previous example, we might put memory address A1 in cache block 1, and address A2 in block 2. Then subsequent accesses to A1 and A2 would all be hits instead of misses.
- If all the blocks are already in use, it's usually best to replace the least recently used one which is least used in recent time. Another replacement strategy is most recently used which is replacing the one that is most used in recent time.

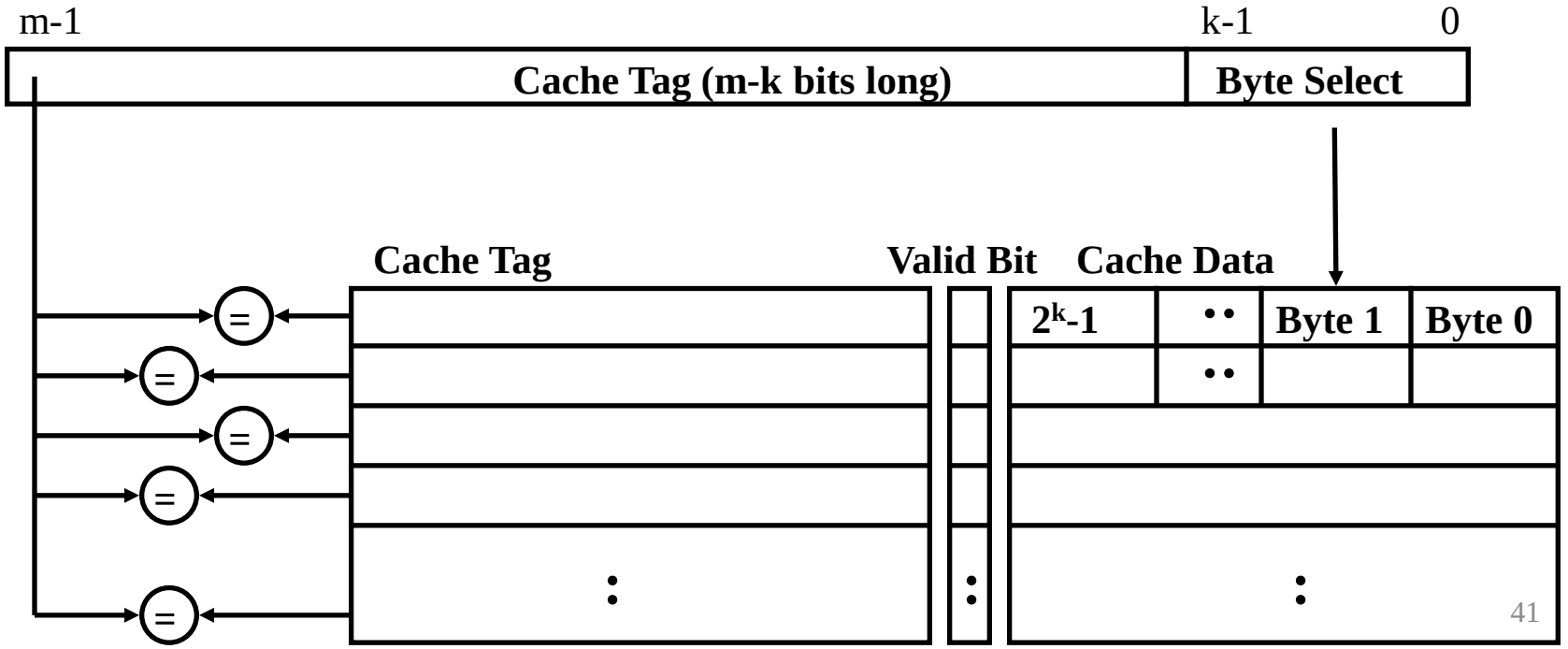
Fully associative cache: keep reading 0000, 1000



Data can be stored in arbitrary location in the cache, thus no cache index field.

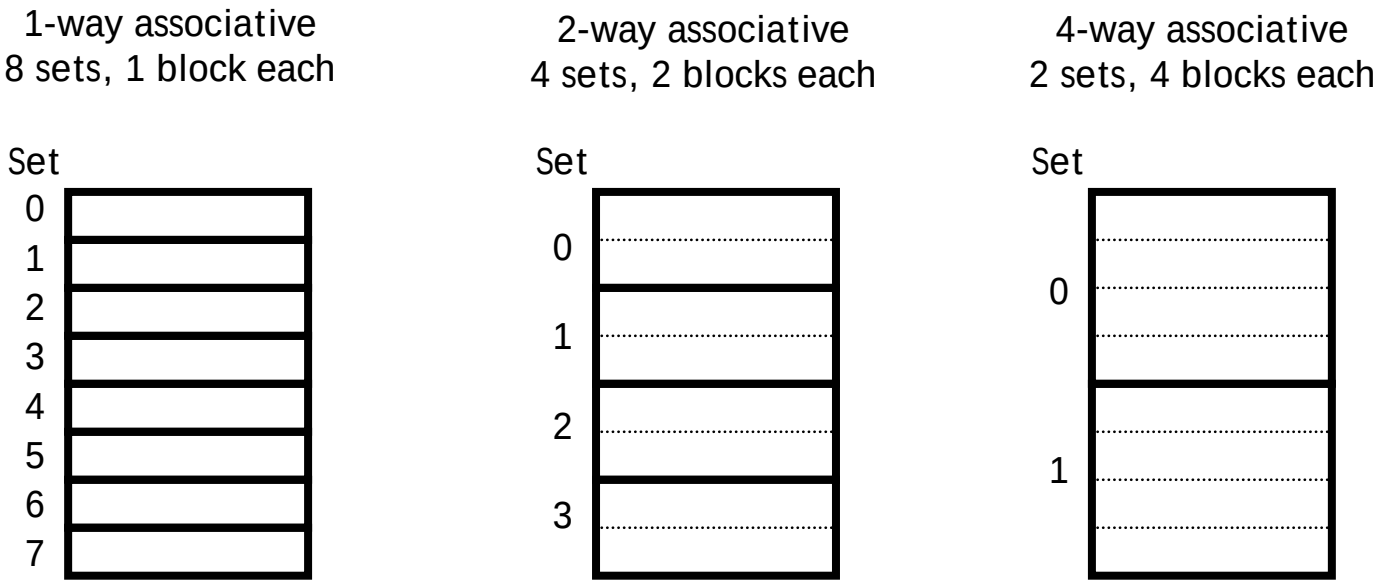
Disadvantages of fully associative cache

- Since data can be put into arbitrary location, so there is no cache index field.
  - Assume the address is m bits, each block contains 2^k bytes of data.
  - The tag is (m-k) bits long.
  - To read a data, we must scan all blocks in the cache and check if the tag match the address we are going to read. → Require too many comparators, huge hardware size.



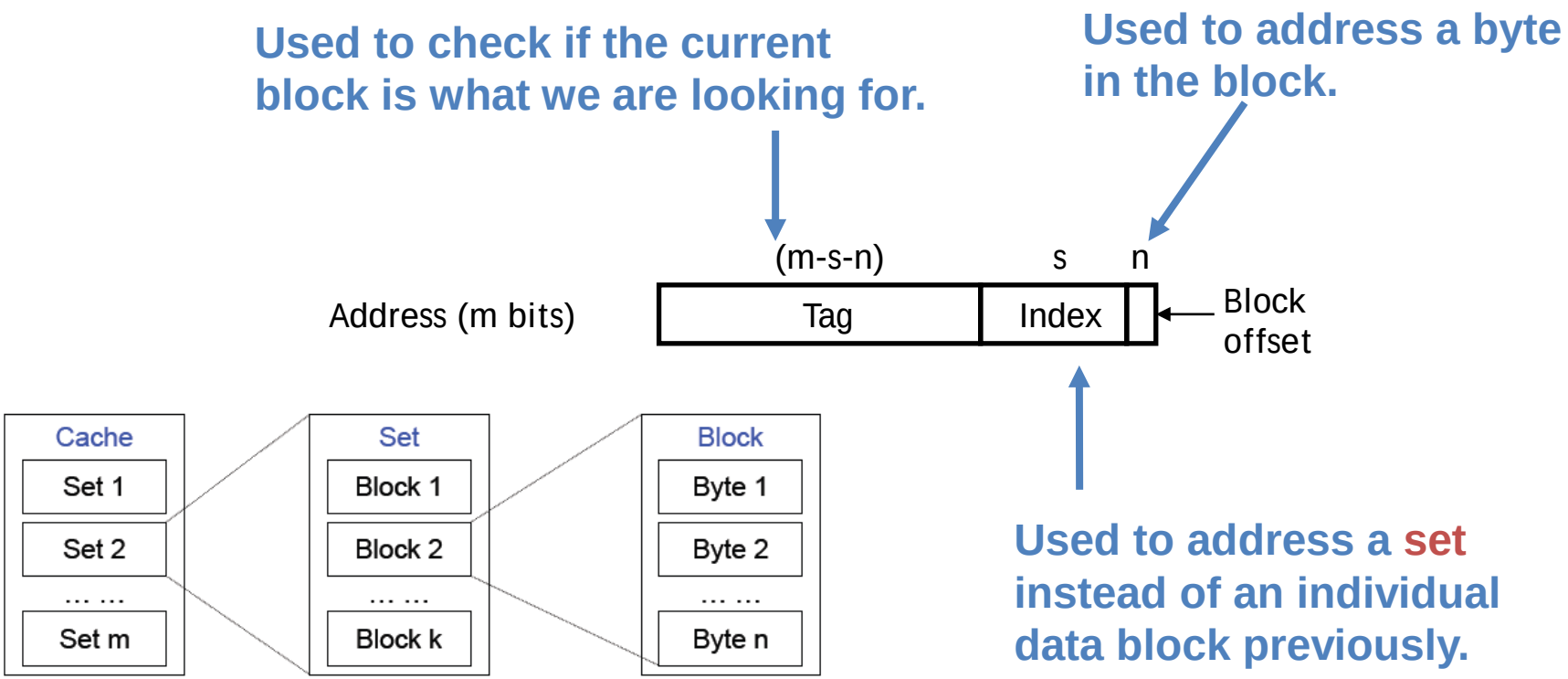
Set-associative cache

- Set-associative cache provides an intermediate solution.
  - Cache blocks are divided into *groups*, each group is called a *set*.
  - Each memory address is mapped to exactly one set in the cache, but data can be stored in arbitrary block within the set.
- If each set has N blocks, the cache is called a N-way associative cache.
- Here are several possible organizations of an eight-block cache.



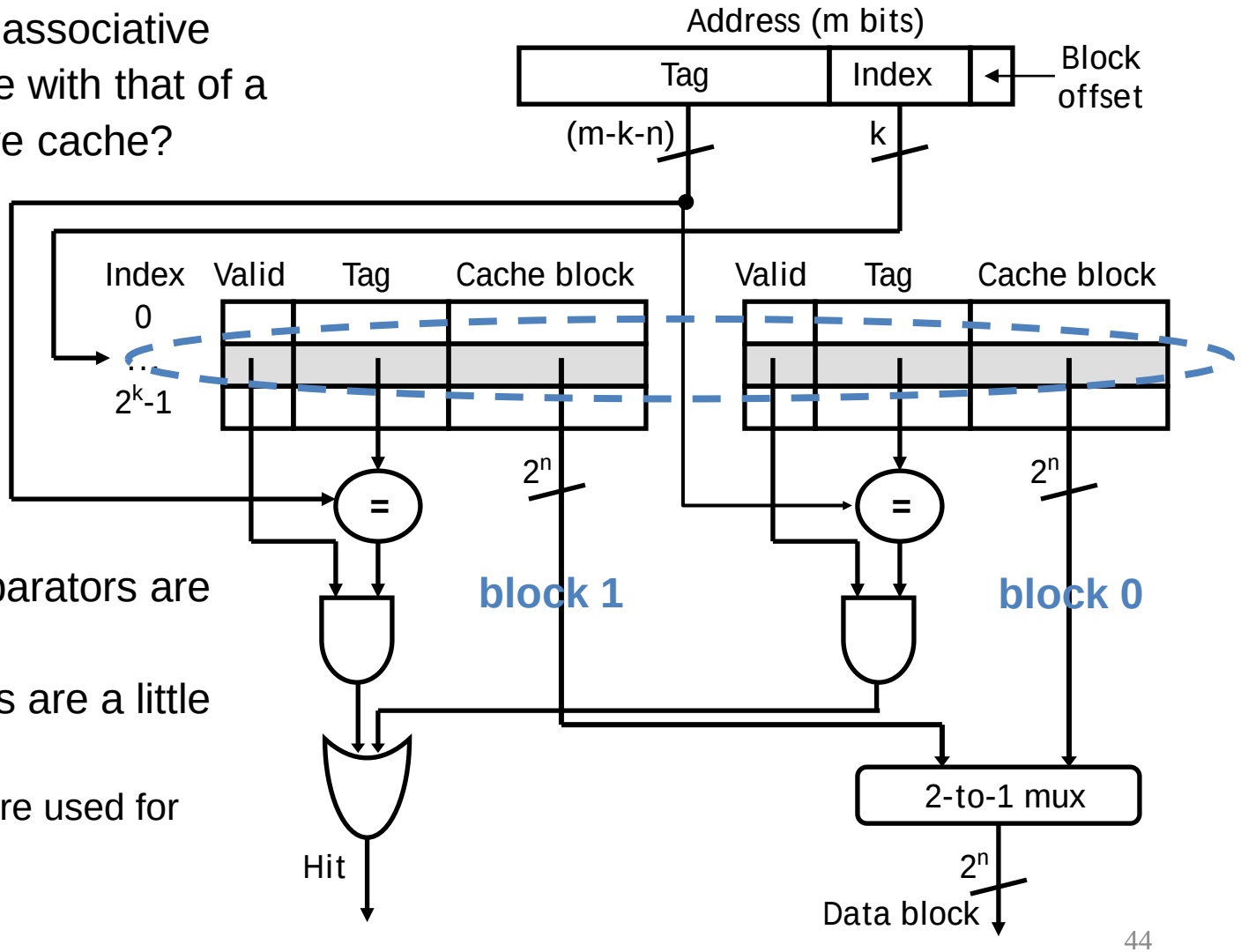
Locating a set associative block

- We can determine where a memory address belongs to an associative cache block in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



2-way set associative cache implementation

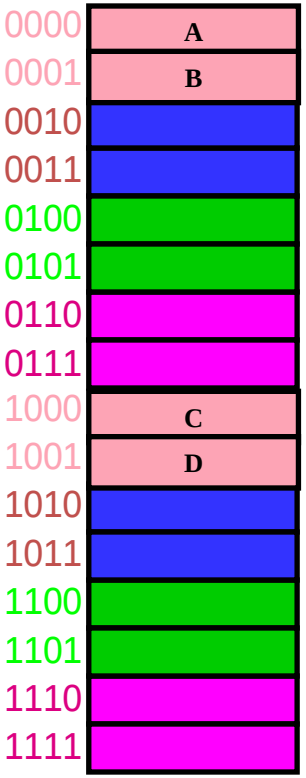
- How does an implementation of a 2-way set associative cache compare with that of a fully-associative cache?



- Only two comparators are needed.
- The cache tags are a little shorter too.
  - Some bits are used for index.

Example: a 2-way set associative cache

Memory Address



Read 0000: miss, load it from memory, store a copy into set 00. As block 0 is free, we can put it into block 0.

Read 1000: miss, load it from memory, store a copy into set 00. As block 1 is free, we can put it into block 1.

The middle two address bits are used as cache index to locate a set, the first address bit is used as tag.

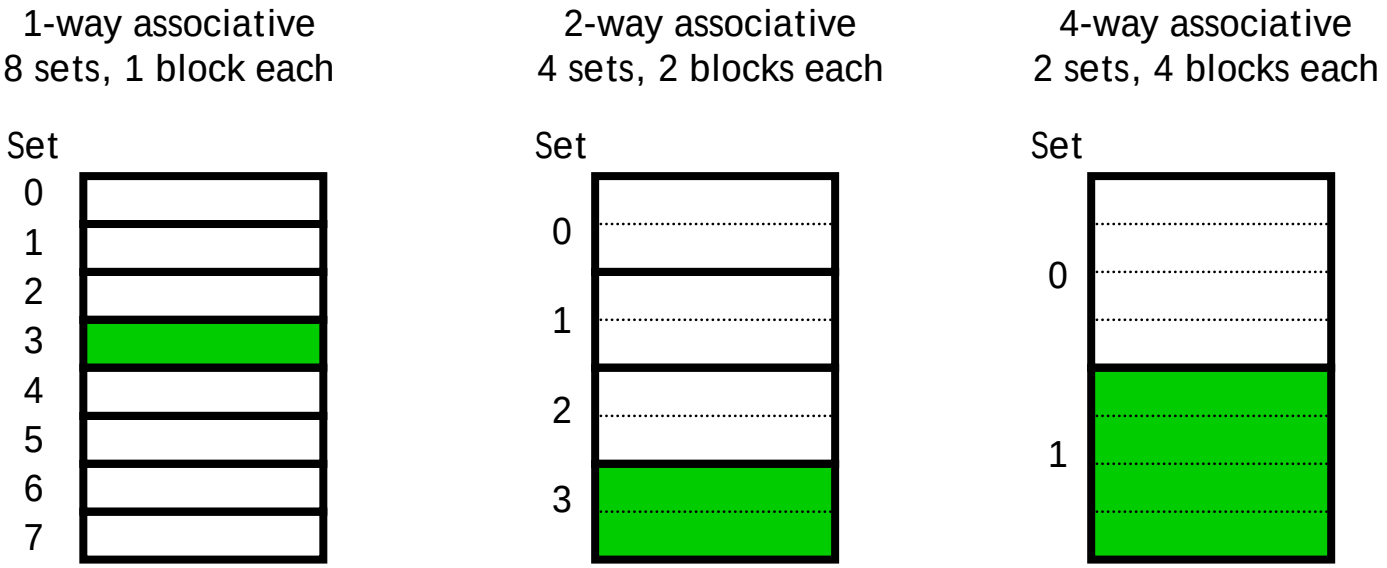
Example

- Where would data from memory byte address 6195 be placed in the cache, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011
- Each block has 16 bytes, so the lowest 4 bits are the block offset.

  For the 1-way cache, the next three bits (011) are the set index.

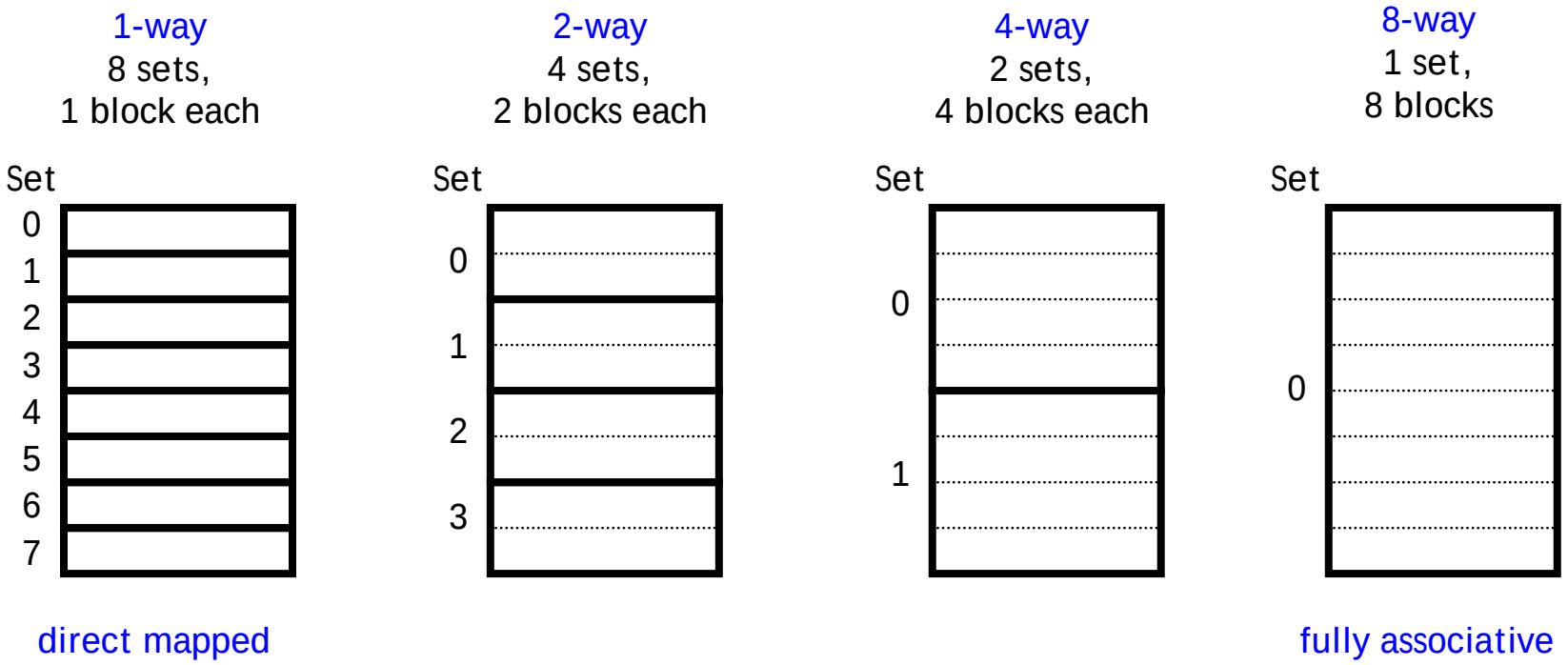
  For the 2-way cache, the next two bits (11) are the set index.

  For the 4-way cache, the next one bit (1) is the set index.
- The data may go to *any* block within the set which is in green.



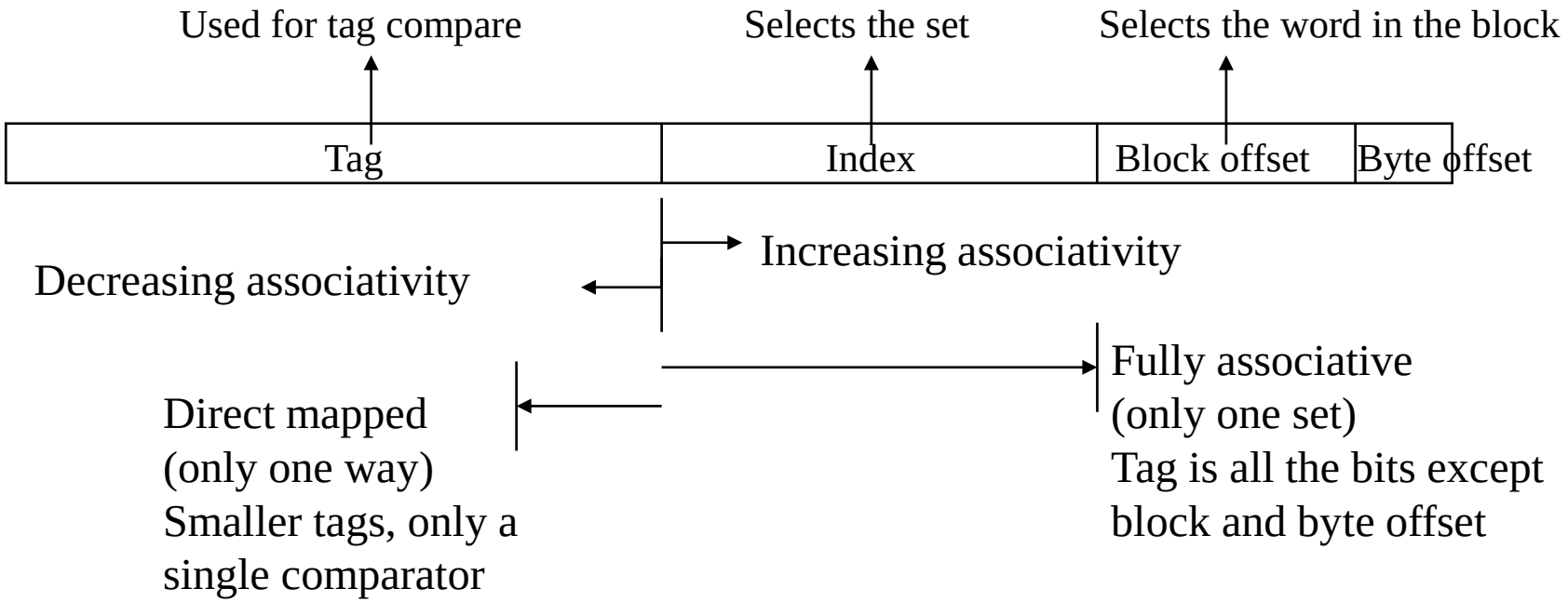
Set associative

- By now you may have noticed the 1-way set associative cache is the same as a direct-mapped cache.
- Similarly, if a cache has N blocks, a N-way set associative cache would be the same as a fully-associative cache.



Range of Set Associative Caches

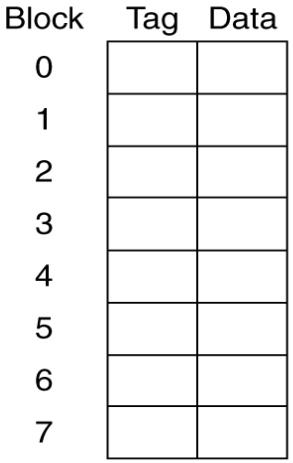
- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.



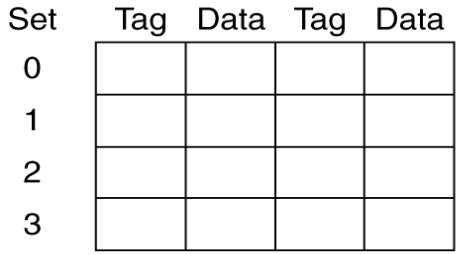
Spectrum of Associativity

- For a cache with 8 entries

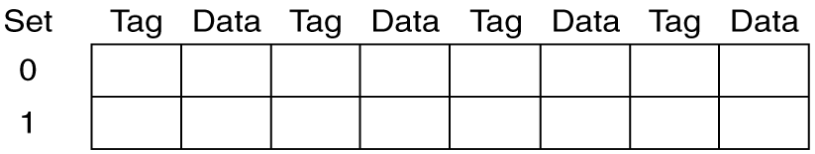
One-way set associative
(direct mapped)



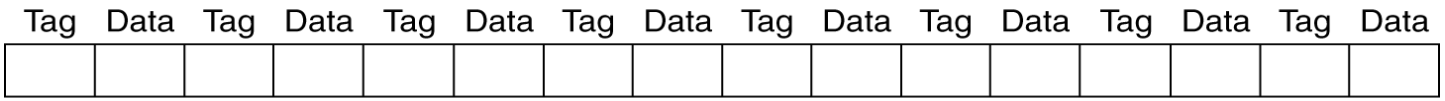
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)



Associativity Example

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

- Fully associative

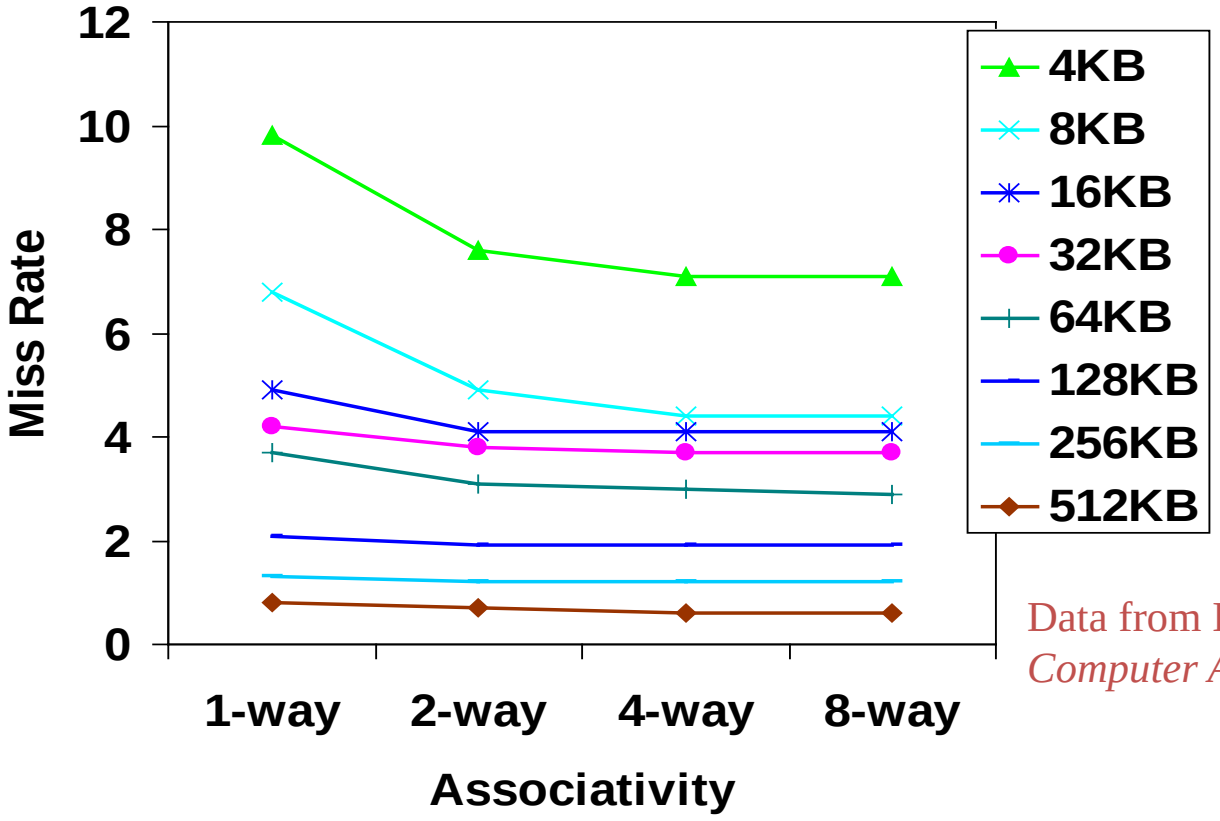
Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

How Much Associativity

- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation.



Data from Hennessy & Patterson, *Computer Architecture*, 2003

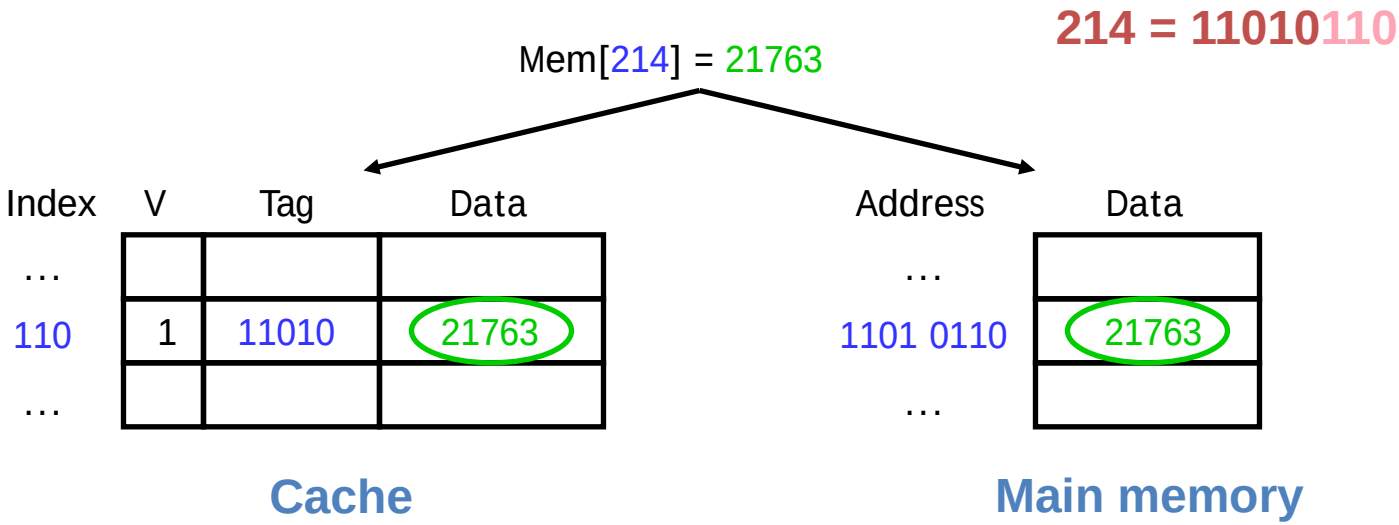
Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

How about the set is full?

- Data can be placed in any block in a set.
- Direct mapped: no choice.
- Select the least recently used (LRU) block to replace if the current set is full.
  - The block which is least accessed in recent time.
  - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set.
  - For 2-way set associative, takes one bit per set → set the bit when a block is referenced (and reset the other way's bit).
- Select the most recently used (MRU) one block to replace.
  - The block which is most accessed in recent time.
  - For random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns), MRU cache algorithms have more hits than LRU due to their tendency to retain older data.
- Random
  - Gives approximately the same performance as LRU for high associativity.

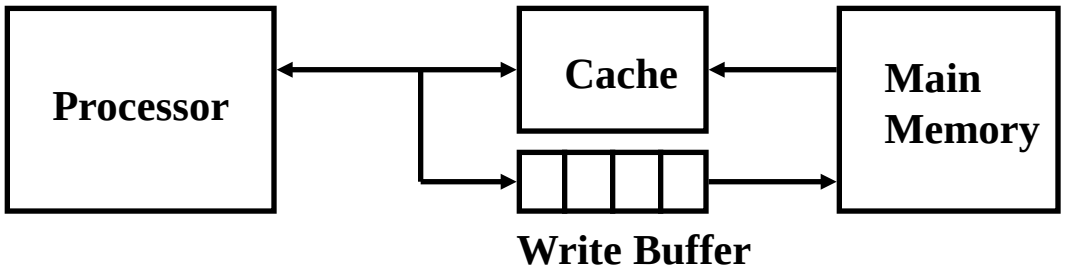
Write-hit policy: write through

- Write-hit: when processor write data (e.g. sw), the address is found in cache.
- Write through: Data is written to both the block in the cache and to the main memory.
  - Advantage: cache and main memory are always consistent.
  - Disadvantage: introduce too many writes to memory, occupy the communication bandwidth.



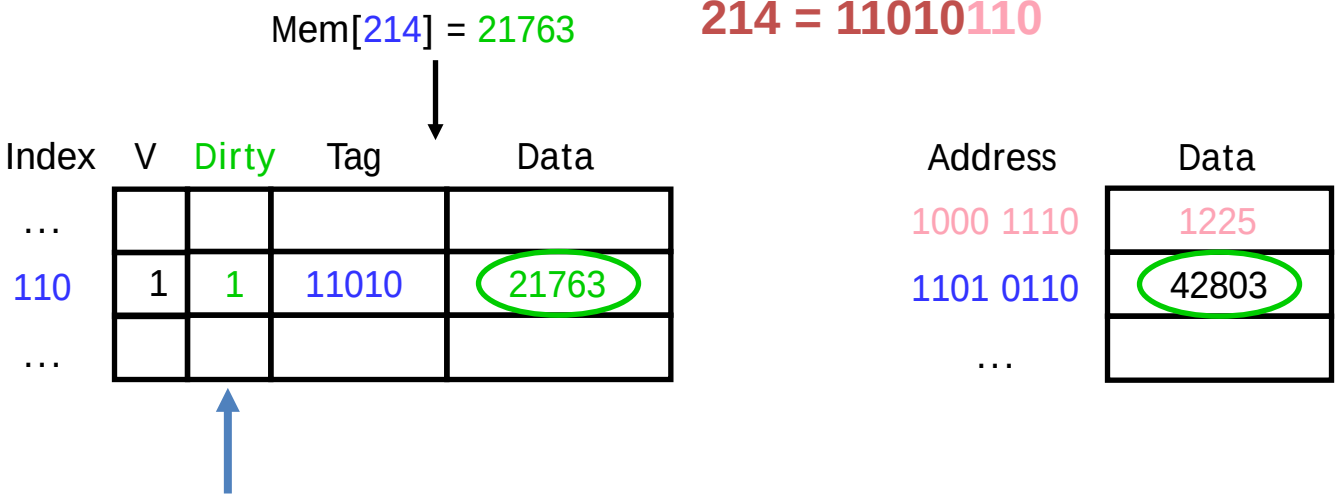
Write buffer

- Write through is always associated with write buffers so that the processor don't need to wait for the slow main memory.
- A write buffer queues the write requests from the processor, and perform the actual write operations on memory later.
  - The processor doesn't need to wait for the write operation to complete. The processor just need to issues write requests to the write buffer and continue to process other instructions.
  - If the processor issues write request too fast, extra data are stored in the buffer, which will be written to main memory later.



Write-hit policy: write back

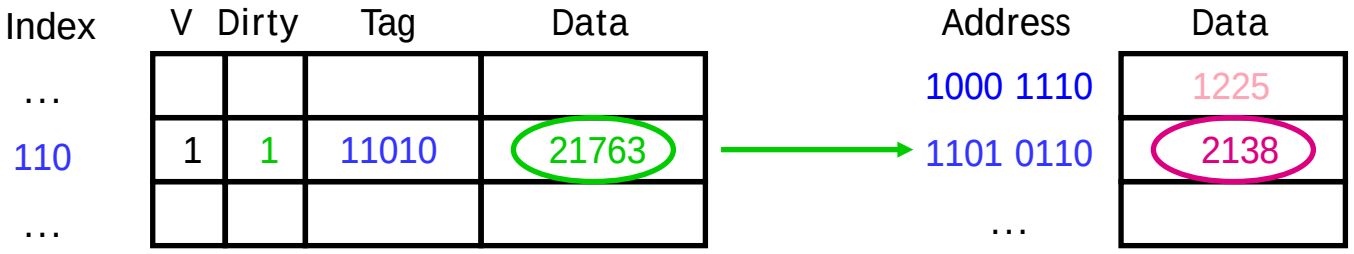
- Write back: Data is written only to the block in the cache. This modified cache block is written to main memory only when it is replaced.
  - Advantage: write to cache is fast, subsequent read will be served by cache. No need to access memory for every write.
  - Disadvantage: data inconsistency between cache and main memory.



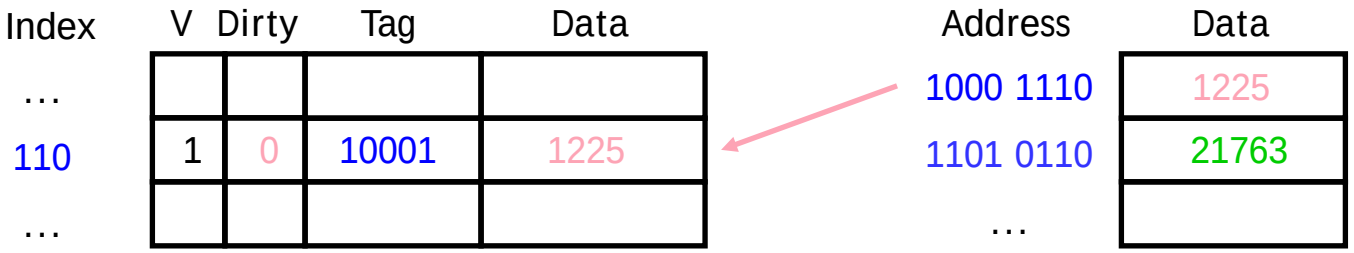
Use a dirty bit to indicate data inconsistency.

Write back: replacement when read miss

- If there is a read miss when cache is full:
- Step 1: select LRU block, check the dirty bit of the mapped cache location. If the dirty bit is 1, means the data in cache has been modified. If the dirty bit is 0, means no modification.
- Step 2: assume dirty bit is 1, we need to store data from cache to memory.

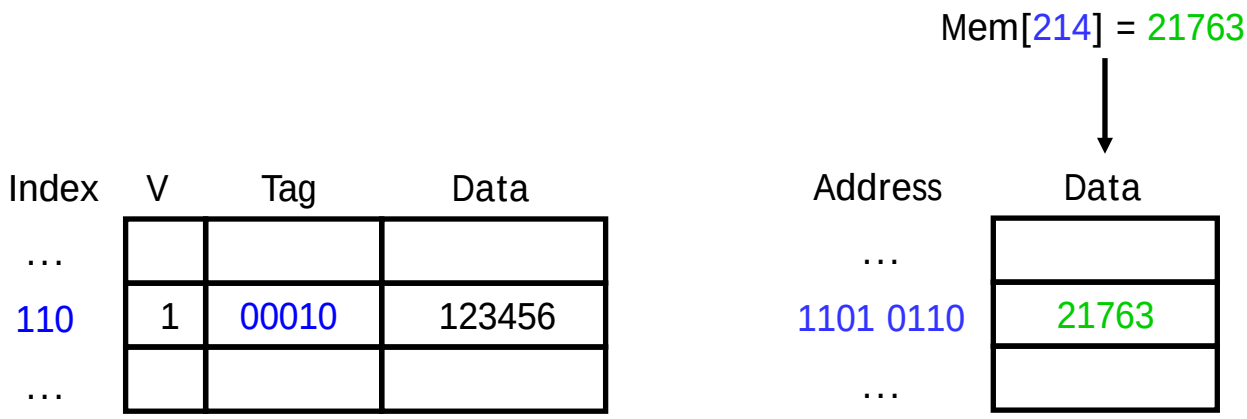


- Step 3: copy the read (requested) data from main memory to cache, set dirty bit to 0.



Write-miss Policy: Write Allocate versus Not Allocate

- When processor write data (e.g. sw), but the data block is not found in cache → write miss.
- Write Allocate
  - Load the block from memory to cache, the write is resumed and results in a write-hit.
  - Use write-hit policy to handle.
- Write Not Allocate
  - Just update memory only



Possible combinations

	Write hit policy	Write miss policy
1.	Write Through	Write Allocate
2.	Write Through	Write Not Allocate
3.	Write Back	Write Allocate
4.	Write Back	Write Not Allocate

Possible combinations

- *Write Through with Write Allocate:*
  - On hits it writes to cache and main memory.
  - On misses it loads the block from main memory to cache, and update the cache.
  - The purpose of using Write Allocate is that subsequent write to the same block will cause cache hit, so the subsequent write just need to update the cache. However, using Write Through cannot make use of this benefit.
  - Since Write Through policy is used, even on cache hit, it will generate a write to both cache and main memory → memory will still be updated which means it cannot make use of cache to save time.
- *Write Through with Write Not Allocate*
  - On hits it writes to cache and main memory.
  - On cache miss it updates the block in main memory only.
  - As a result, on cache miss, time is saved by not loading the block from memory to cache.

Possible combinations

- *Write Back with Write Allocate*
  - On hits it writes to cache only, main memory is not updated.
  - On misses it loads the block from main memory to cache, and update the cache.
  - The purpose of using Write Allocate is that subsequent write to the same block will cause cache hit, so the subsequent write just need to update cache only. Using Write Back can make use of this benefit.
  - Subsequent writes to the same block cause cache hits, Write Back just need to update cache and set dirty bit for the block. This eliminates extra memory accesses and results in very efficient execution compared with the (Write Through + Write Allocate) combination.
- *Write Back with Write Not Allocate*
  - On hits it writes to cache only, main memory is not updated.
  - On misses it updates the block in main memory only.
  - Subsequent writes to the same block, if the block originally caused a write miss, it will always generate misses and result in very inefficient execution.

Two Machines' Cache Parameters

	Intel Nehalem	AMD Barcelona
L1 cache organization & size	Split I$ and D$; 32KB for each per core; 64B blocks	Split I$ and D$; 64KB for each per core; 64B blocks
L1 associativity	4-way (I), 8-way (D) set assoc.; ~LRU replacement	2-way set assoc.; LRU replacement
L1 write policy	write-back, write-allocate	write-back, write-allocate
L2 cache organization & size	Unified; 256KB (0.25MB) per core; 64B blocks	Unified; 512KB (0.5MB) per core; 64B blocks
L2 associativity	8-way set assoc.; ~LRU	16-way set assoc.; ~LRU
L2 write policy	write-back	write-back
L2 write policy	write-back, write-allocate	write-back, write-allocate
L3 cache organization & size	Unified; 8192KB (8MB) shared by cores; 64B blocks	Unified; 2048KB (2MB) shared by cores; 64B blocks
L3 associativity	16-way set assoc.	32-way set assoc.; evict block shared by fewest cores
L3 write policy	write-back, write-allocate	write-back; write-allocate

Memory Performance

- Programmers want unlimited amounts of memory with low latency.
- Fast memory technology is more expensive per bit than slower memory.
- Solution: organize memory system into a hierarchy
  - Entire addressable memory space available in largest, slowest memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all references can be found in smaller memories
  - Gives the allusion of a large, fast memory being presented to the processor

Handling Cache Hits

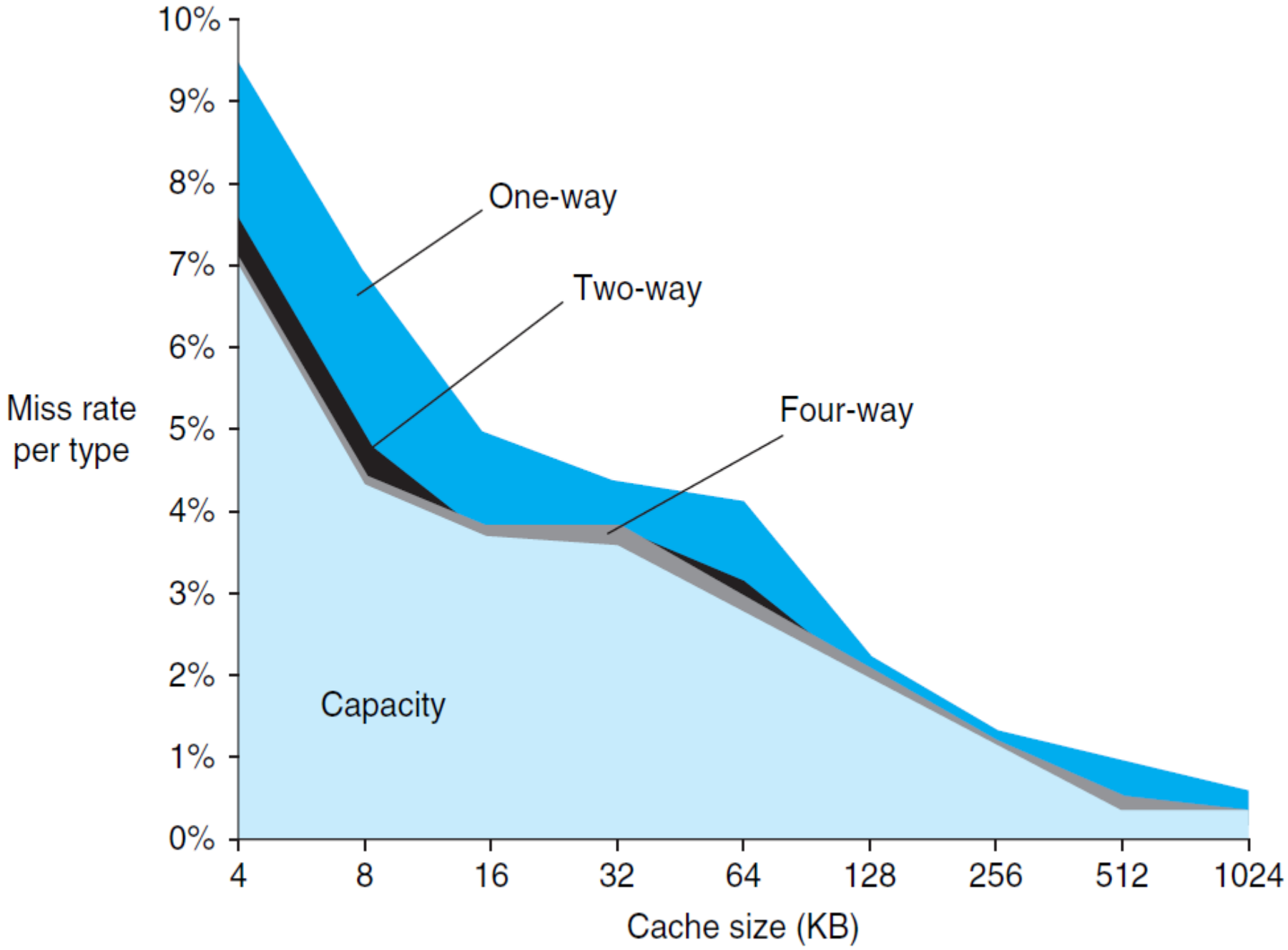
- Read hits (I$ and D$)
  - this is what we want!
- Write hits (D$ only)
  - require the cache and memory to be consistent
    - require the cache and memory to be consistent the memory hierarchy (write-through)
    - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a write buffer and stall only if the write buffer is full.
  - allow cache and memory to be inconsistent
    - write the data only into the cache block (write-back the cache block to the next level in the memory hierarchy when that cache block is "evicted").
    - need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a write buffer to help "buffer" write-backs of dirty blocks.

Handling Cache Misses

- Read misses (I$ and D$)
  - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume.
- Write misses (D$ only)
  - stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume.
  - Write allocate – just write the word into the cache updating both the tag and data, no need to stall.
  - Write not allocate – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full.

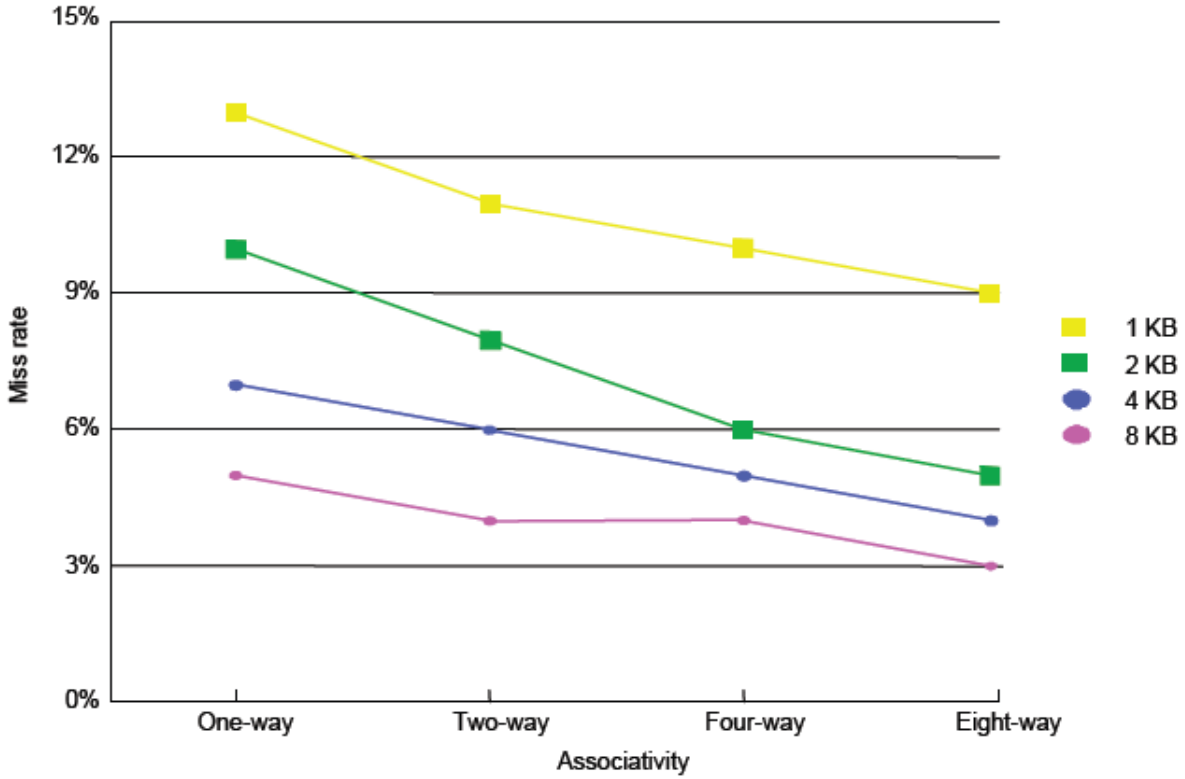
Cache Misses

- Miss rate
  - Fraction of cache access that result in a miss
- Causes of misses
  - Compulsory (cold start or process migration, first (cold start or process migration, first reference):
    - First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant.
    - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate).
  - Capacity:
    - Cache cannot contain all blocks accessed by the program.
    - Solution: increase cache size (may increase access time).
  - Conflict (collision):
    - Multiple memory locations mapped to the same cache location.
    - Solution 1: increase cache size.
    - Solution 2: increase associativity (may increase access time).


Miss rate per type
10%
9%
8%
7%
6%
5%
4%
3%
2%
1%
0%
One-way
Two-way
Four-way
Capacity
4
8
16
32
64
128
256
512
1024
Cache size (KB)

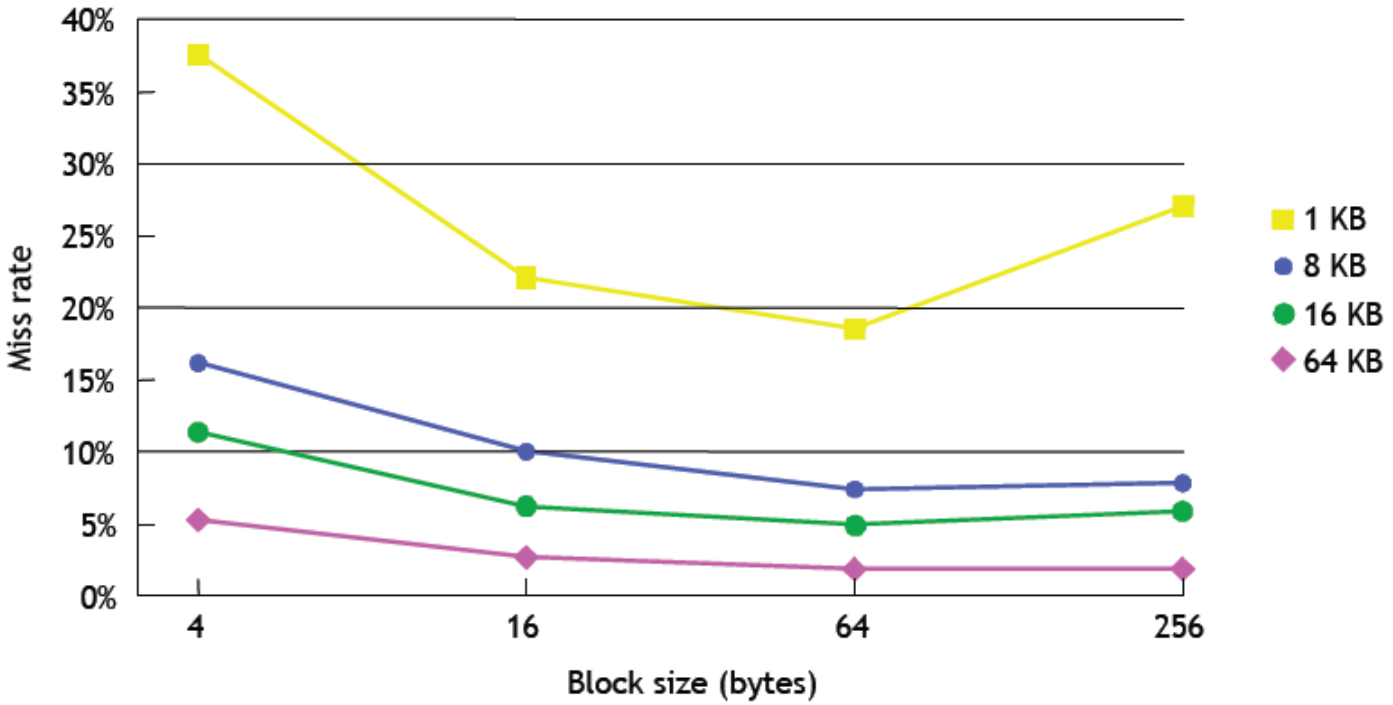
Cache size and miss rate

- The cache size also has a significant impact on performance.
  - Larger cache can reduce address conflict.
  - Again this means the miss rate decreases, so the AMAT and number of memory stall cycles is also lower.



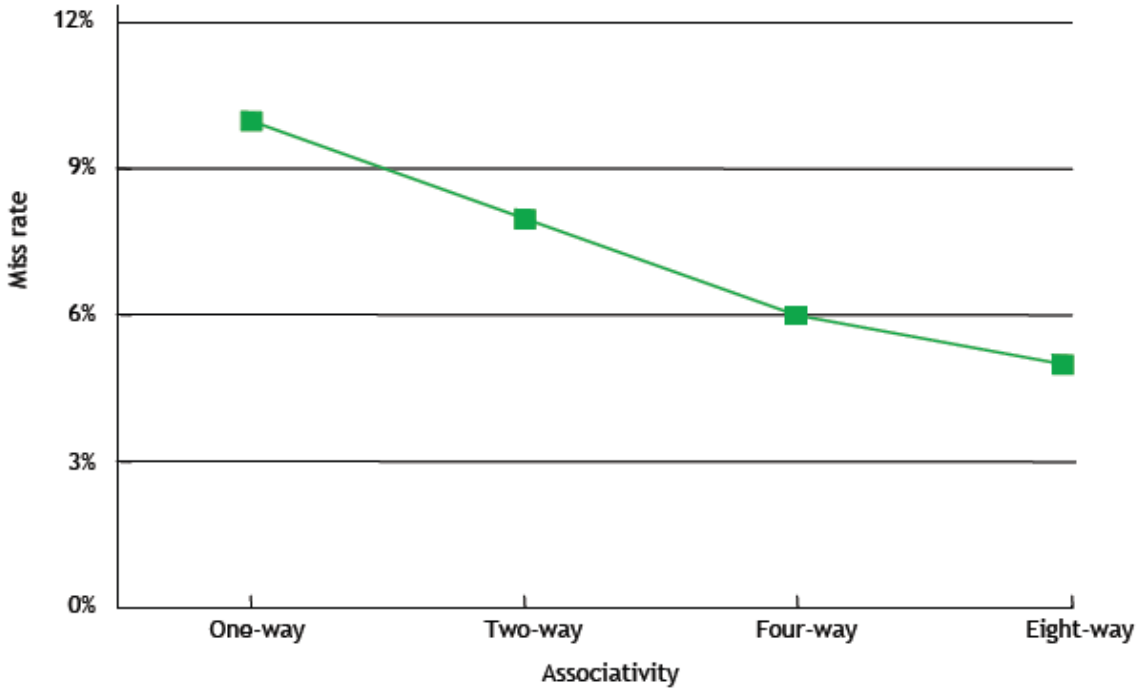
Block size and miss rate

- The following figure shows miss rates relative to the block size and overall cache size.
  - Smaller block size doesn't take maximum advantage of spatial locality.
  - But if blocks are *too* large, the number of blocks is smaller, which may increase address conflict.



Associativity and miss rate

- Higher associative cache means more complex hardware. But a highly-associative cache will also exhibit a lower miss rate.
  - Each set has more blocks, it helps to reduce address conflict.
  - Overall, this will reduce AMAT and memory stall cycles.



Memory and overall performance

- Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{CC}$$
$$= \text{IC} \times \underbrace{(\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles})}_{\text{CPI}_{\text{stall}}} \times \text{CC}$$

- The total number of stall cycles depends on the number of cache misses *and* the miss penalty.

  Memory stall cycles = Num of memory accesses x miss rate x miss penalty

- To include stalls due to cache misses in CPU performance equations, we have to add them to the "base" number of execution cycles.

  CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

Impacts of Cache Performance

- Assume the total number of instruction in a program is N, and 33% of the instructions are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

  Memory stall cycles = Memory accesses x Miss rate x Miss penalty

  = 0.33 x N x 0.03 x 20 cycles

  = 0.2 x N cycles

- The number of wasted cycles is 0.2N, total cycles will be 1.2N cycles.
- This code is 1.2 times slower than a program with a “perfect” CPI of 1!

Average memory access time (AMAT)

- Factors that affect memory performance:
  - Hit time: the access time when cache is hit, i.e. the cache accessing time.
  - Miss rate: the percentage of memory accesses that cannot be handled by the cache.
  - Miss penalty: the additional time to load data from main memory for a cache miss.
- The average memory accessing time (AMAT) is determined by the above three factors:

  AMAT = Hit time + (Miss rate $\times$ Miss penalty)
- E.g. L1 hit time = 2, L2 hit time = 16, Memory time = 200, L1 miss rate = 1%, L2 miss rate = 5%

  AMAT = 2 +
  0.01 $\times$ 16 +
  0.01 $\times$ 0.05 $\times$ 200
  = 2.26 cycles

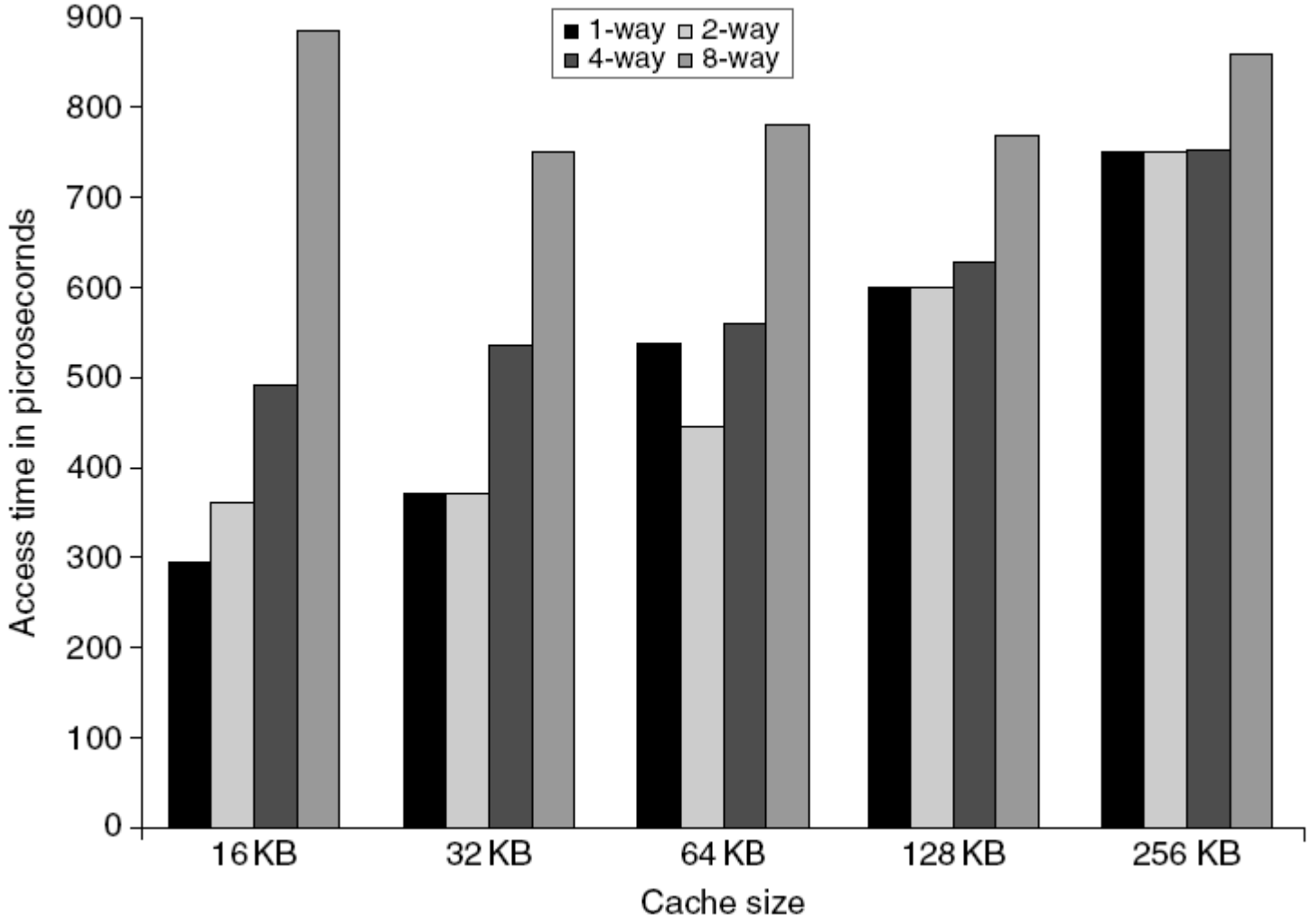
Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different.
  - Primary cache should focus on minimizing hit time in support of a shorter clock cycle.
    - Smaller with smaller block sizes
    - Direct-mapped
      - caches can overlap tag compare and transmission of data
    - Lower associativity
      - reduces power because fewer cache lines are accessed
  - Secondary cache(s) should focus on reducing miss rate to reduce the penalty of long main memory access times.
    - Larger with larger block sizes
    - Higher levels of associativity

Multilevel Cache Design Considerations

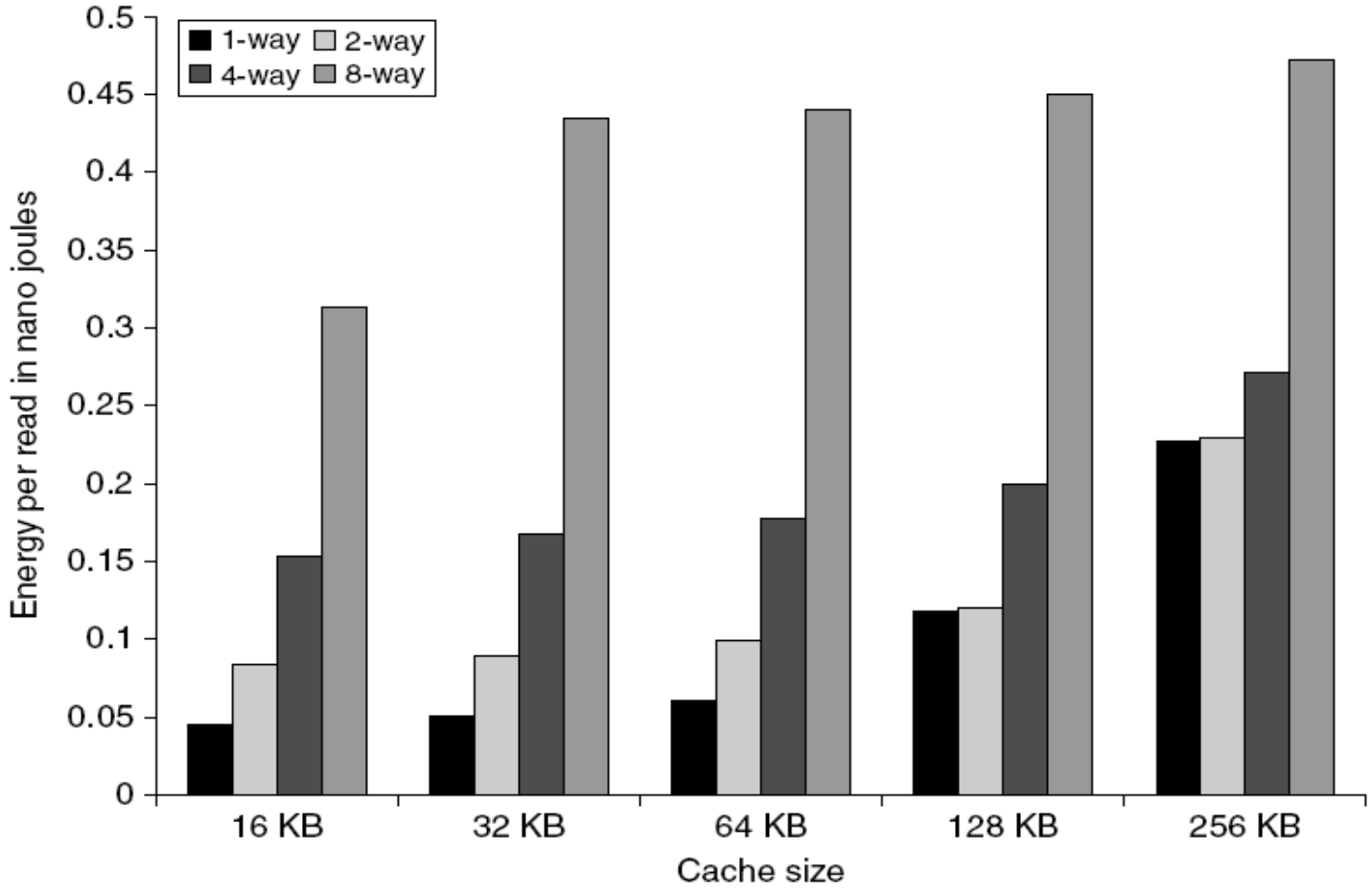
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate.
- For the L2 cache, hit time is less important than miss rate.
  - The L2$ hit time determines L1$'s miss penalty.
  - L2$ local miss rate >> than the global miss rate.

L1 Size and Associativity



Access time vs. size and associativity

L1 Size and Associativity



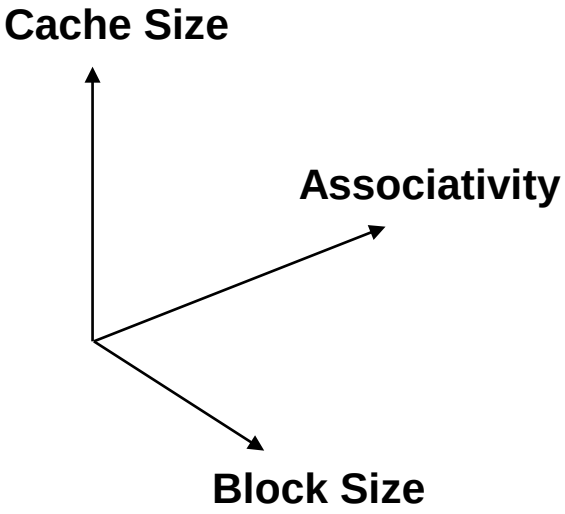
Energy per read vs. size and associativity

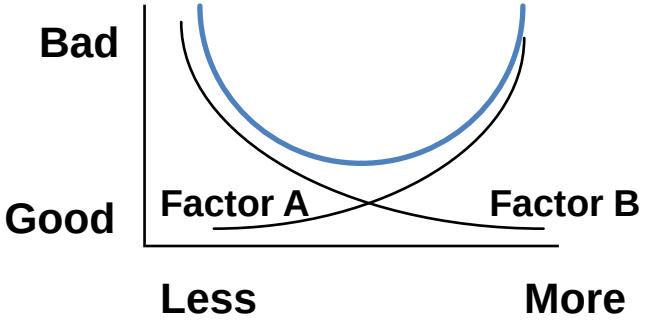
Summary: Improving Cache Performance

- Reduce the time to hit in the cache
  - smaller cache
  - direct mapped cache
  - smaller blocks
- Reduce the miss rate
  - bigger cache
  - more flexible placement (increase associativity)
  - larger blocks (16 to 64 bytes typical)
  - victim cache – small buffer holding most recently discarded blocks
- Reduce the miss penalty
  - smaller blocks
  - use multiple cache levels – L2 cache not tied to CPU clock rate
  - faster backing store/improved memory bandwidth
    - wider buses
    - memory interleaving, DDR SDRAMs

Summary: The Cache Design Space

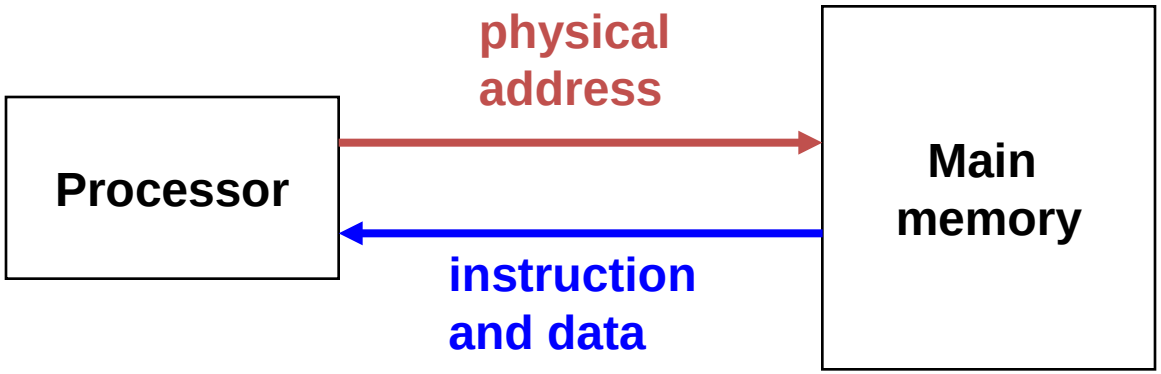
- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





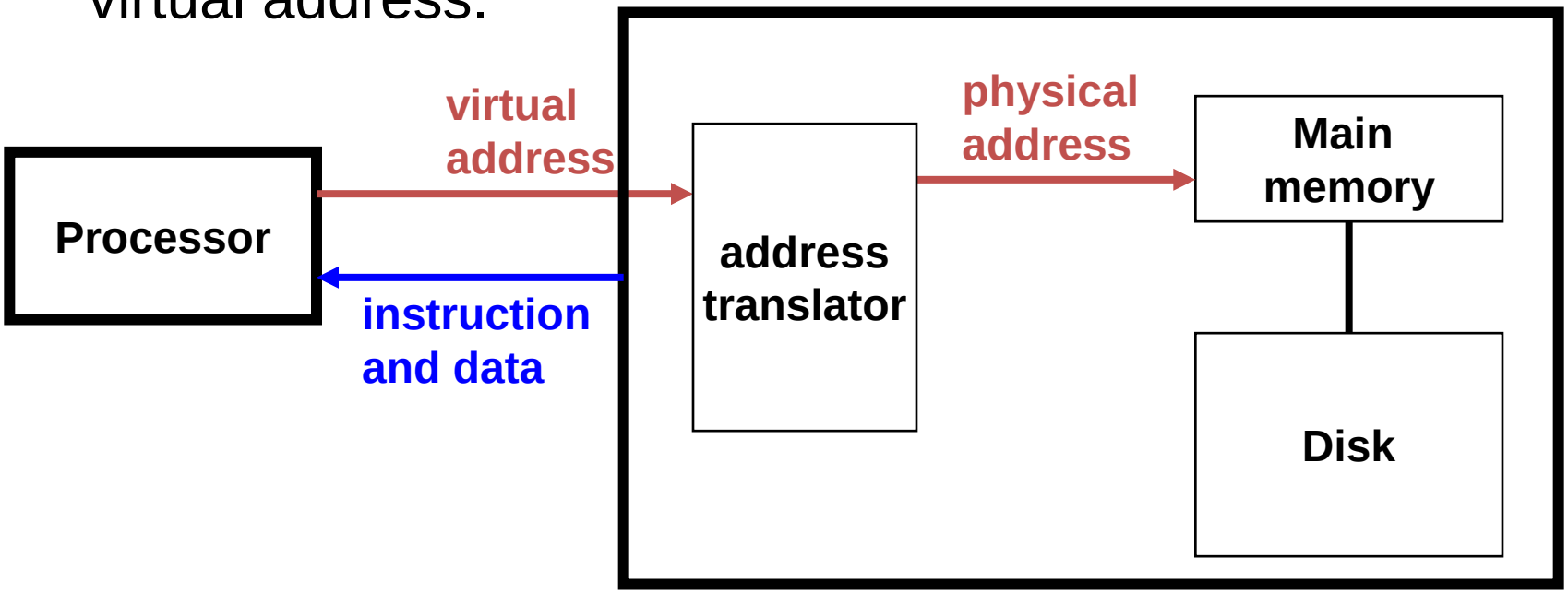
Problems

- Not enough main memory?
  - A program requests too much memory space.
- Multiple programs?
  - Different programs try to access the same address → conflict.
  - How to protect a program's data from being modified by other programs?



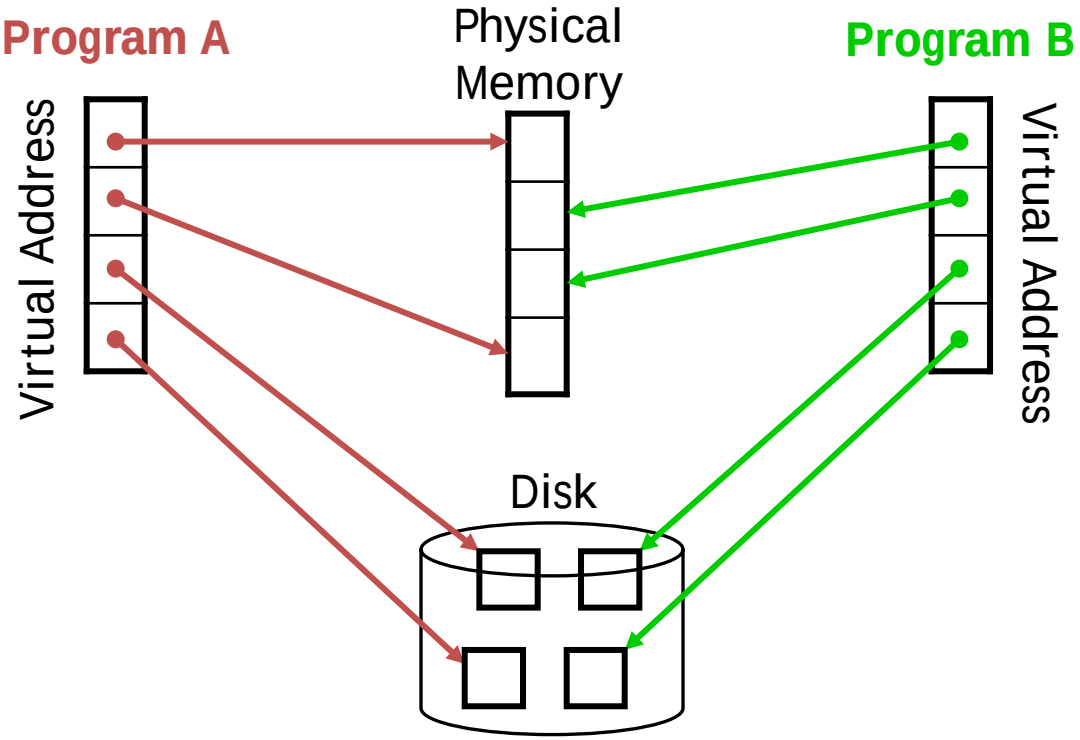
Solution for size limitation

- Create a big virtual memory and presents to processor.
  - The processor just know it has a big memory.
  - Actually, some data are stored in main memory, some are stored in hard disk. Processor doesn't know where data are stored.
- Processor read/write data from virtual memory through a virtual address.



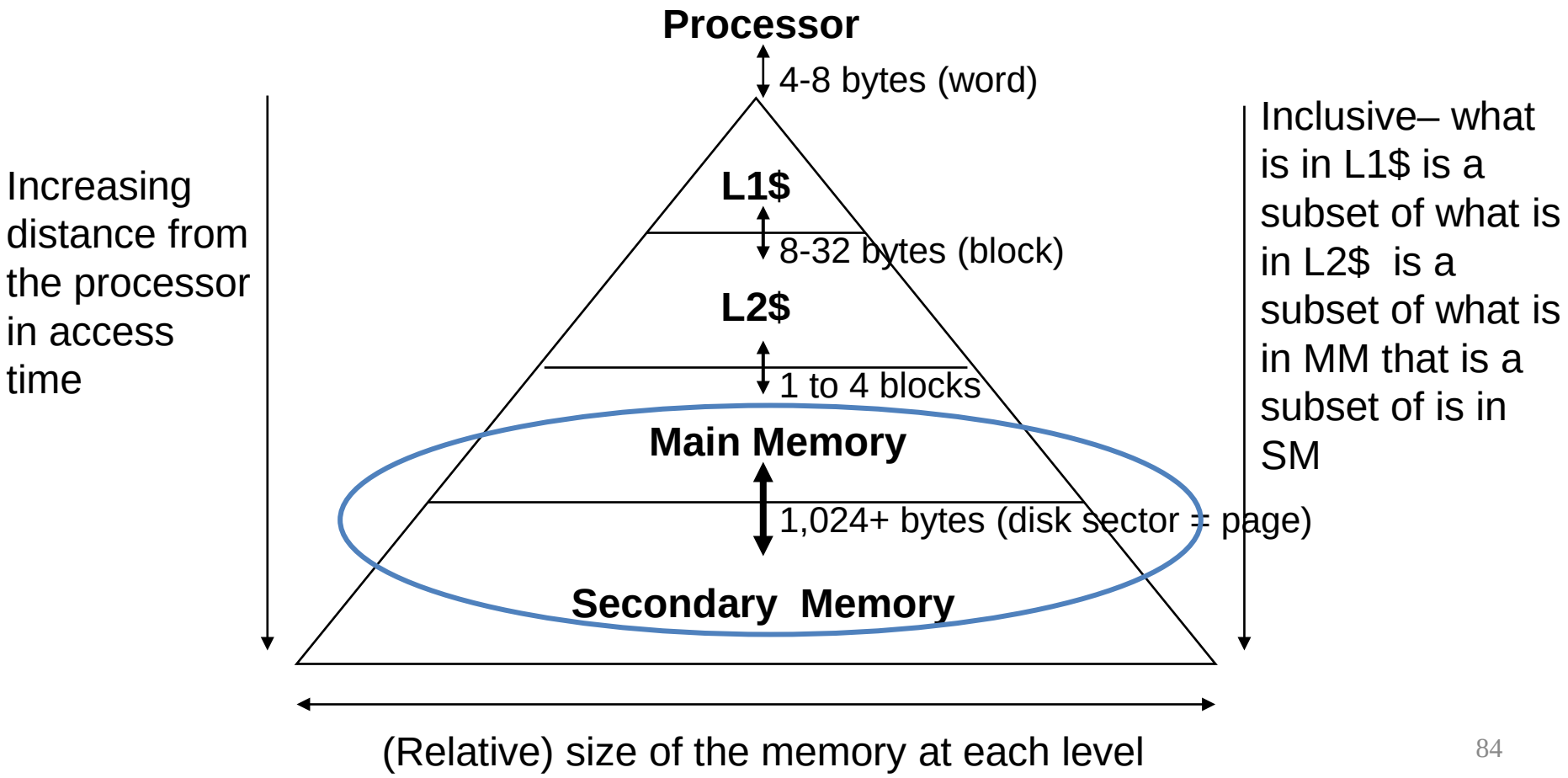
Solution for conflict

- Each program can use the virtual address arbitrarily, and use as much space (virtual memory) as it wants.
- E.g. two distinct programs can use the same virtual address, which can be mapped to two different physical memory locations by the operating system → resolve address conflict.



Review: The Memory Hierarchy

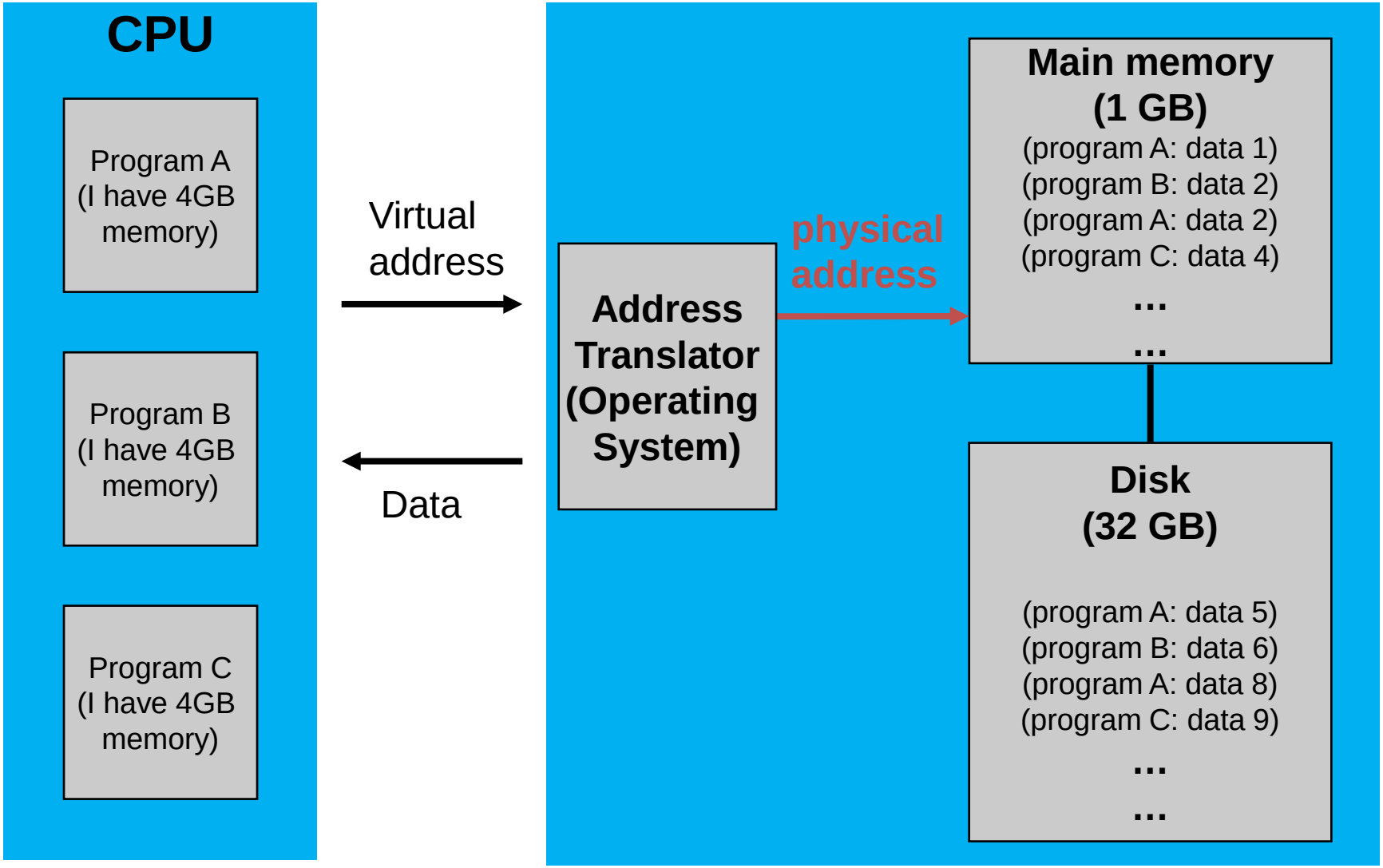
- Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology



Virtual Memory

- Use main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
  - Managed jointly by CPU hardware and the operating system (OS)
- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
  - During run-time each virtual address must be translated to a physical address (an address in main memory)
  - VM “block” is called a page
  - VM translation “miss” is called a page fault

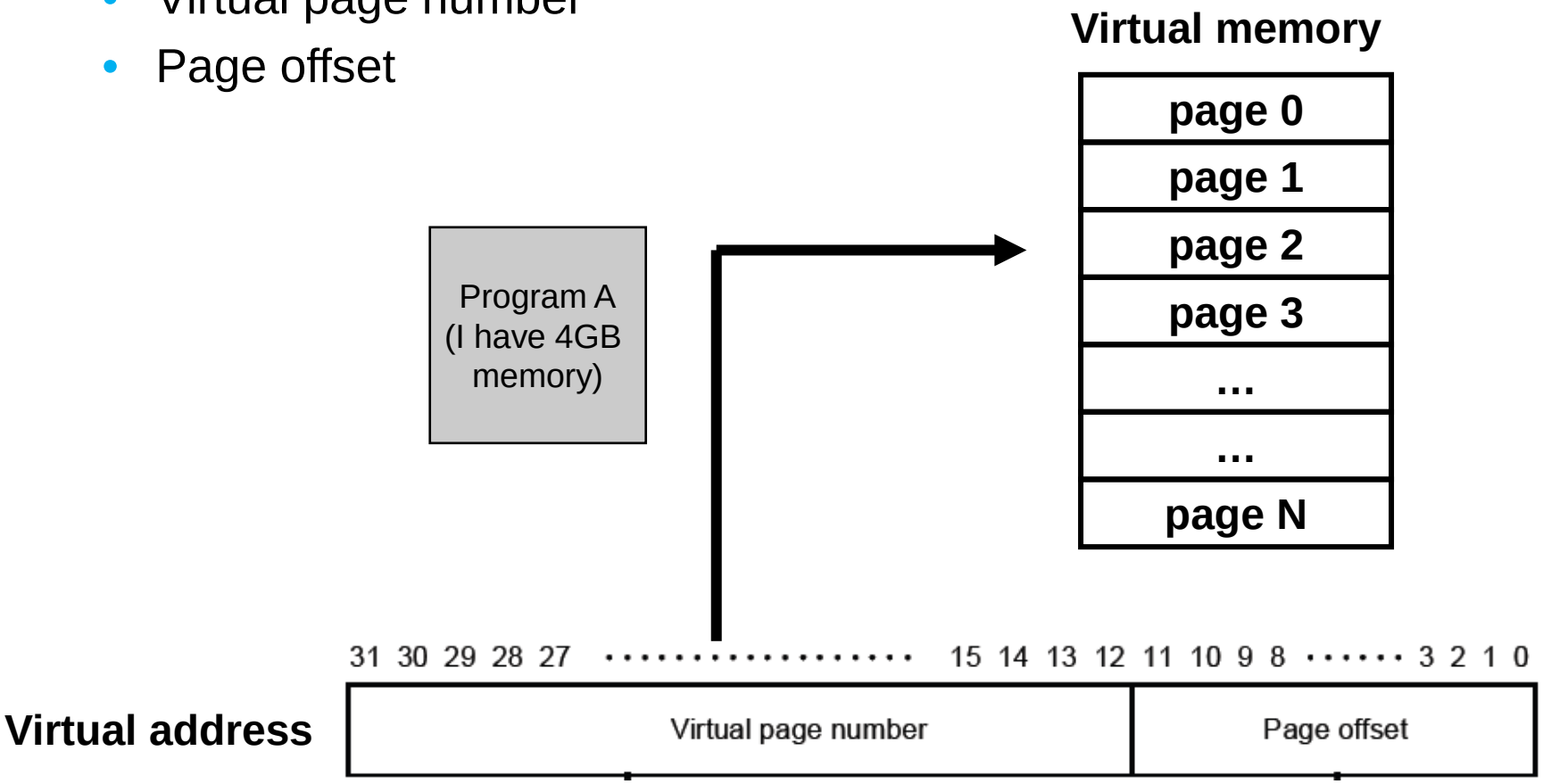
The big picture (32-bit address)



- Use the main memory as a cache, reduce the read/write delay of accessing disk.
- Fully associative, least recently used policy for replacement.

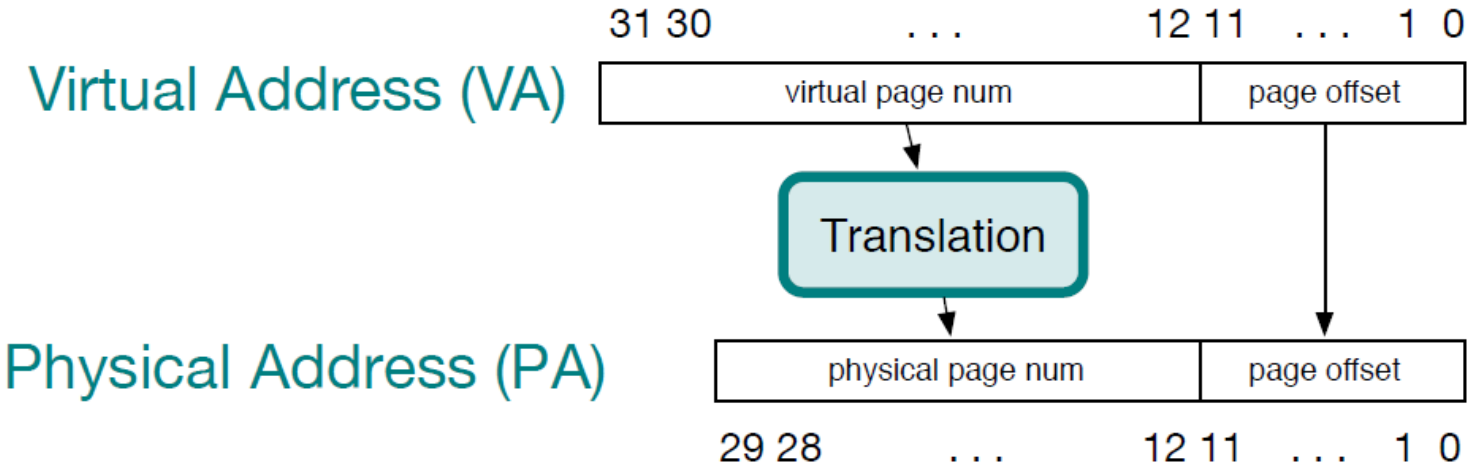
Virtual memory

- Virtual memory is divided into pages.
- Virtual address is divided into two fields.
  - Virtual page number
  - Page offset

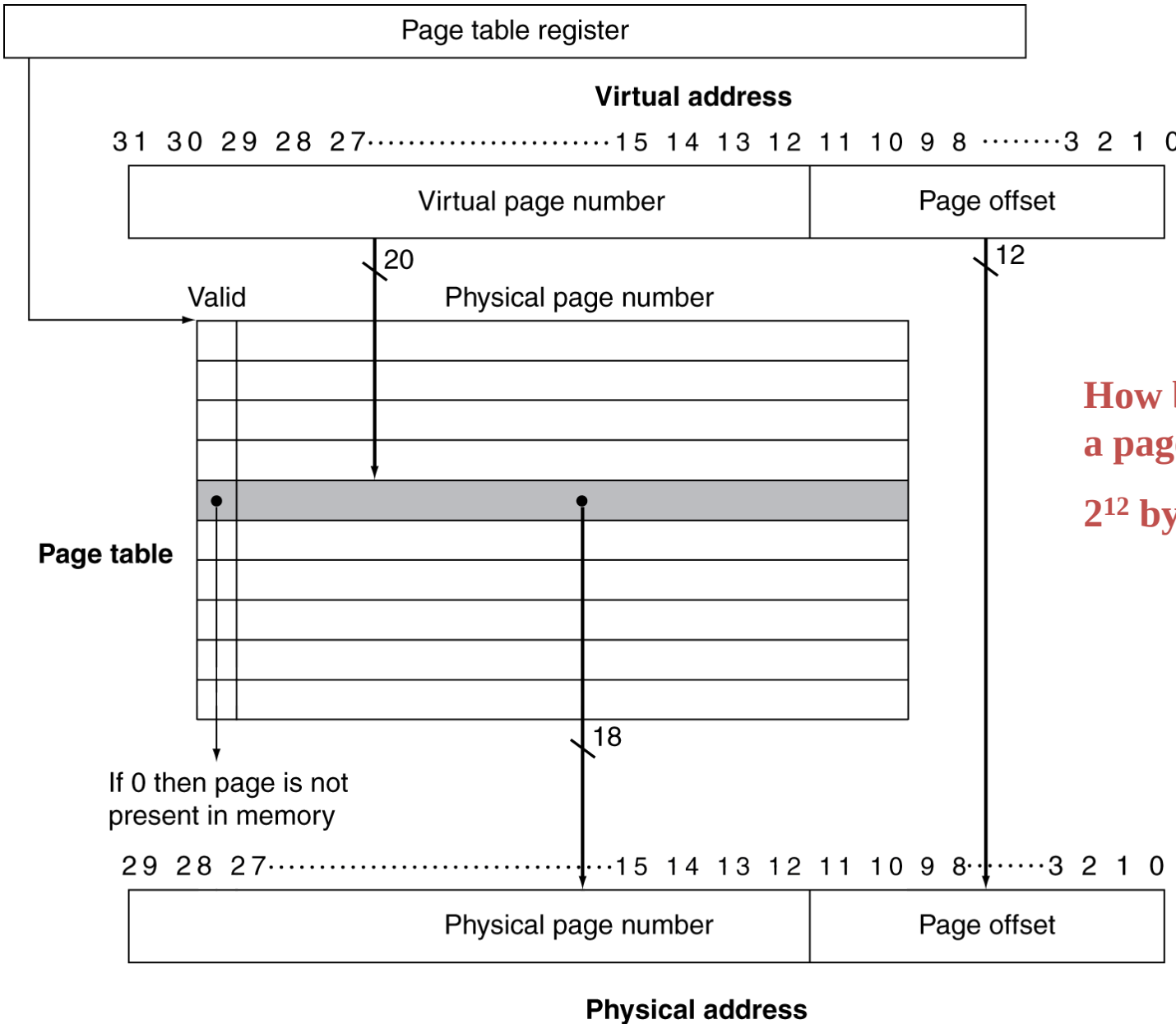


Convert virtual address to physical address

- The conversion is done using a page table.
- Each program (process) has a separate page table stored in main memory. A “page table register” points to the page table of current program (process).
- The page table is indexed with the virtual page number (VPN)
- Each entry contains a valid bit, and a physical page number (PPN)
  - The PPN is concatenated with the page offset to form a physical address.
- No tag is needed because the index is the full VPN.
- Virtual address → physical address by combination of HW/SW.
- Each memory request needs first an address translation.



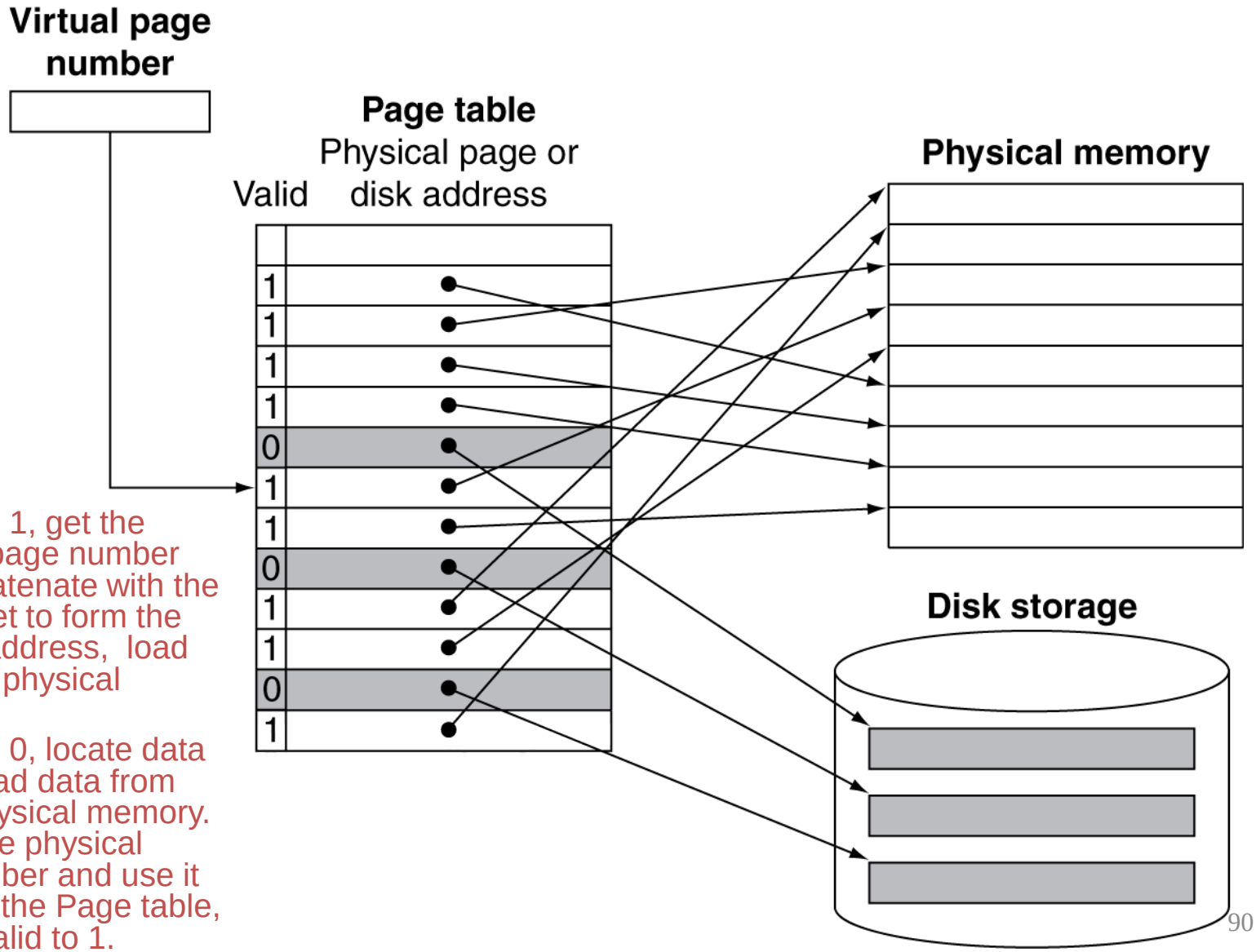
Convert virtual address to physical address



How big is a page?

2^{12} bytes

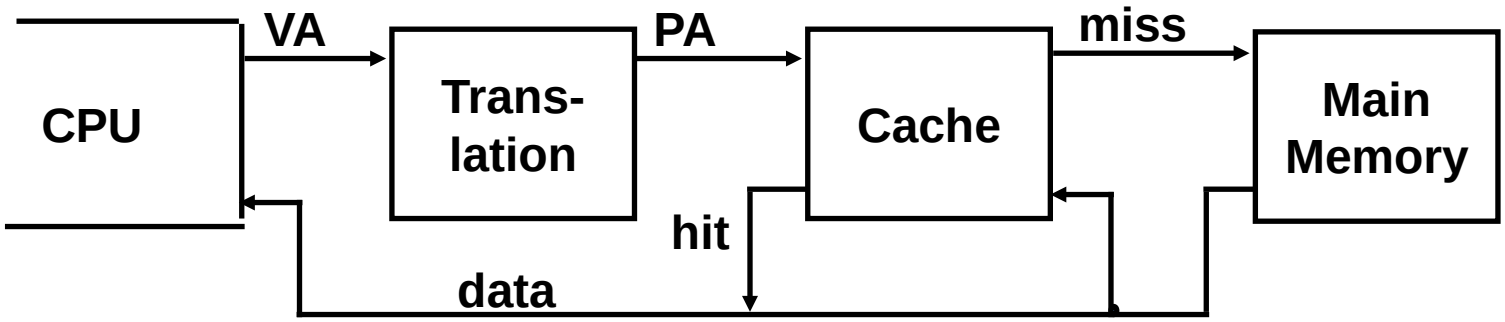
Loading data using virtual address



- If Valid == 1, get the Physical page number and concatenate with the page offset to form the physical address, load data from physical memory.
- If Valid == 0, locate data in disk, load data from disk to physical memory. Record the physical page number and use it to update the Page table, change Valid to 1.

Virtual Addressing with a Cache

- Thus it takes an *extra* memory access to translate a VA to a PA.



Problems

- How big is the page table?
  - In the previous example, virtual page number is 20-bit long → the table contains 2^{20} entries.
- Where can we store this table?
  - It can be stored in main memory, but the time for an address translation is much longer. We need to access the main memory and do an address translation.

To read a data using virtual address:

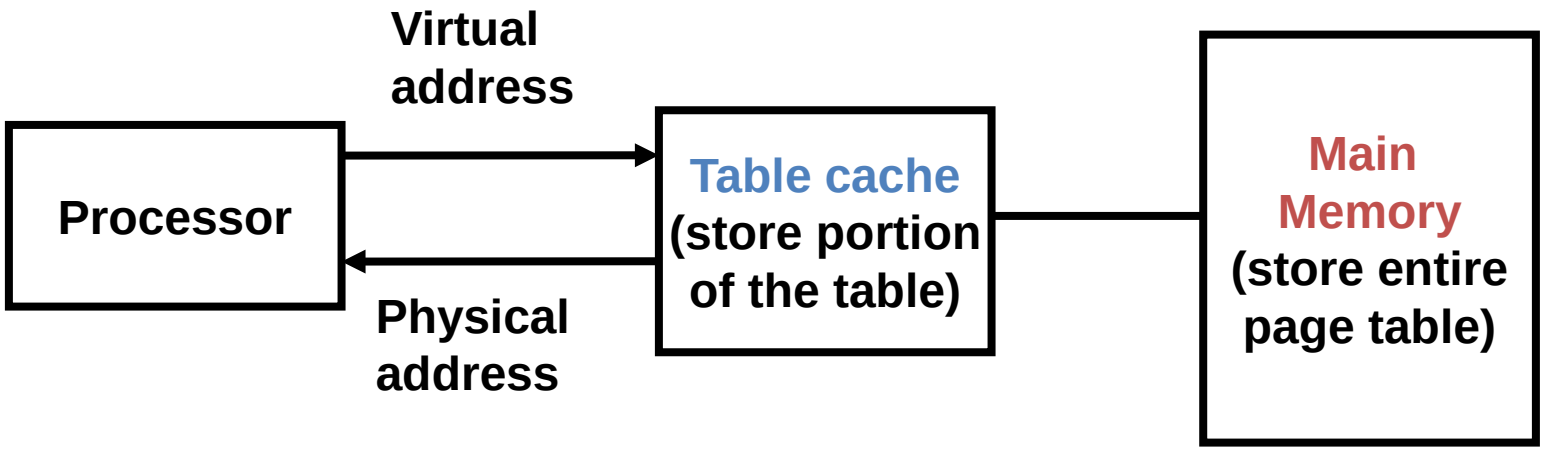
Step 1: read the page table stored in main memory, load the physical page number and calculate the physical address.

Step 2: load data using the physical address.

→ Involves two memory accesses!!

Solution

- Add a cache to store portion of the page table.
- Instead of read table entries from main memory, load table entries from cache.
  - Just like the data cache introduced before, the only difference is that the cache here is used to store table entries.



Translation Lookaside Buffers (TLBs)

- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.
- TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
  - TLBs are typically not more than 512 entries even on high end machines.

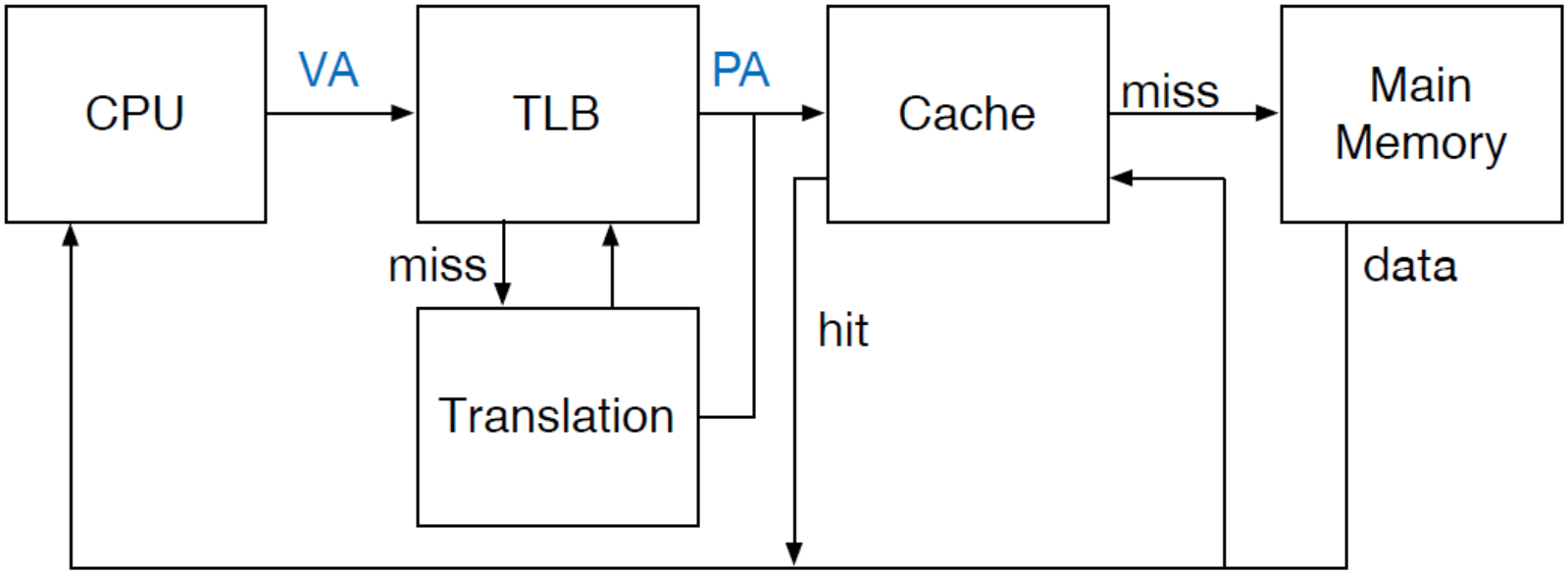
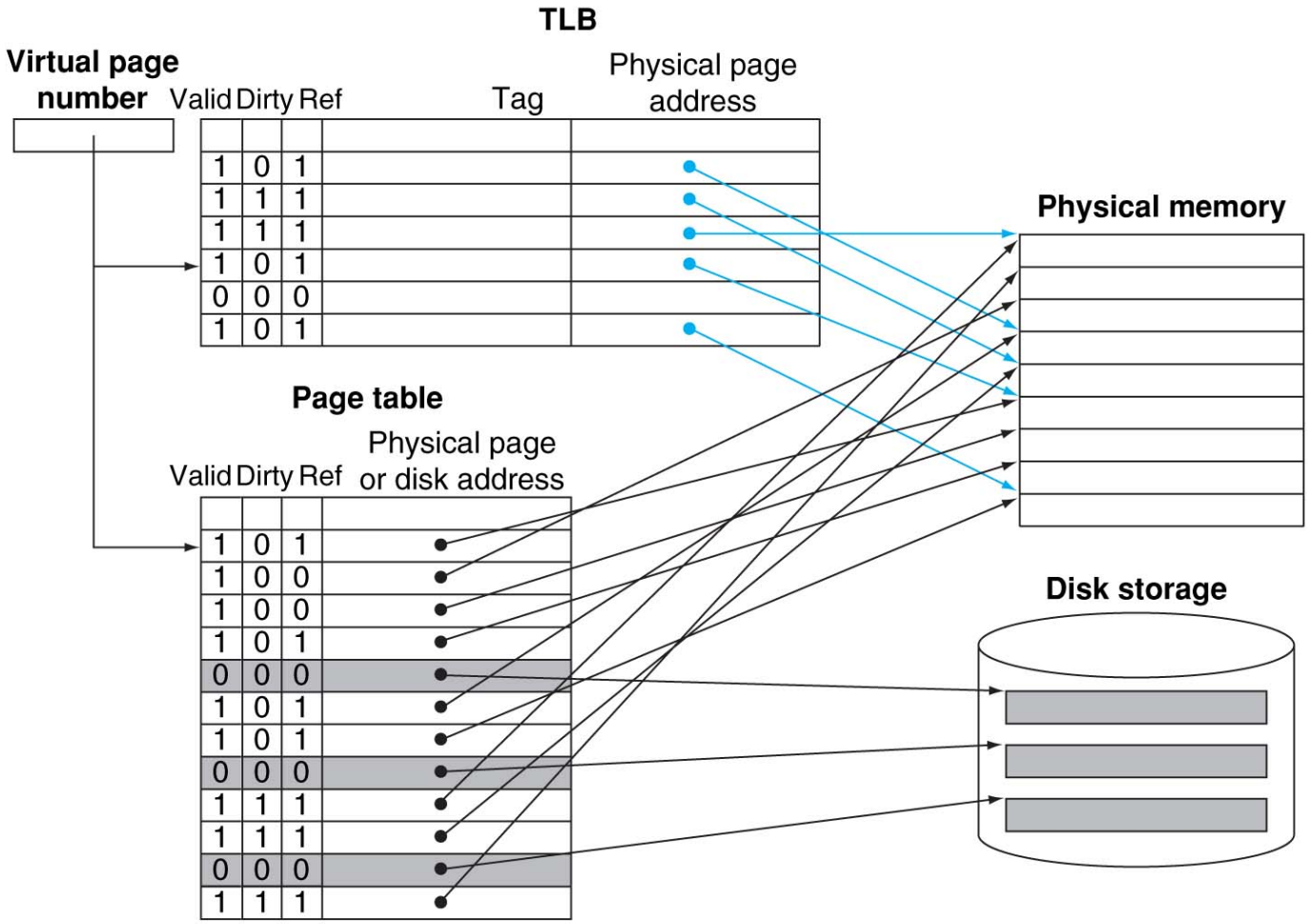


Table cache: Translation Lookaside Buffer (TLB)



1. TLB is a fast cache to store a portion of the page table.
2. As a result, instead of two memory accesses for each read, now we need 1 cache access and 1 memory access.
3. Since it is a cache, we need a tag field.

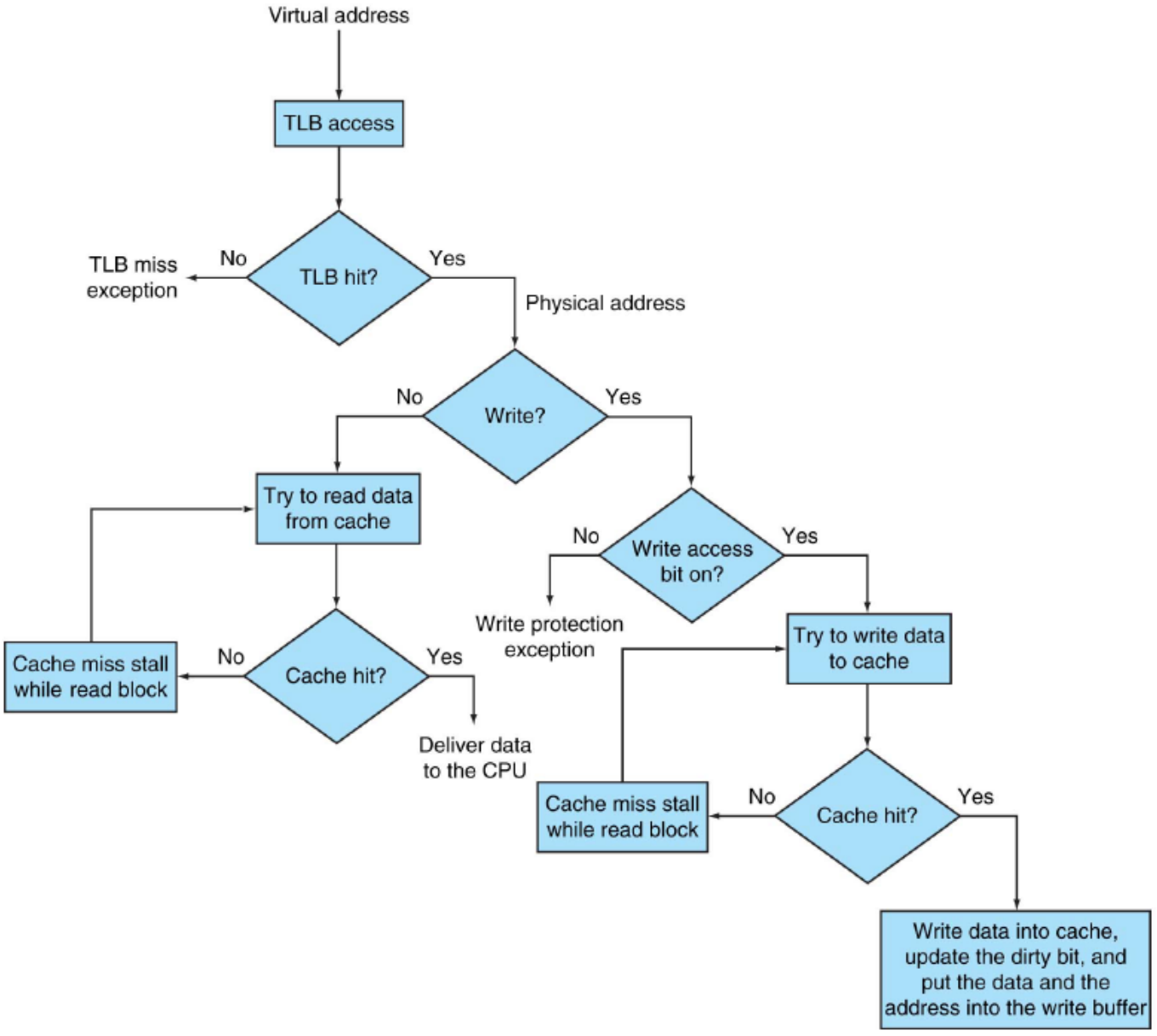
TLB miss and page fault

- Address cannot be found in TLB → TLB miss
  - Need to search the page table (e.g. in main memory), load the table entry into TLB, and the program will be resumed.
- If a table entry is found, but the valid bit is off. It means data is not in main memory → page fault.
  - Handled by operating system.
  - Copy data from disk to main memory, which may previously be swapped out of main memory to make room for other processes.
  - Record the physical page number of the data in main memory.
  - Update page table entry: store the physical page number in the table entry indexed by the virtual page number, update the valid bit.
  - Resume the program, now the data can be found in main memory.

TLB miss and page fault

- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB.
    - Takes 10's of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it's a true page fault.
    - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults.
- On page fault, the page must be fetched from disk.
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

Cooperation of TLB & Cache



TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – TLB miss, PA in page table
Miss	Hit	Miss	Yes – TLB miss, PA in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

4 Questions for the Memory Hierarchy

- Q1: Where can a entry be placed in the upper level? *(Entry placement)*
- Q2: How is a entry found if it is in the upper level? *(Entry identification)*
- Q3: Which entry should be replaced on a miss? *(Entry replacement)*
- Q4: What happens on a write? *(Write strategy)*