Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2024

Lecture 20:
Memory Hierarchy pt 3

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

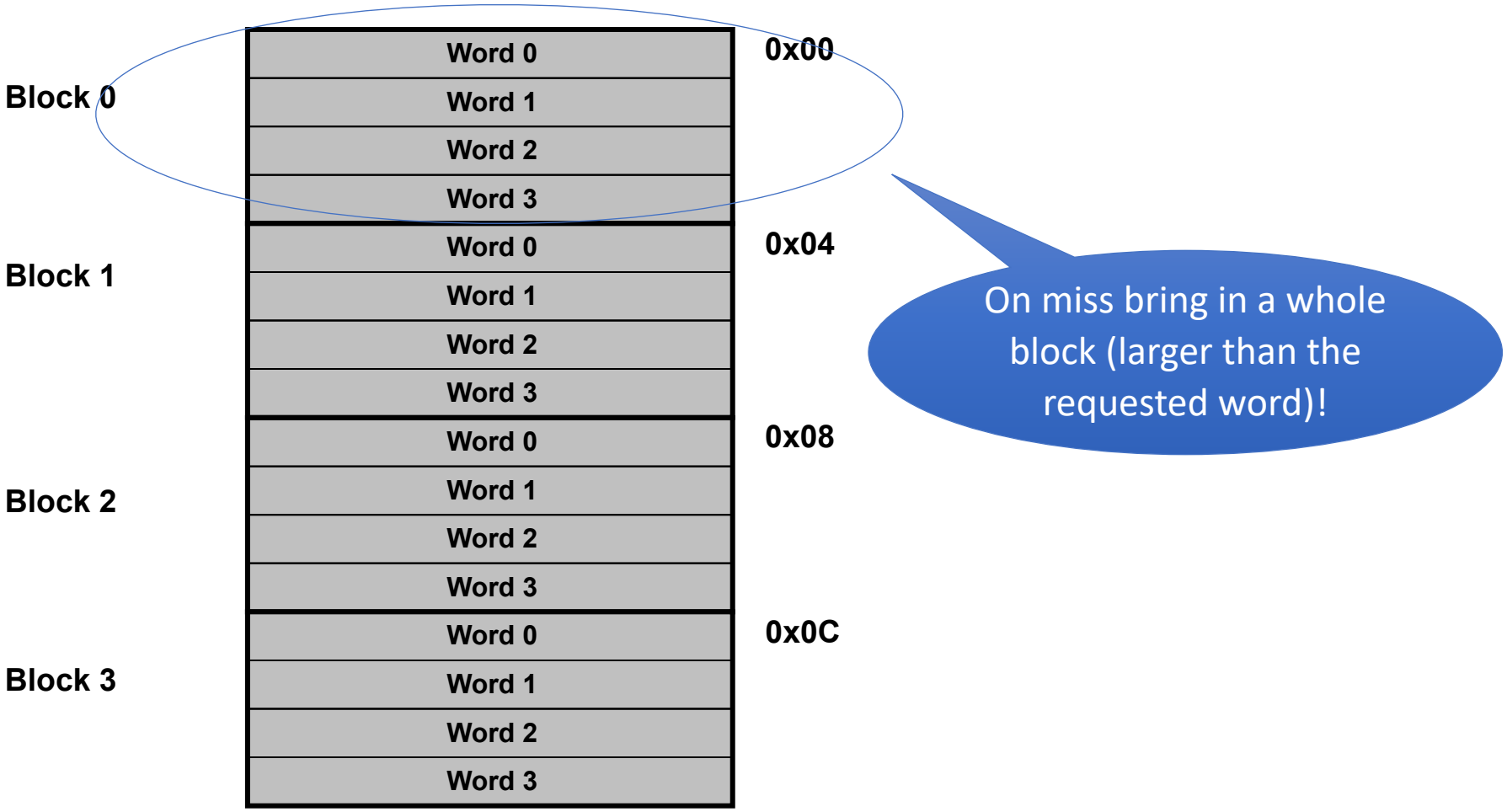
Lecture slides adapted from Bill Leahy and Charles Lively of Georgia Tech

How to improve cache efficiency

How to improve cache efficiency

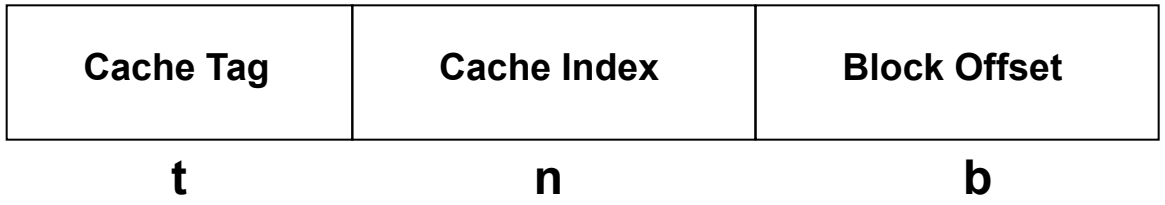
- Exploit spatial locality
  - Bring more from memory into cache at a time
- Better organization
  - Exploit working set concept

Spatial Locality



(Re)Interpreting memory address

S = Size of cache; B = Block size; L = sets in cache



$b = log_2 B$

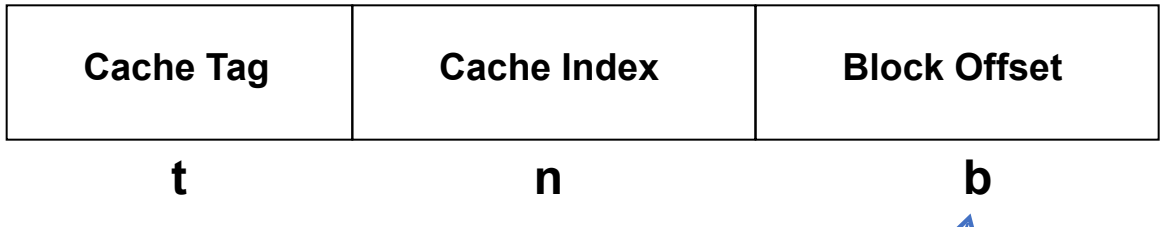
$L = S/B$

$n = log_2 L$

$t = address\ size - (b+n)$

(Re)Interpreting memory address

S = Size of cache; B = Block size; L = sets in cache



$b = log_2 B$

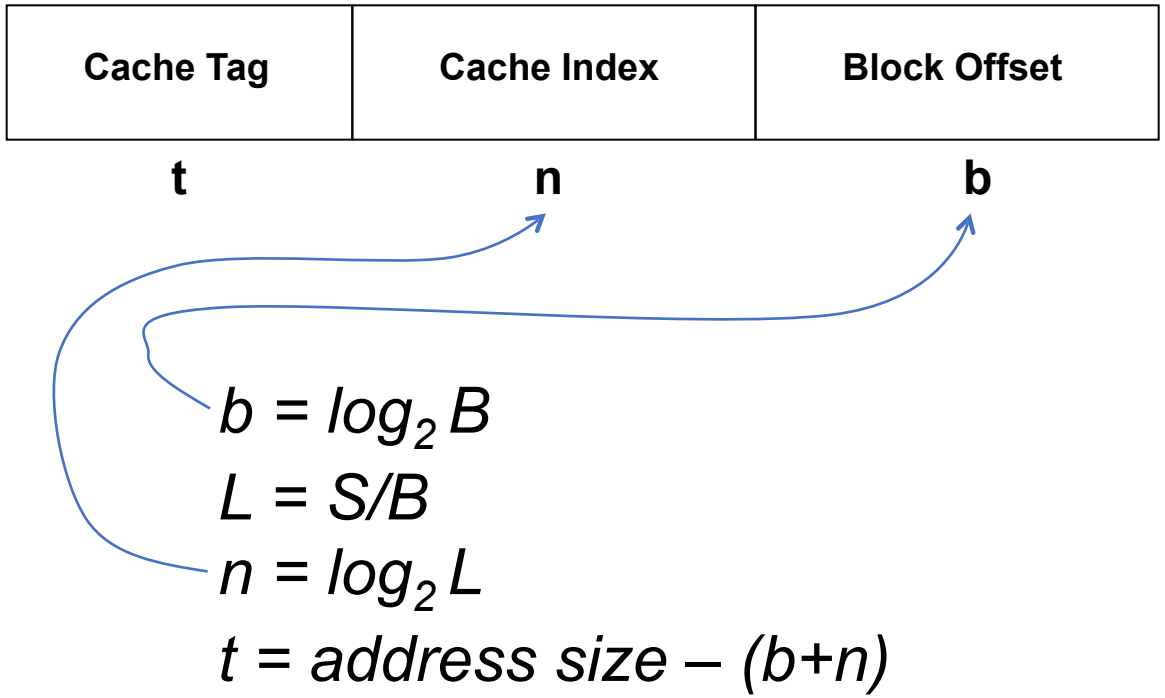
$L = S/B$

$n = log_2 L$

$t = address\ size - (b+n)$

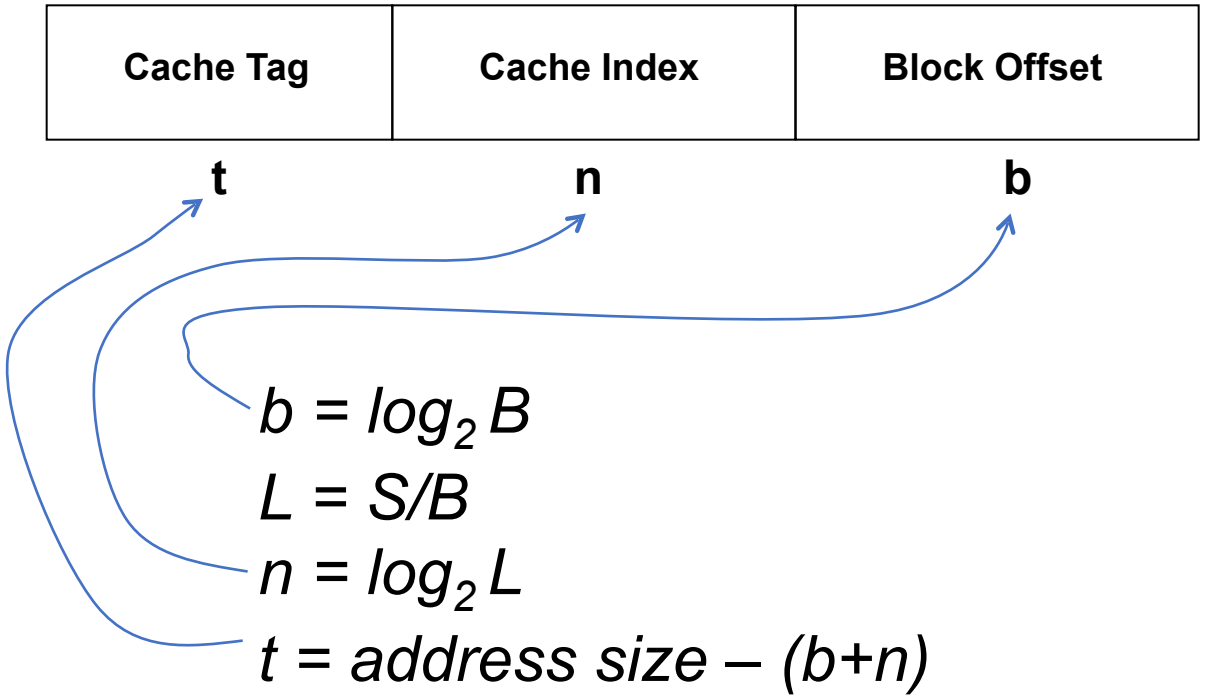
(Re)Interpreting memory address

S = Size of cache; B = Block size; L = sets in cache



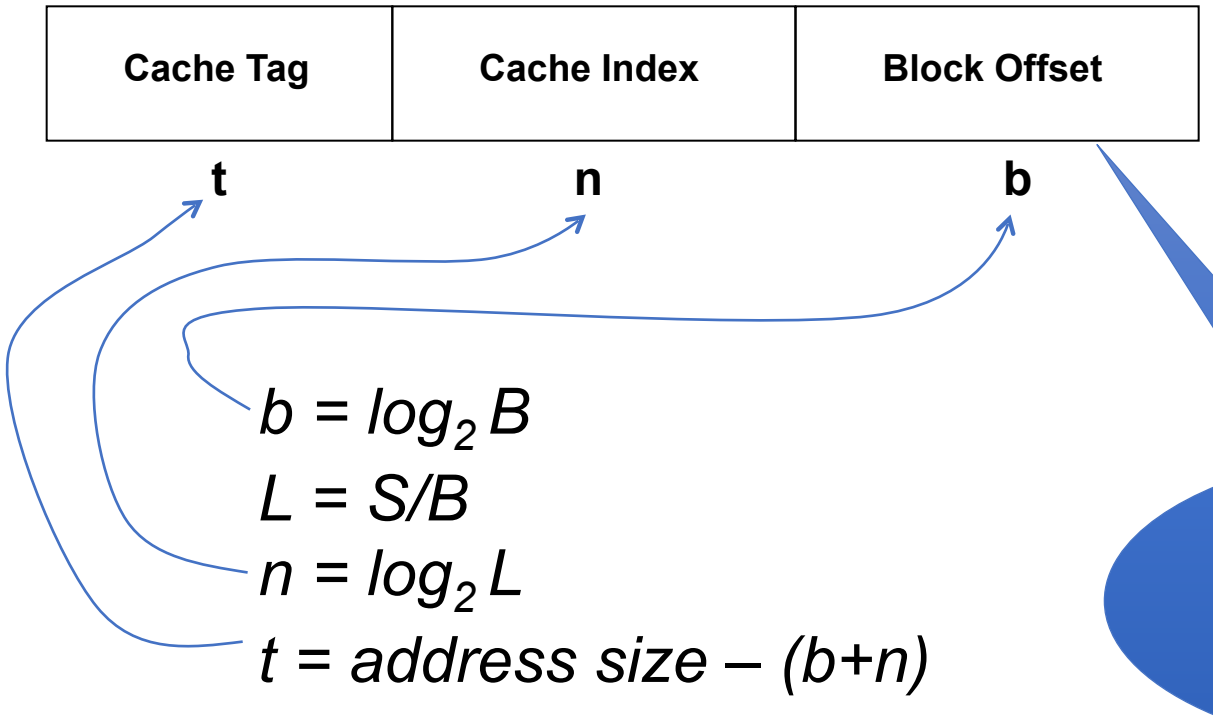
(Re)Interpreting memory address

S = Size of cache; B = Block size; L = sets in cache



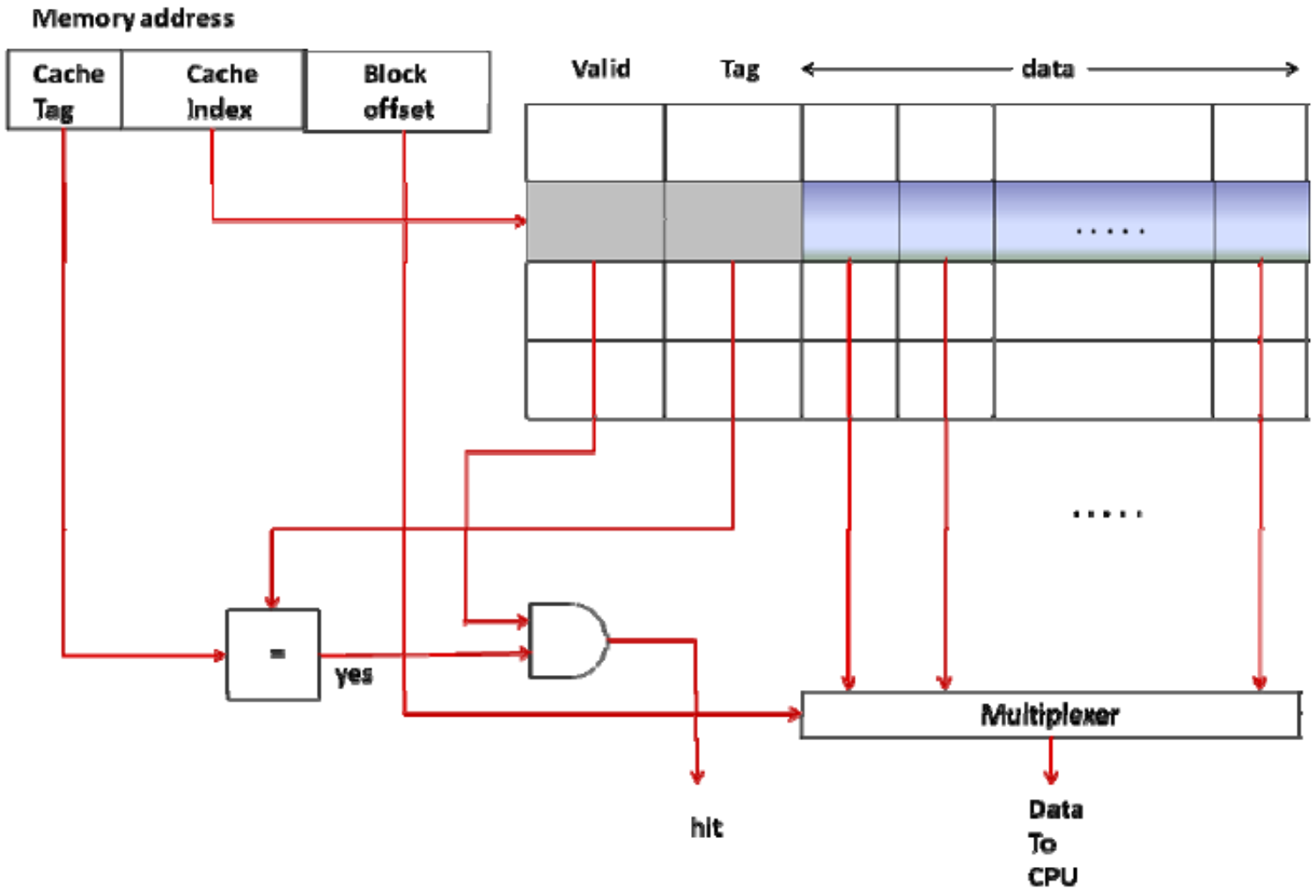
(Re)Interpreting memory address

S = Size of cache; B = Block size; L = sets in cache

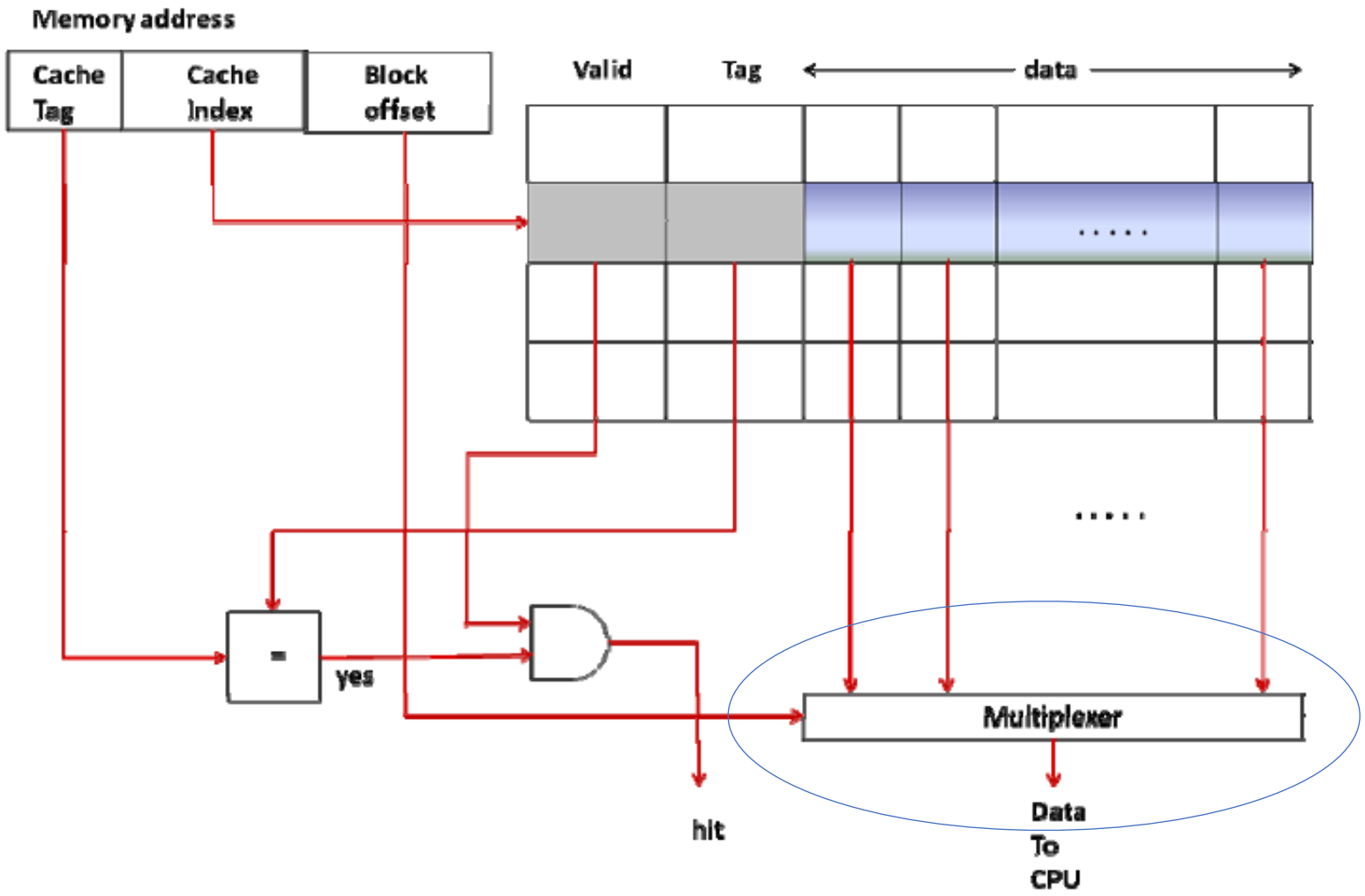


This is really just the word (or byte) number within the block

Multi-word cache organization

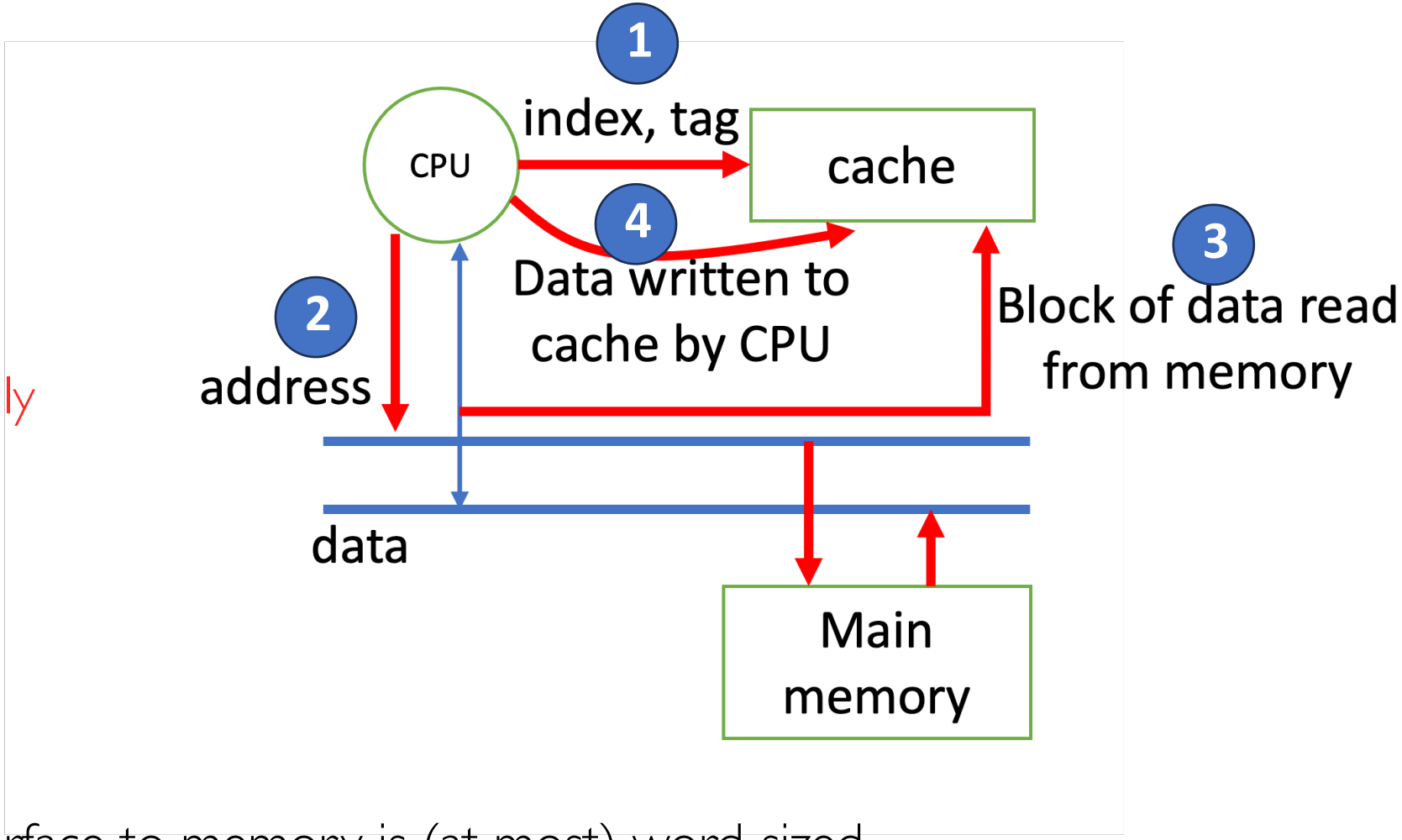


Multi-word cache organization



Write miss with multi-word cache block

The missing block is first copied from memory into the cache; only then do we update the specific word being written



Remember: the processor's interface to memory is (at most) word-sized

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Blk off

Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

Multi-word cache block example

Direct-mapped cache

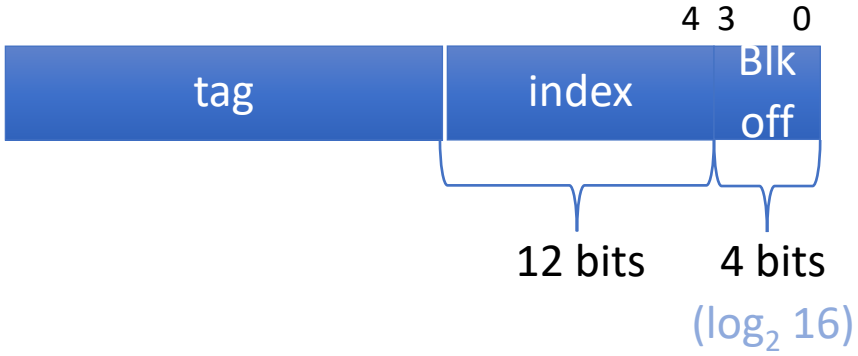
- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

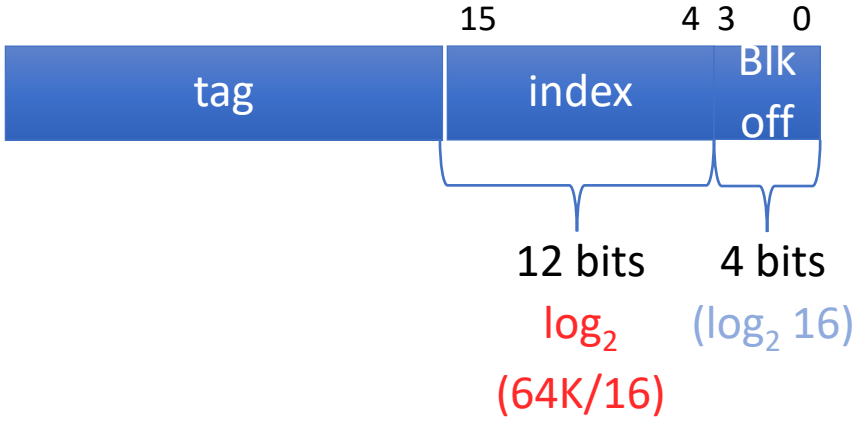


Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

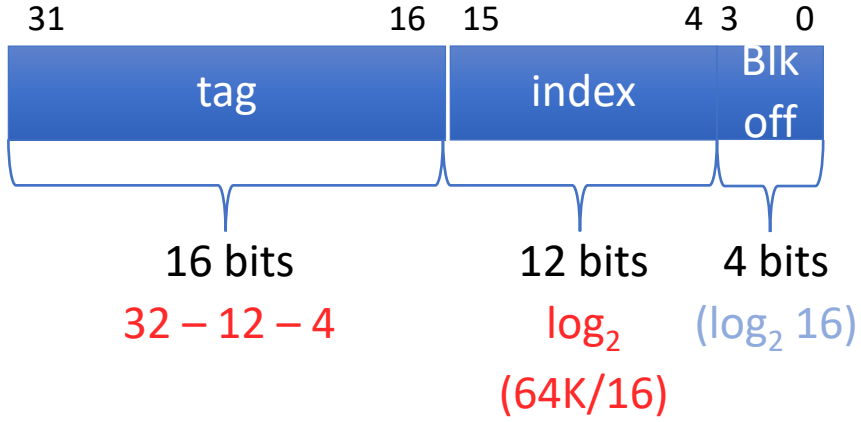


Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

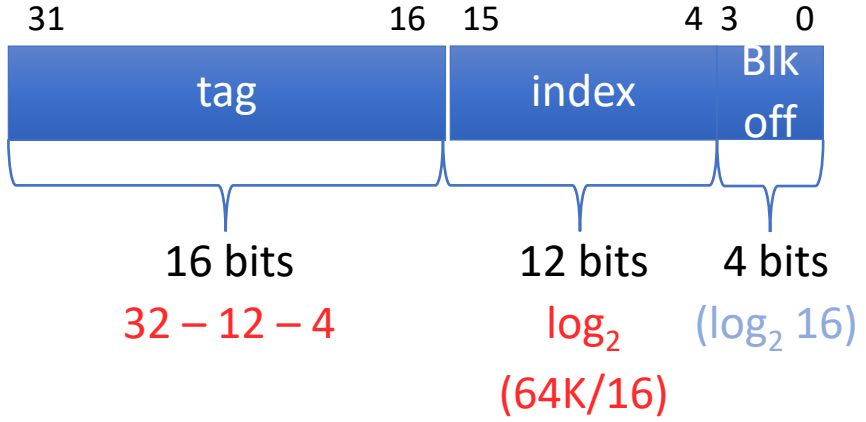


Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache



	valid	dirty	tag	word 0	word 1	word 2	word 3
0	1	0	0001				
1	1	1	0021				
2							
…							
4094							
4095							

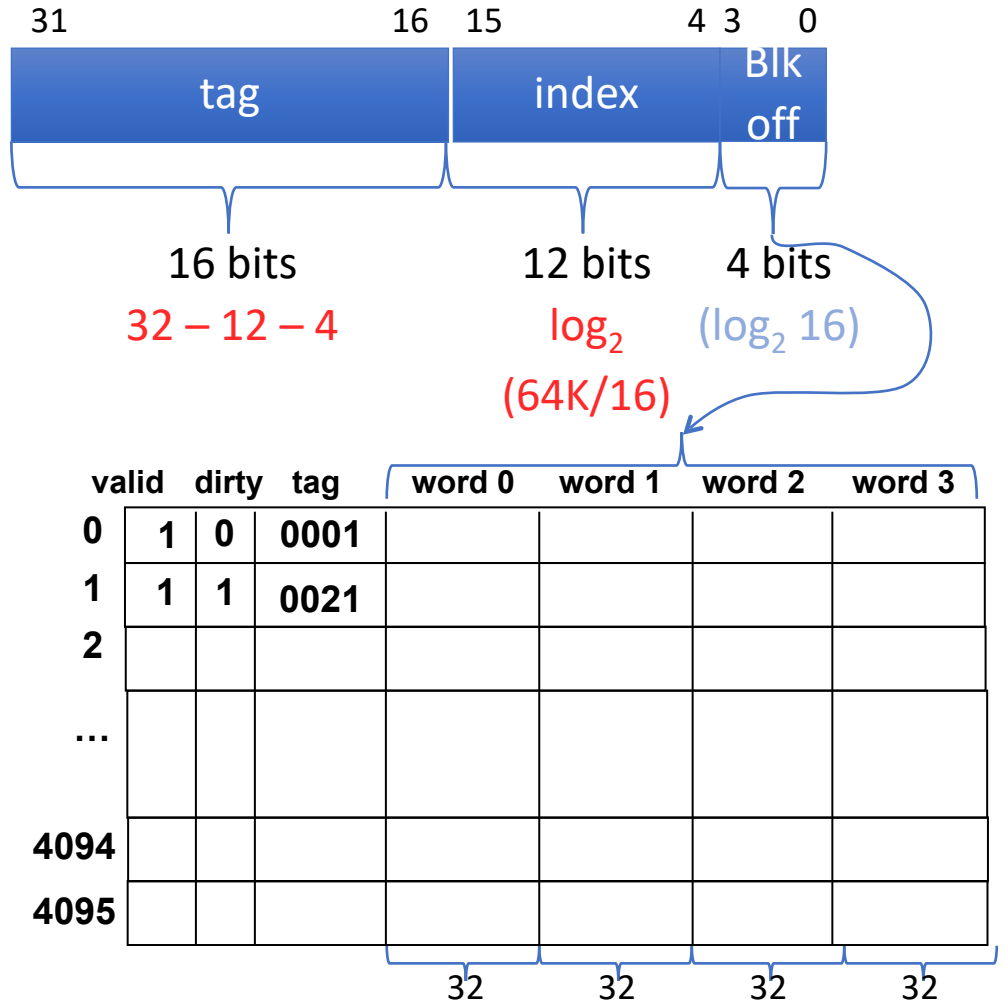
Draw cache structure

Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

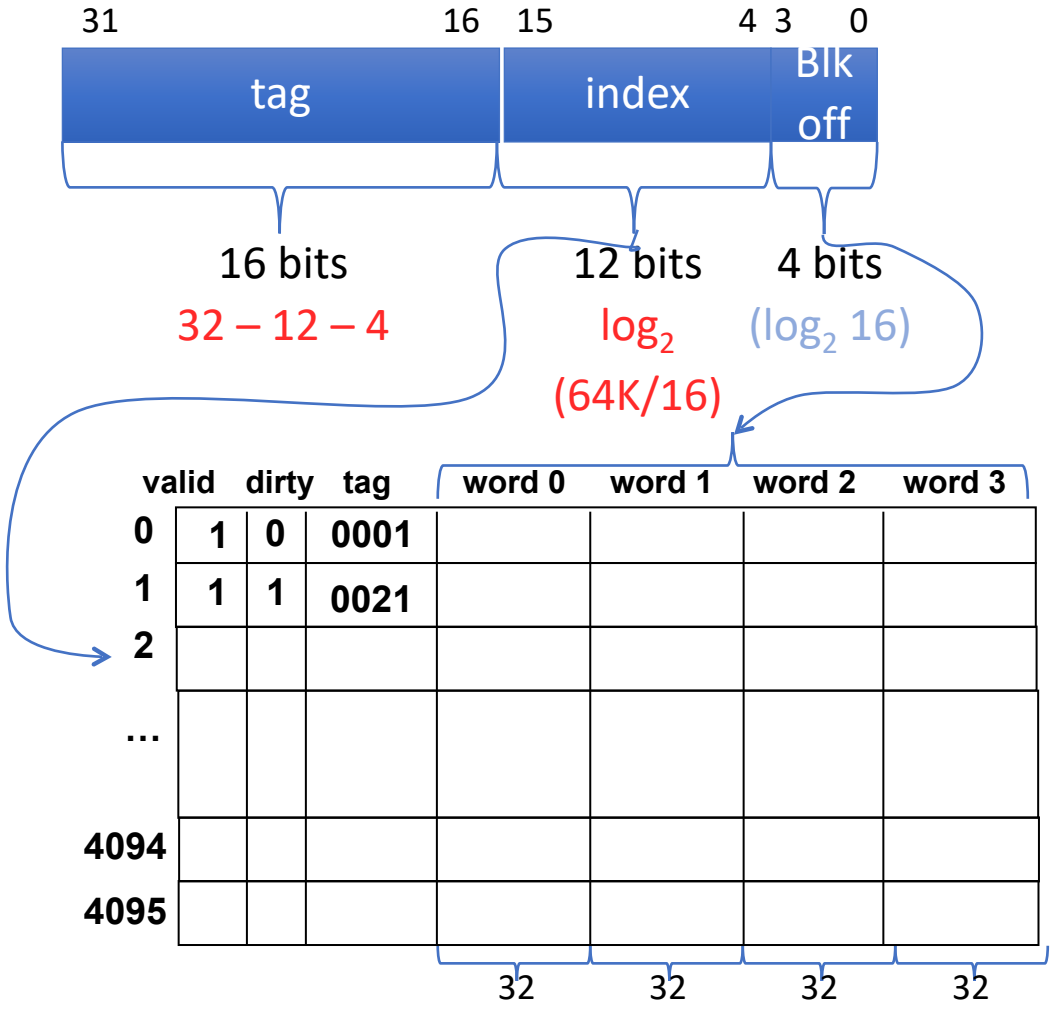


Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

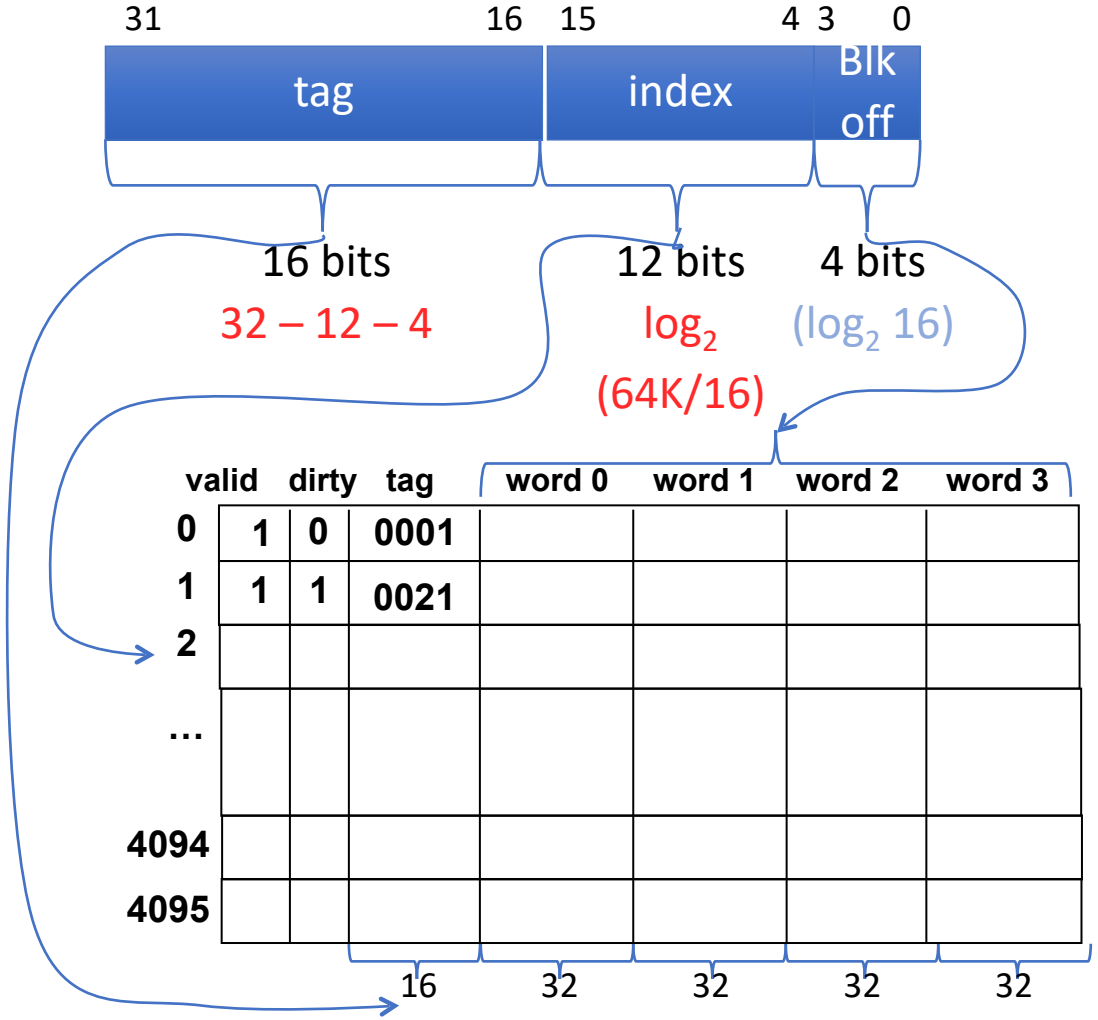


Multi-word cache block example

Direct-mapped cache

- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure

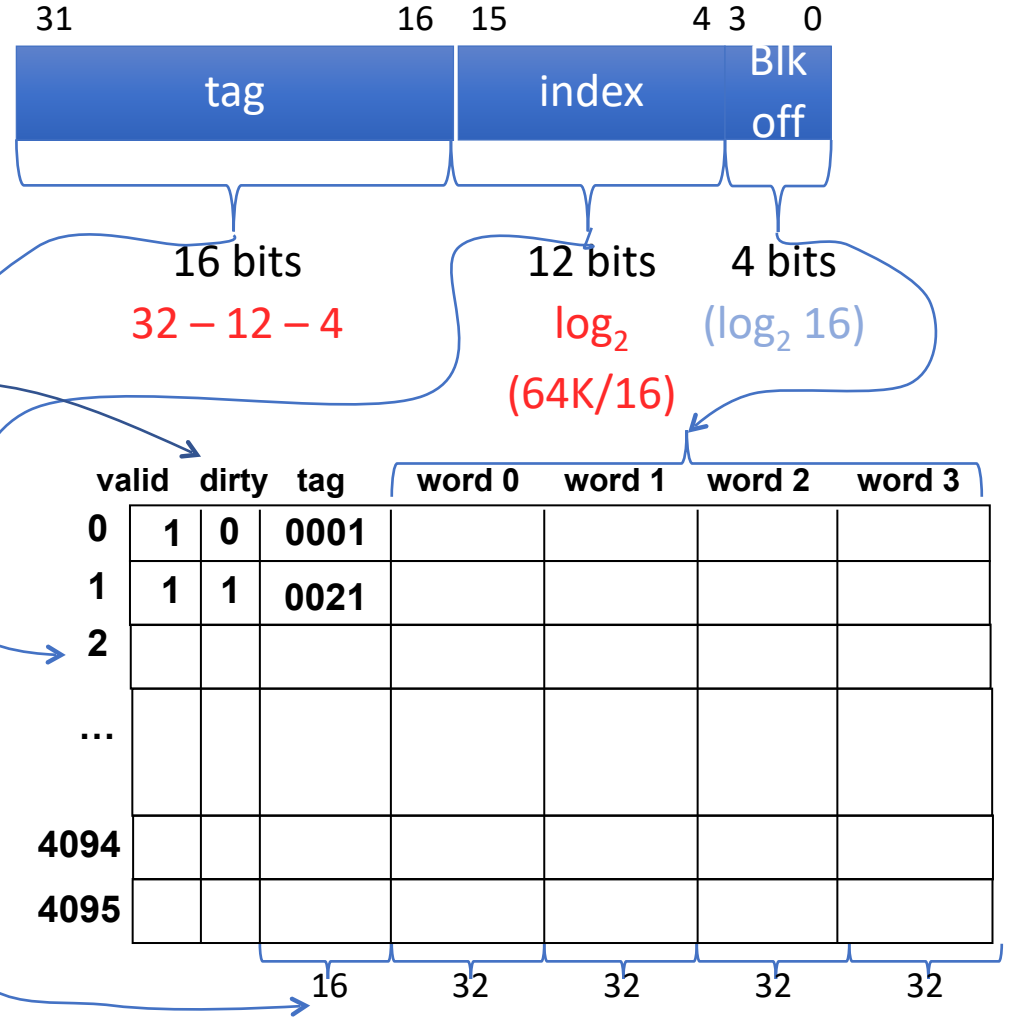


Multi-word cache block example

Direct-mapped cache

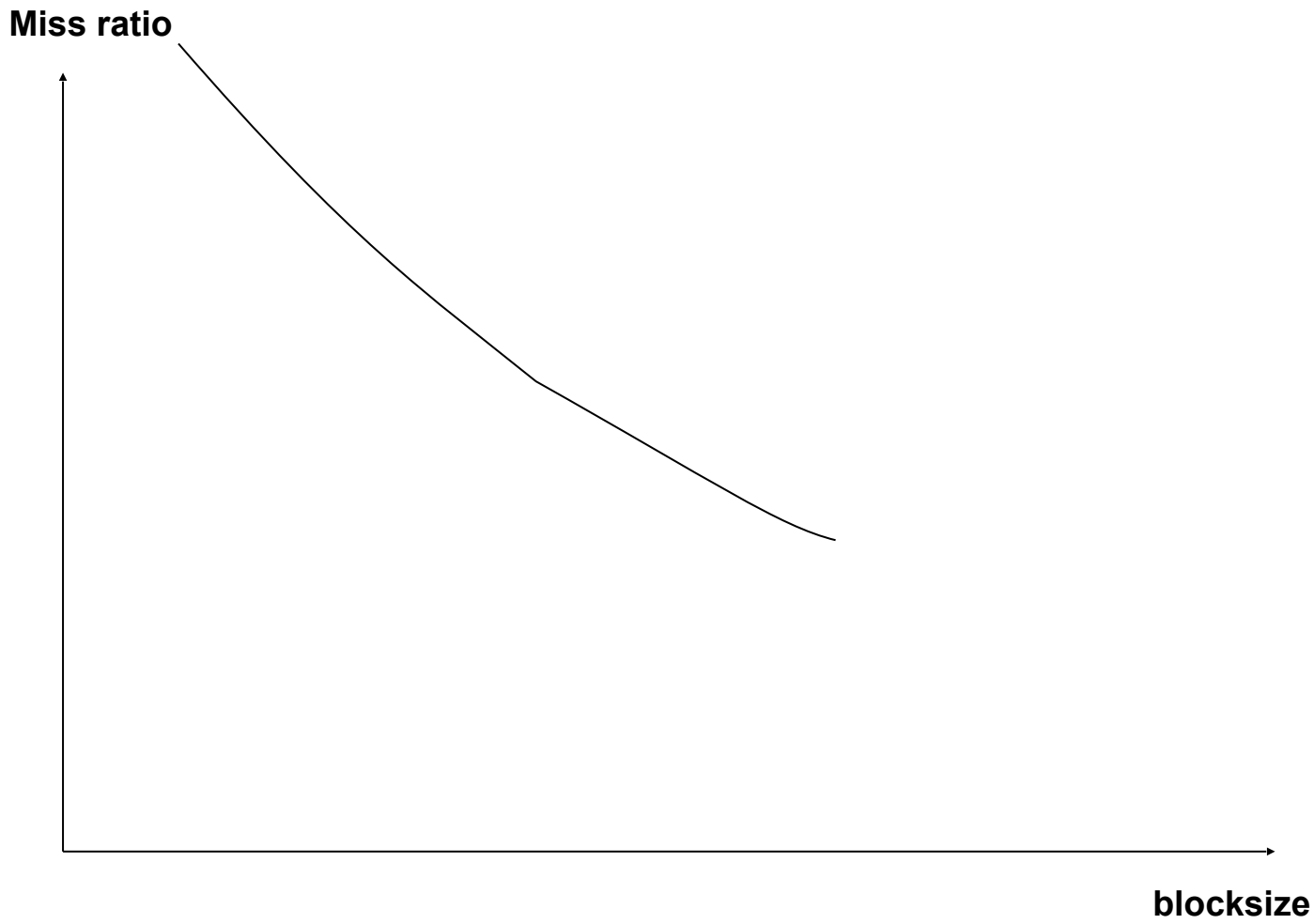
- 32-bit byte-addressable memory address
- Each memory word contains 4 bytes
- Block size = 4 words (16 bytes)
  - A memory access brings in a block
- 64K byte write-back cache

Draw cache structure



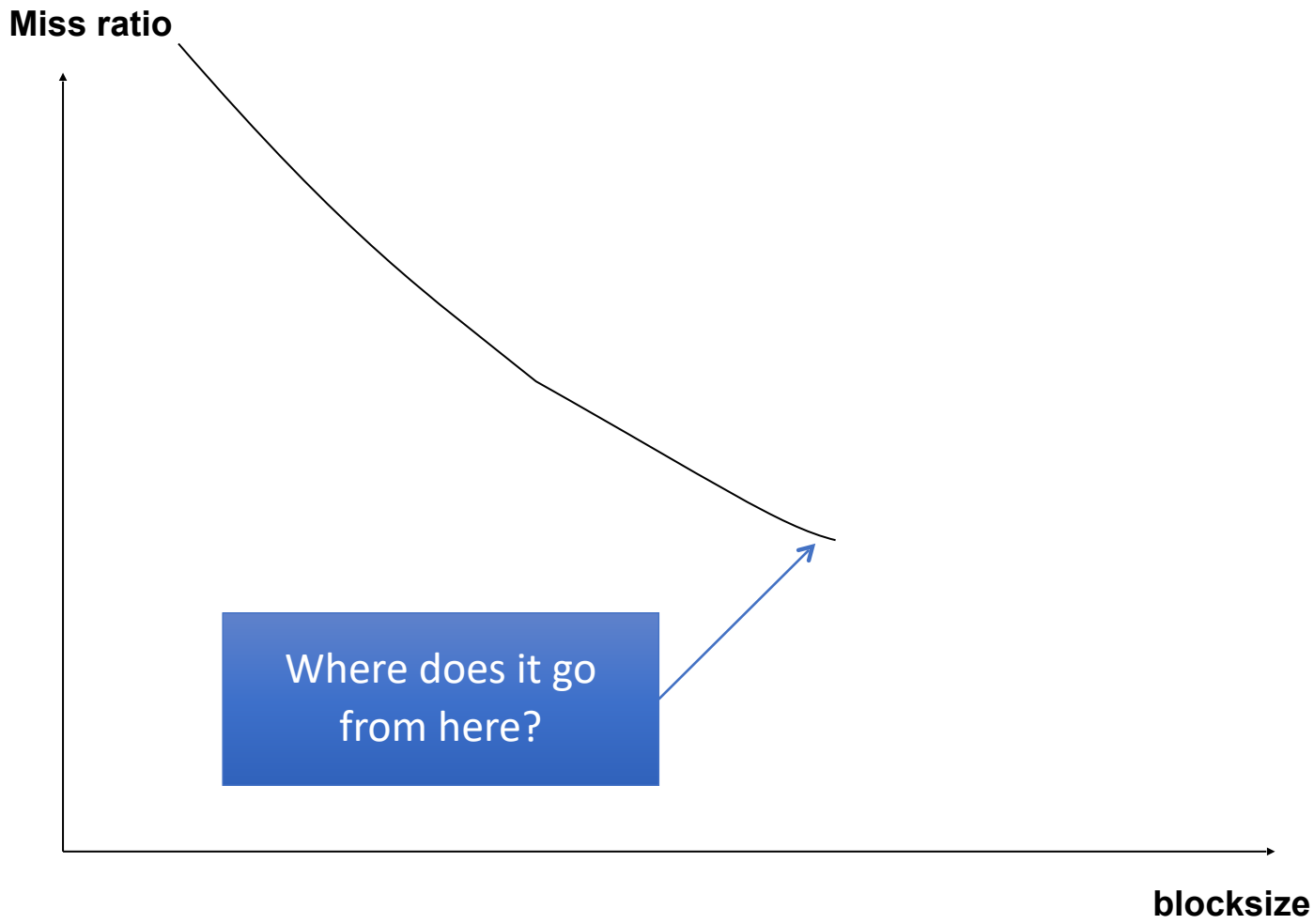
Increased block size?

- Exploits more spatial locality
- Reduces miss ratio



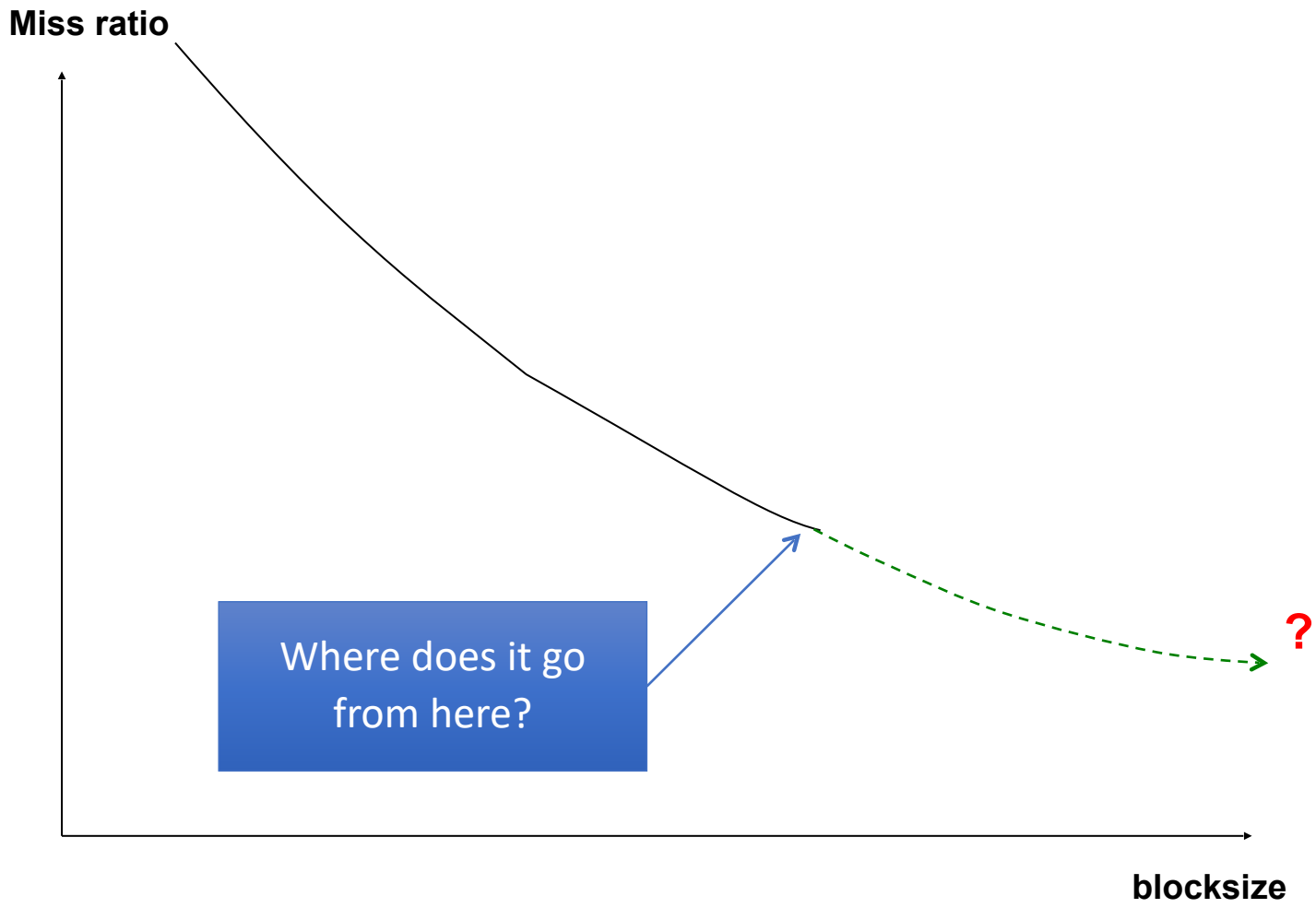
Increased block size?

- Exploits more spatial locality
- Reduces miss ratio



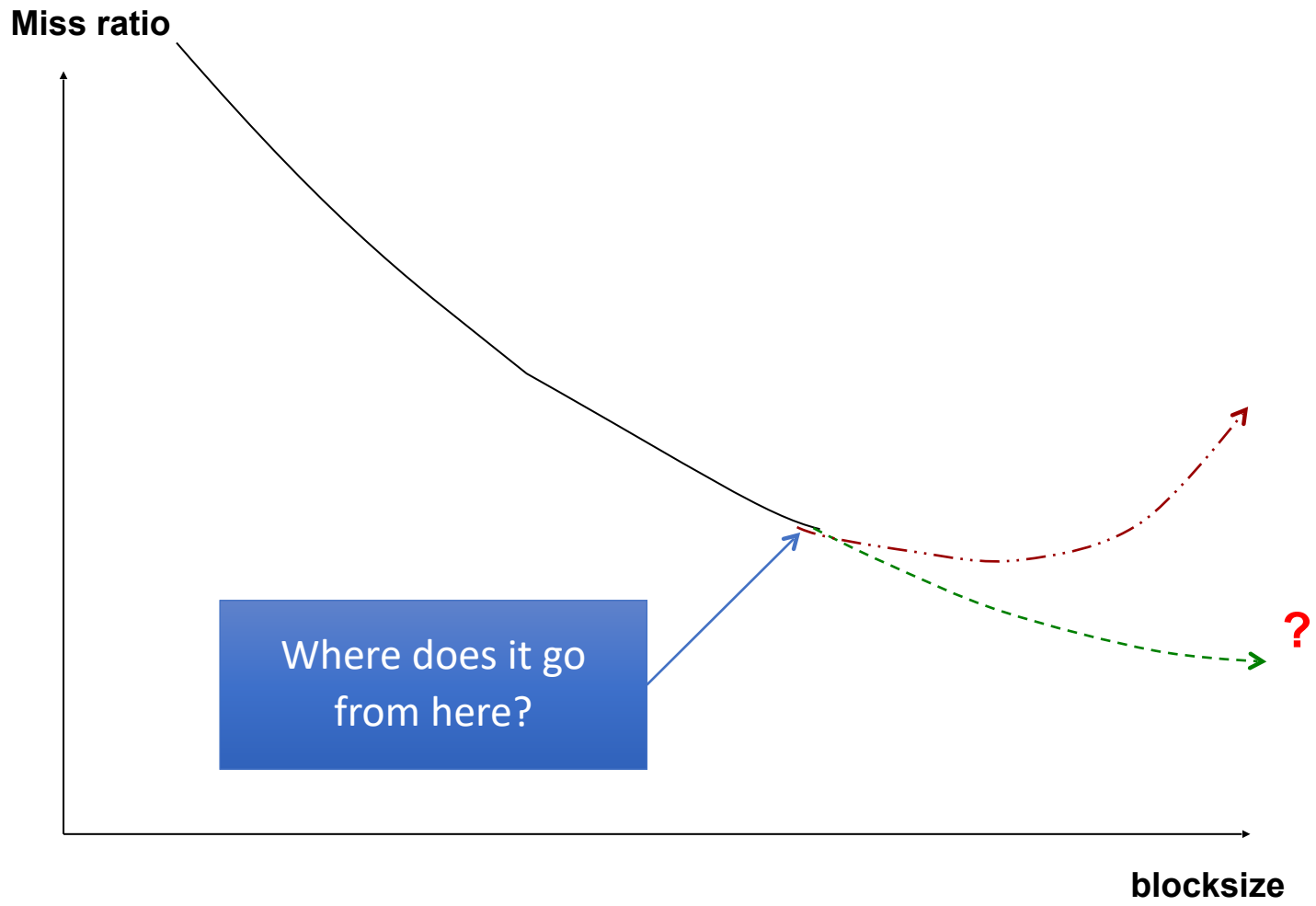
Increased block size?

- Exploits more spatial locality
- Reduces miss ratio



Increased block size?

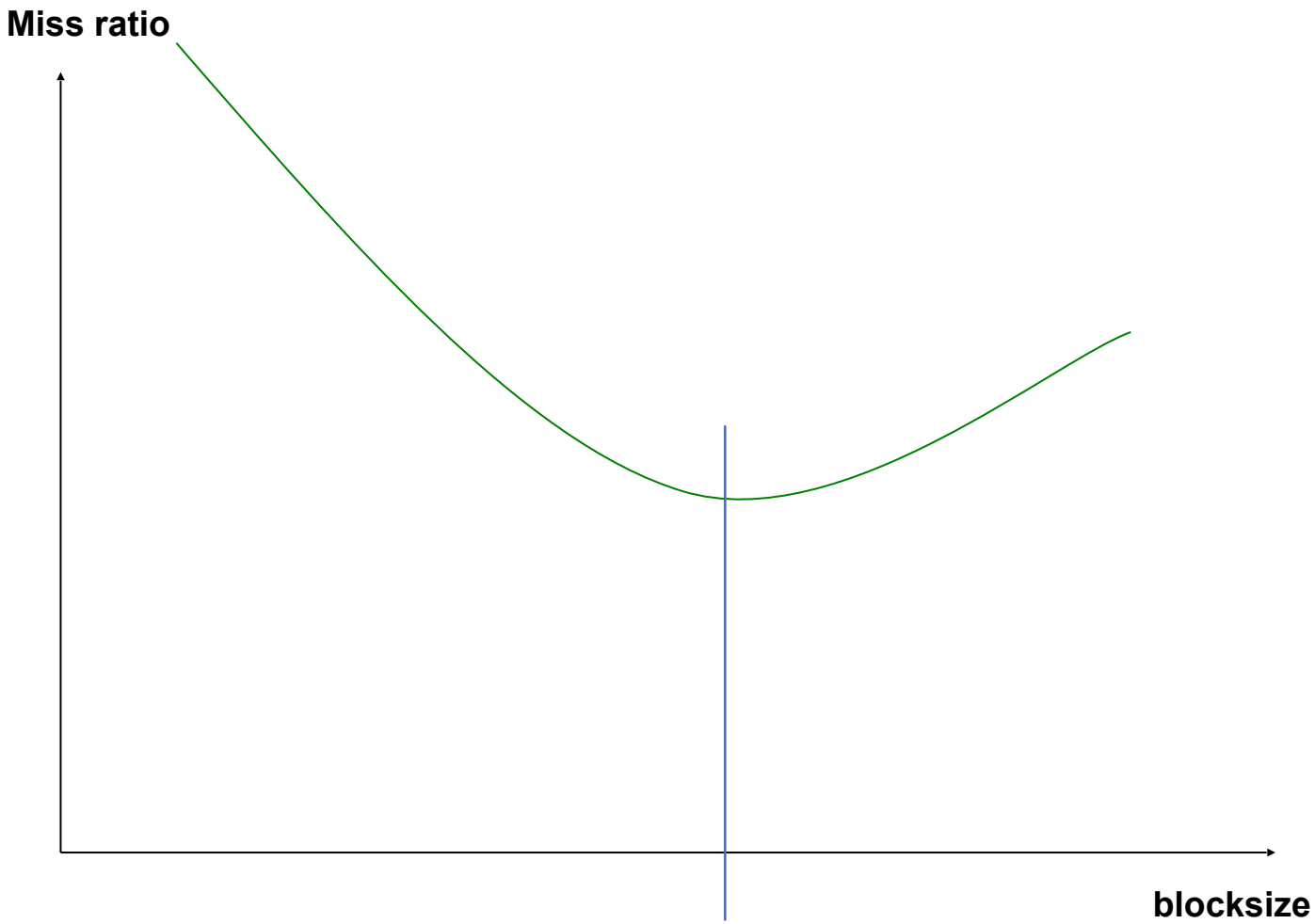
- Exploits more spatial locality
- Reduces miss ratio



Increased block size?

There is a point where things get worse

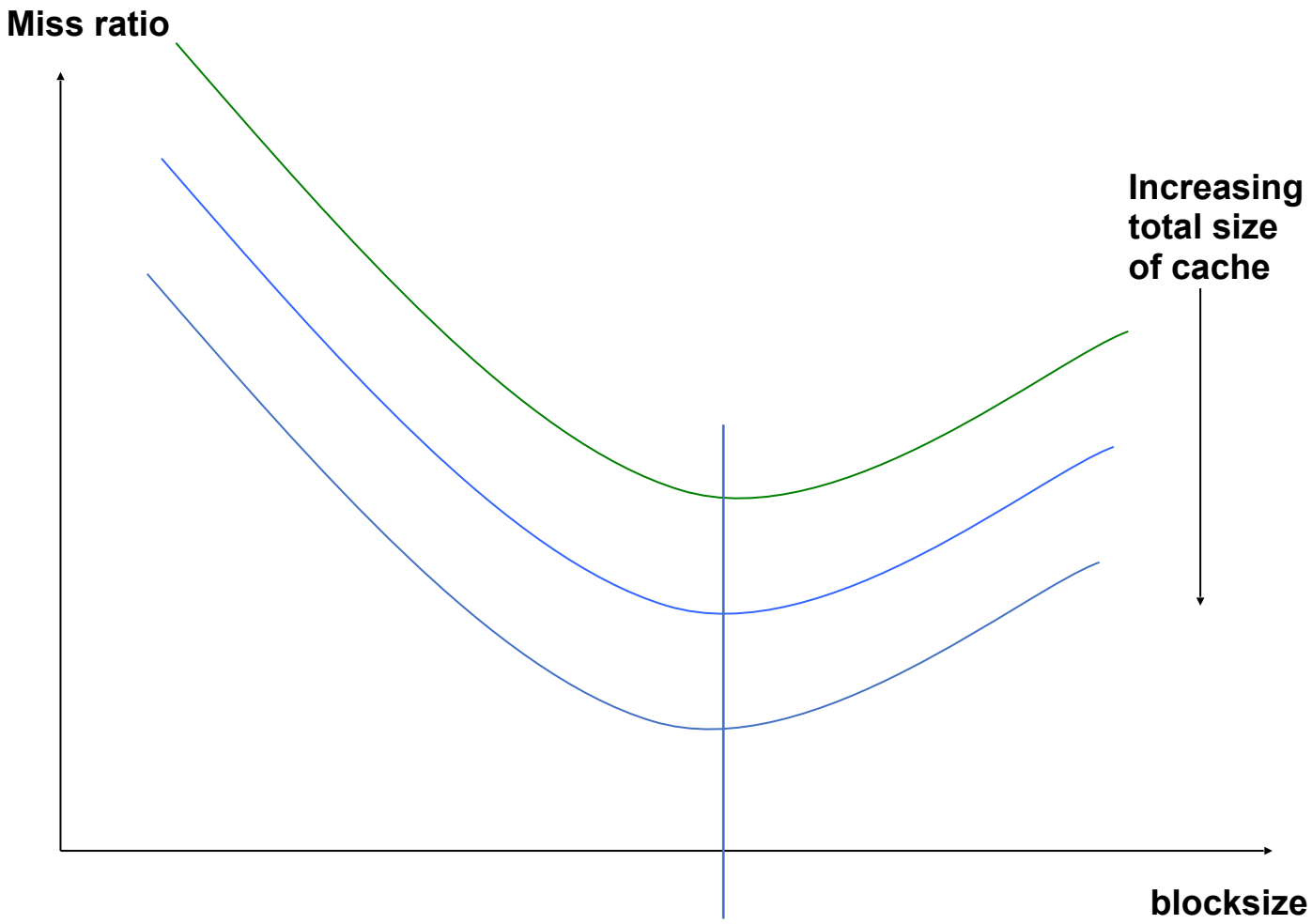
1. We reduce effective cache capacity by bringing in too much data (beyond useful spatial locality)
2. When the working set changes, larger blocks have to be fetched
   - Memory can only transfer so fast and it can become the bottleneck



Increased block size?

There is a point where things get worse

1. We reduce effective cache capacity by bringing in too much data (beyond useful spatial locality)
2. When the working set changes, larger blocks have to be fetched
   - Memory can only transfer so fast and it can become the bottleneck

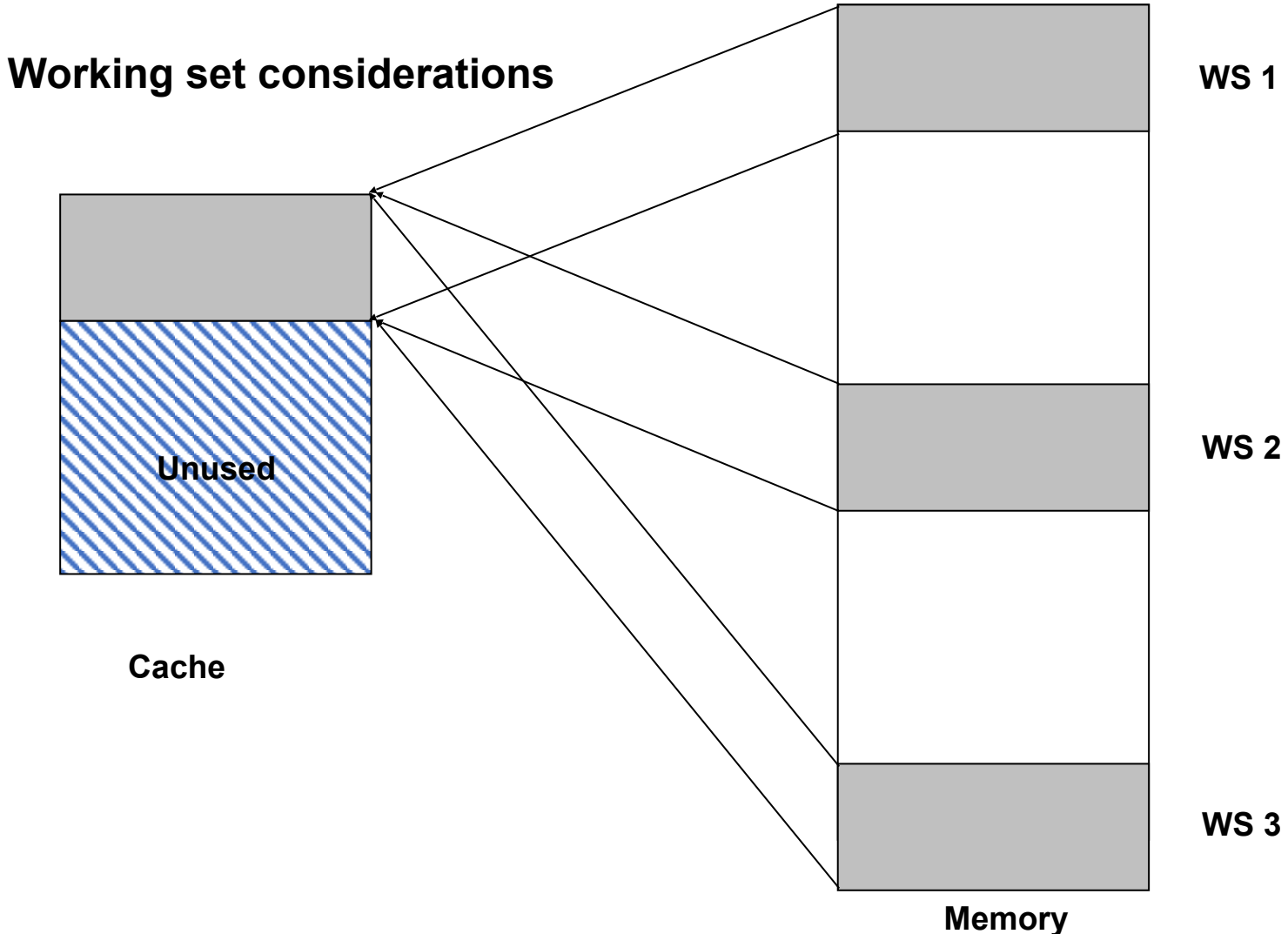


How to improve cache efficiency

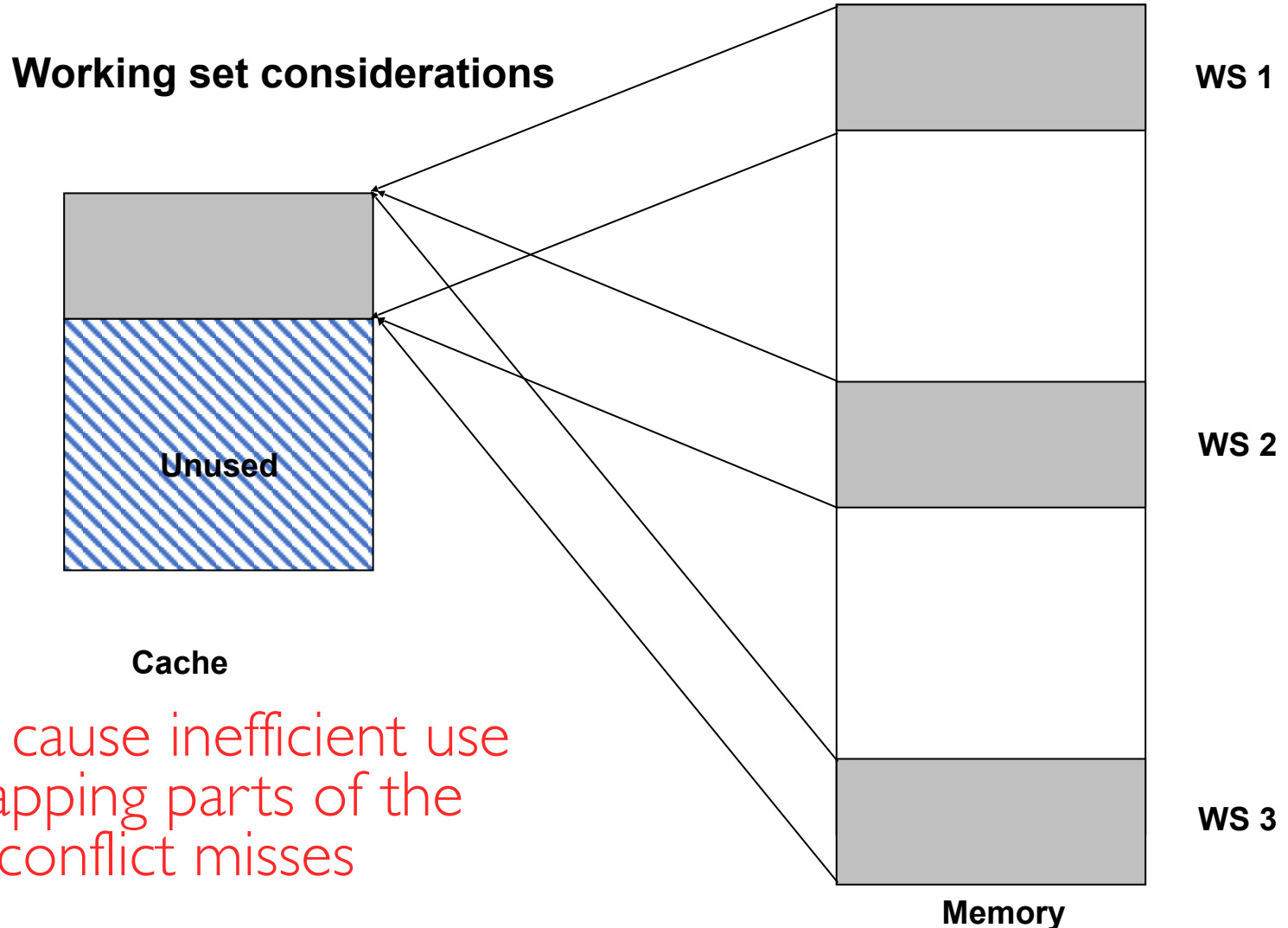
- Exploit spatial locality
  - Bring more from memory into cache at a time

- Better organization
  - Exploit working set concept

Working set considerations



Working set considerations



Direct mapping can cause inefficient use of cache with overlapping parts of the working set due to conflict misses

What would be best?

- Allow any memory block to be brought into any cache block

What would be best?

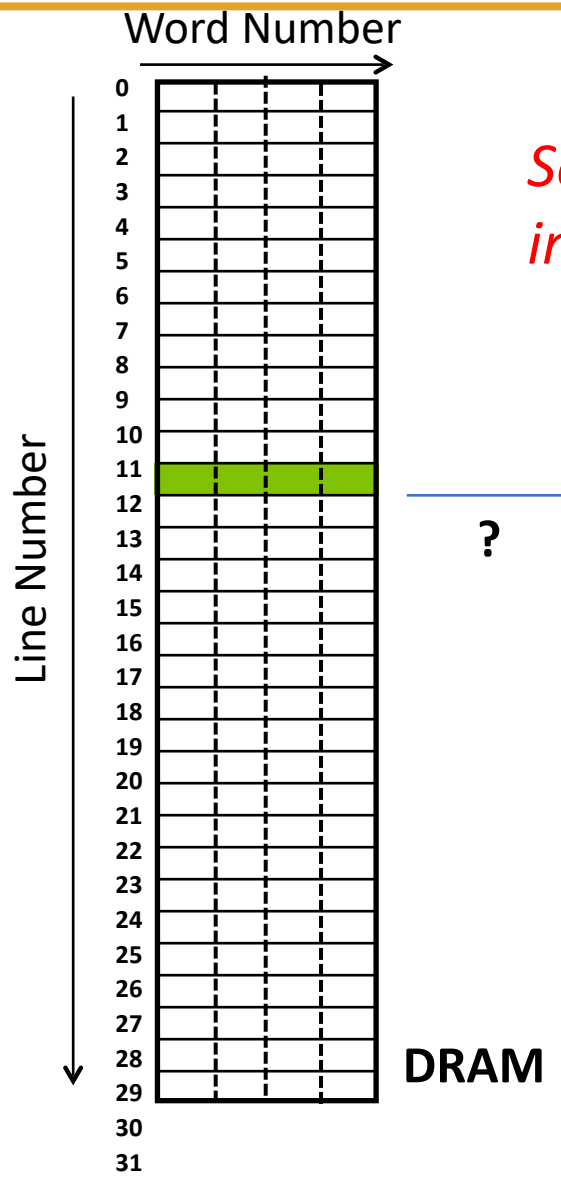
- Allow any memory block to be brought into any cache block
- This is similar to the ability of mapping any virtual page to any available physical page frame

What would be best?

- Allow any memory block to be brought into any cache block
- This is similar to the ability of mapping any virtual page to any available physical page frame

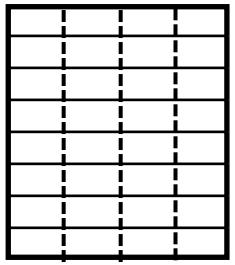
➔ Fully associative mapping

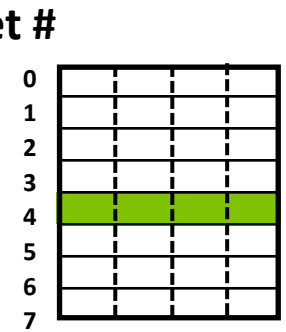
Cache Placement



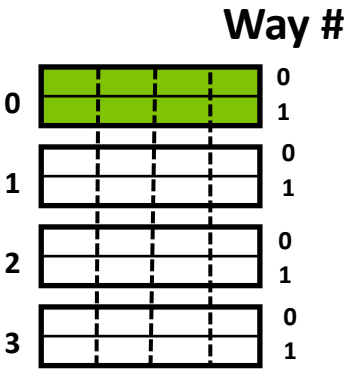
Searching is easy but inefficient use of space

?



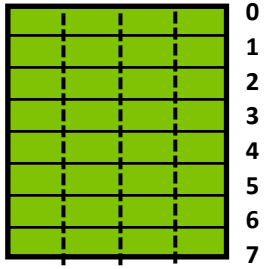


Direct Mapped
one location



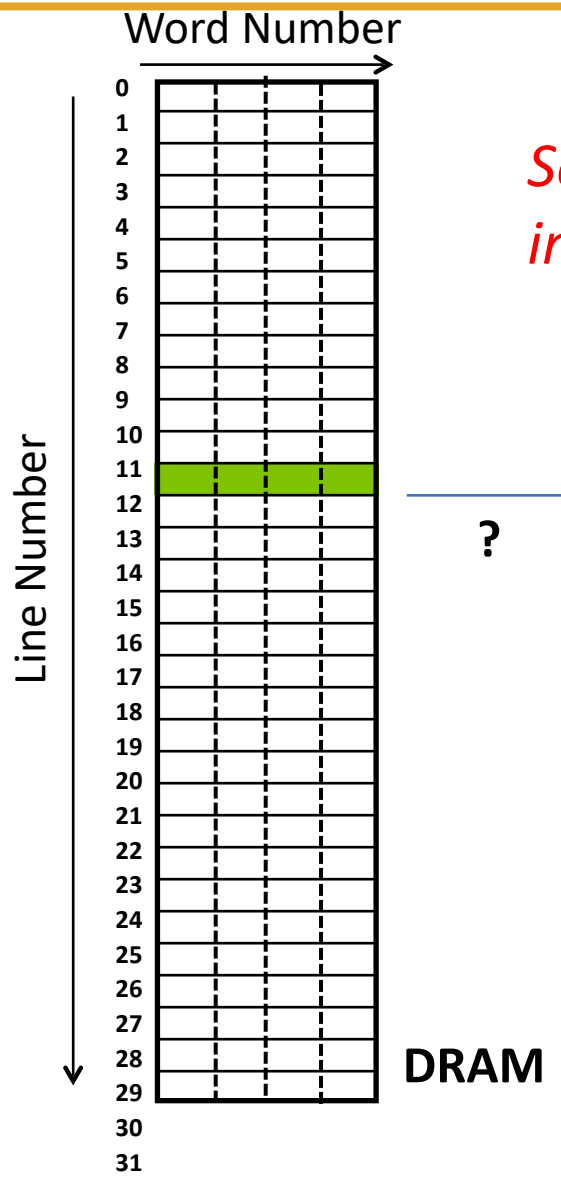
(2-way) Set Associative
2 locations

Searching is more expensive



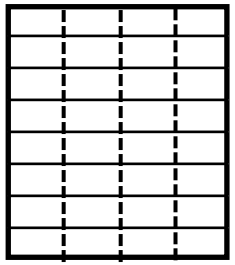
Fully Associative
anywhere

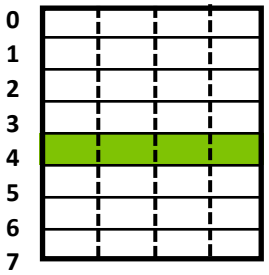
Cache Placement



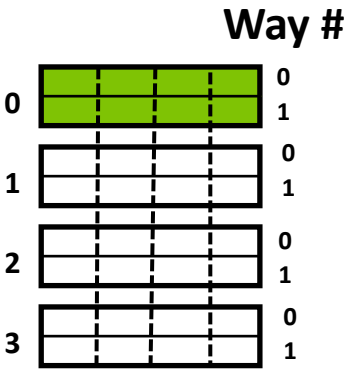
Searching is easy but inefficient use of space

?



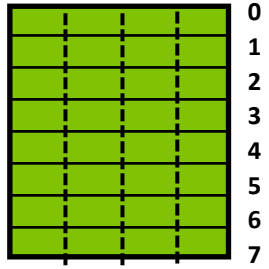


Direct Mapped
one location



(2-way) Set Associative
2 locations

Searching is more expensive



Fully Associative
anywhere

Address interpretation in FA cache

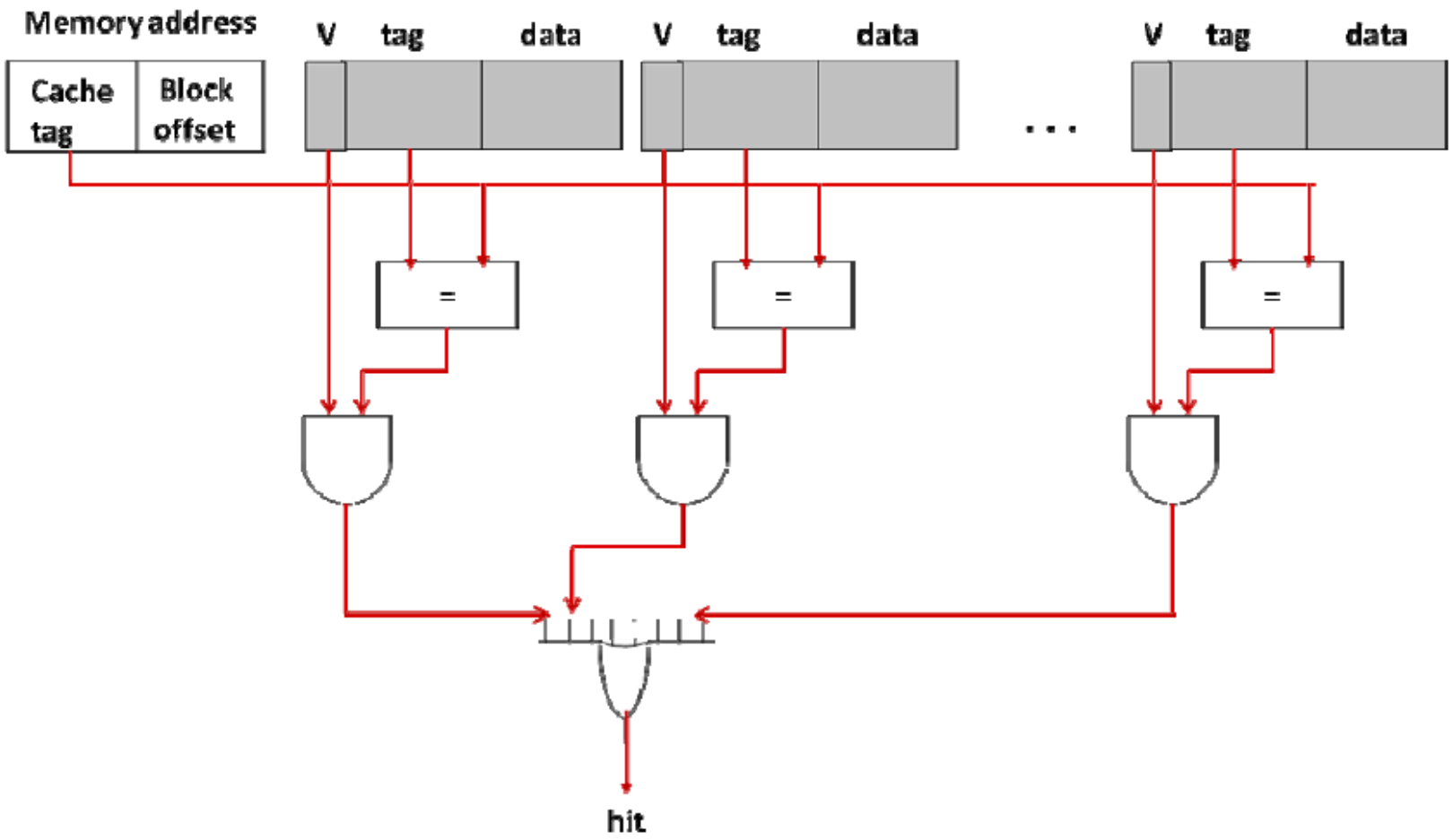
Address interpretation in FA cache

- No splitting memory addresses into "index" and "tag"

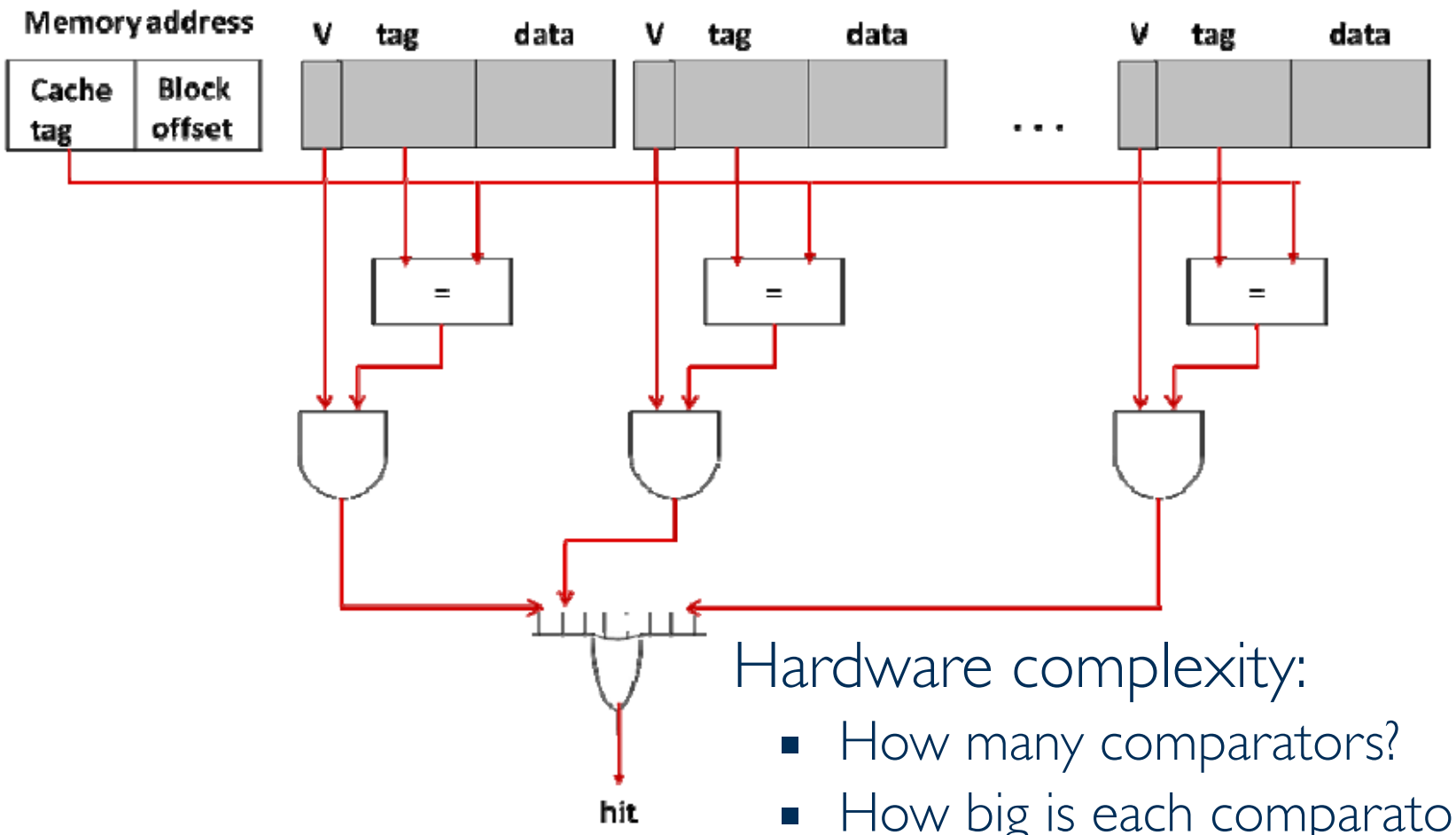
Address interpretation in FA cache

- No splitting memory addresses into "index" and "tag"
- It all becomes tag!

Fully associative cache circuitry



Fully associative cache circuitry



Hardware complexity:

- How many comparators?
- How big is each comparator?

Cache organizations

- Fully associative cache ➔
  - Too much hardware complexity
  - Most flexible

- Direct mapped cache ➔
  - Least hardware complexity
  - Least flexible

Cache organizations

- Fully associative cache ➔
  - Too much hardware complexity
  - Most flexible

- Direct mapped cache ➔
  - Least hardware complexity
  - Least flexible

- Can we do better? Is there a compromise?

Cache organizations

- Fully associative cache ➔
  - Too much hardware complexity
  - Most flexible

- Direct mapped cache ➔
  - Least hardware complexity
  - Least flexible

- Can we do better? Is there a compromise?
- Yes! It's called a set-associative cache

Cache organizations

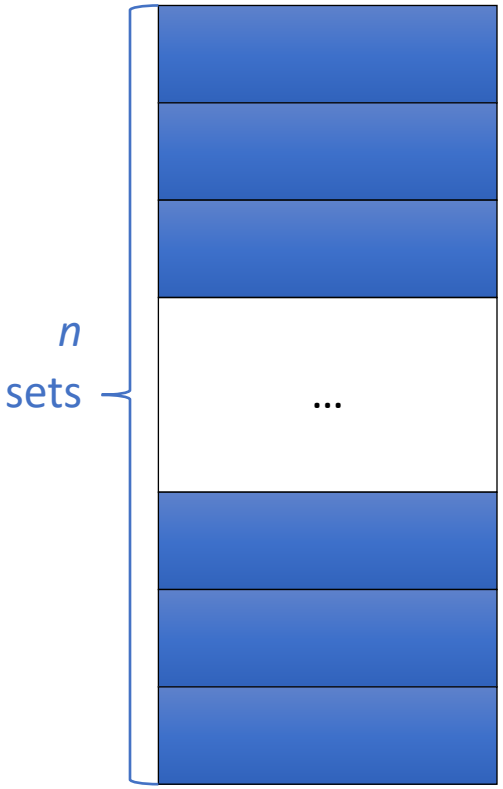
- Fully associative cache ➔
  - Too much hardware complexity
  - Most flexible

- Direct mapped cache ➔
  - Least hardware complexity
  - Least flexible

- Can we do better? Is there a compromise?
- Yes! It's called a set-associative cache
- Direct-mapped and fully-associative caches are special cases of a set-associative cache on opposite ends of the spectrum!

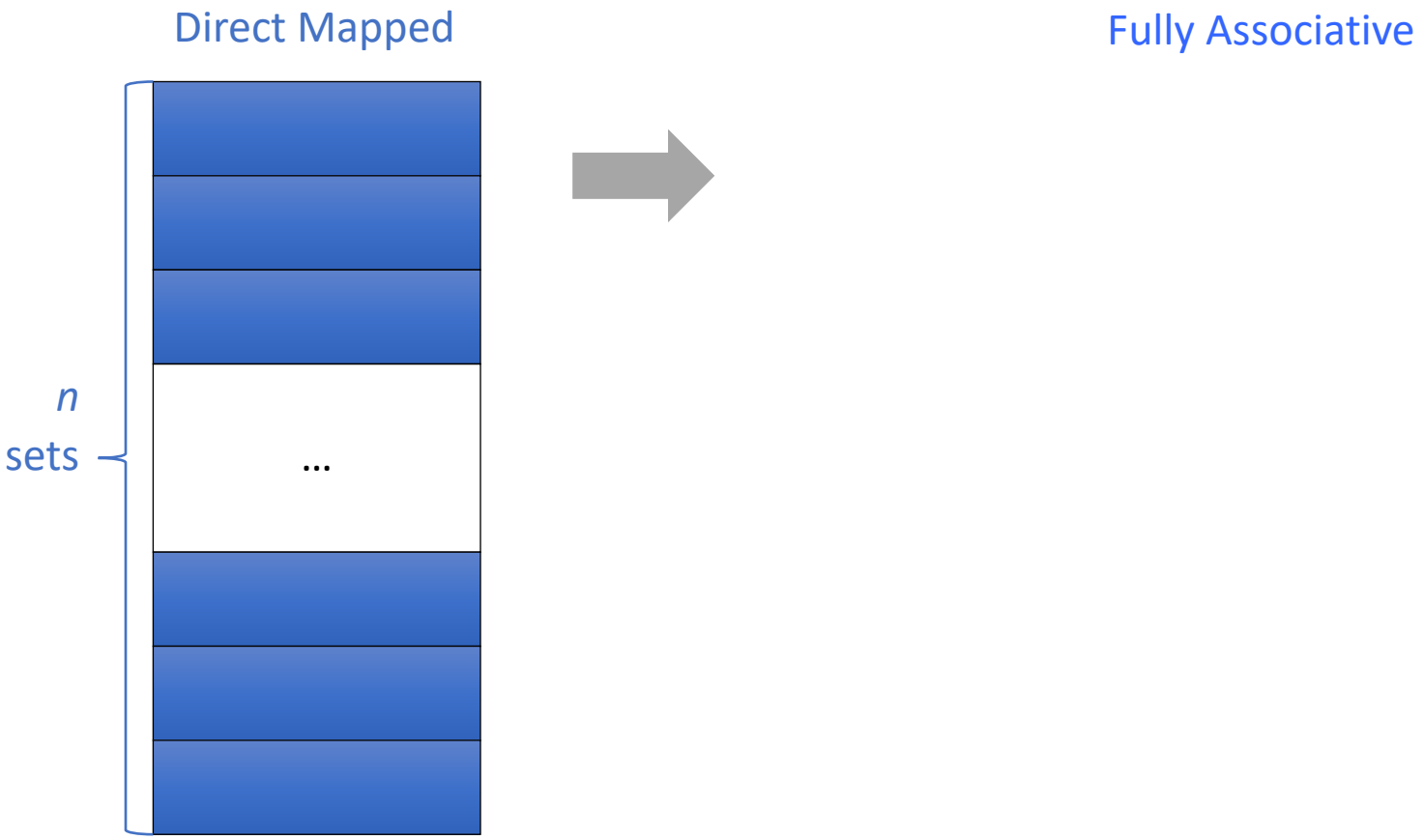
Generalization?

Direct Mapped

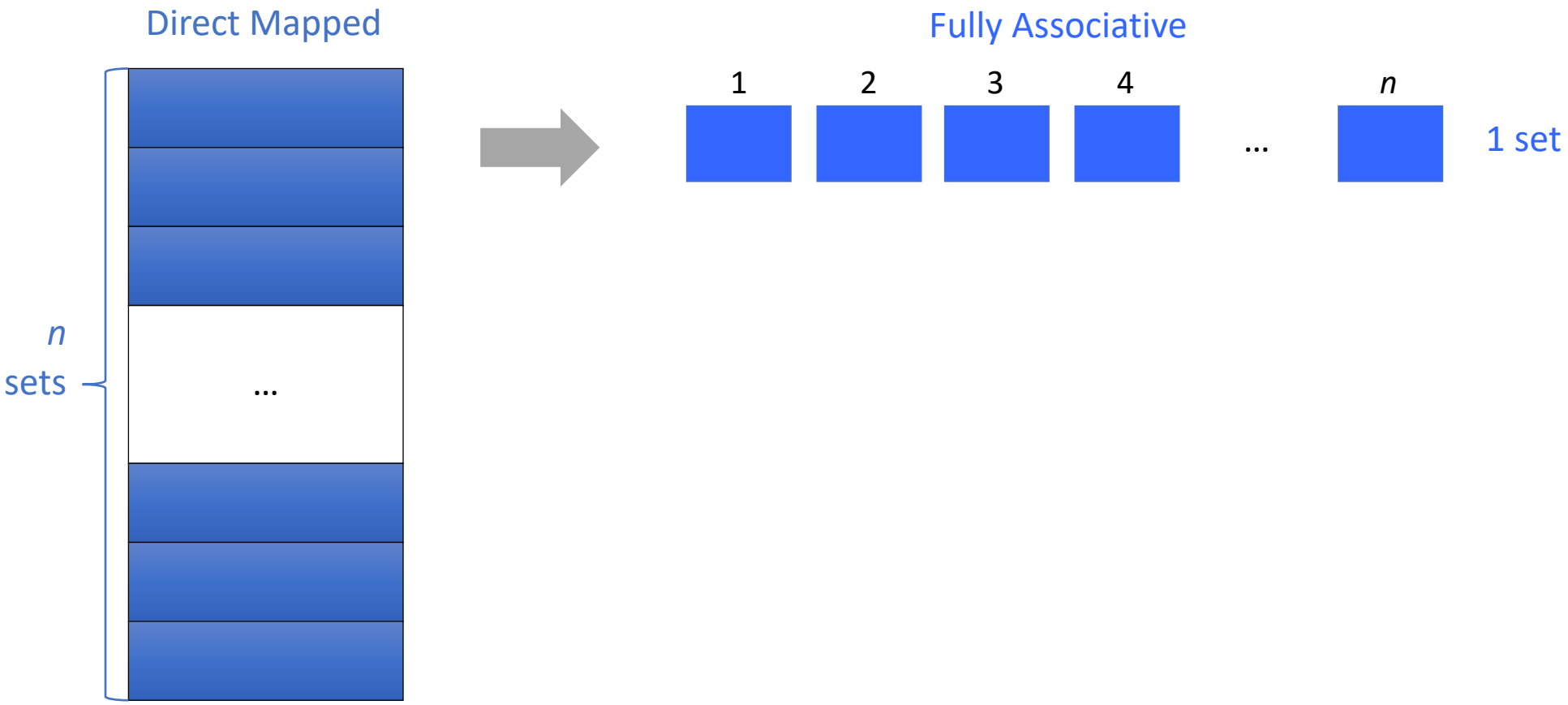


Fully Associative

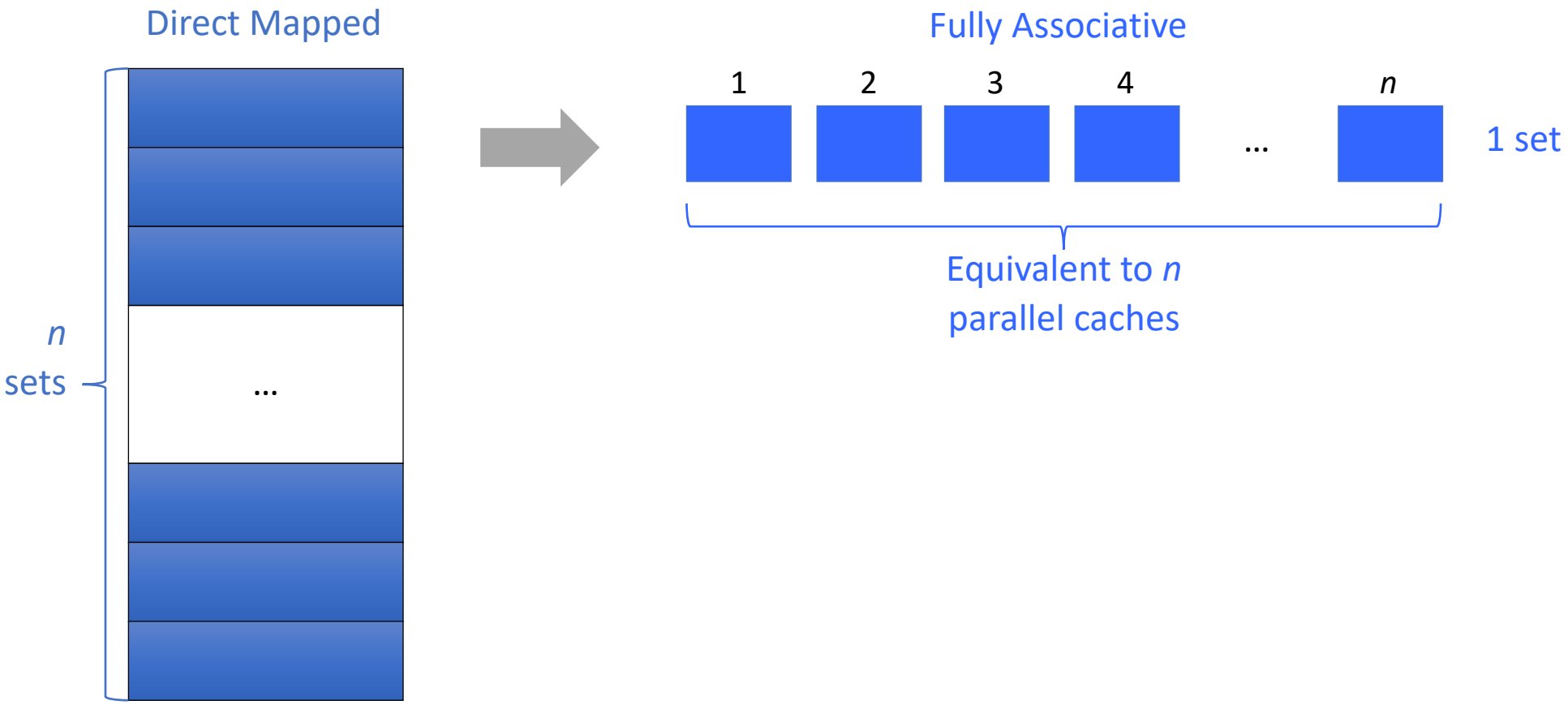
Generalization?



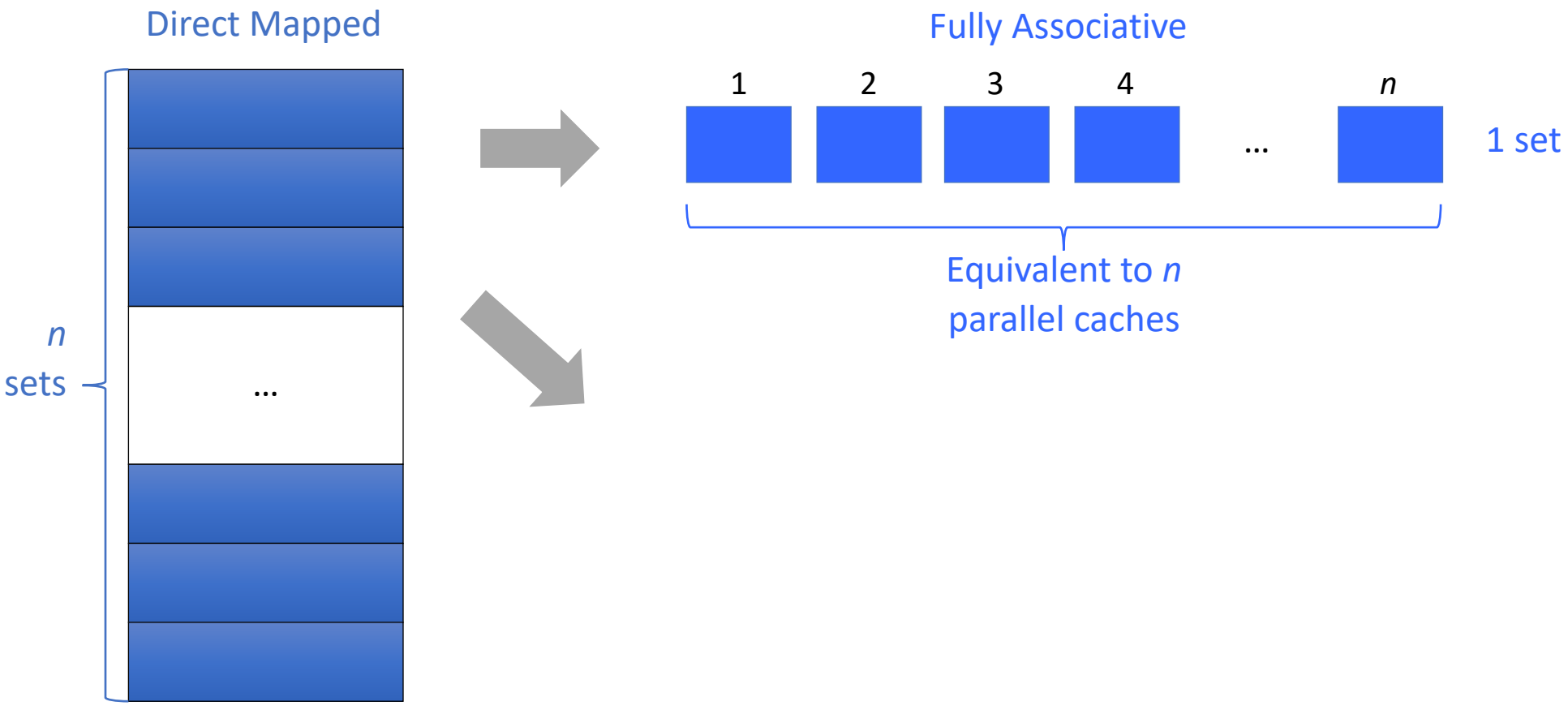
Generalization?



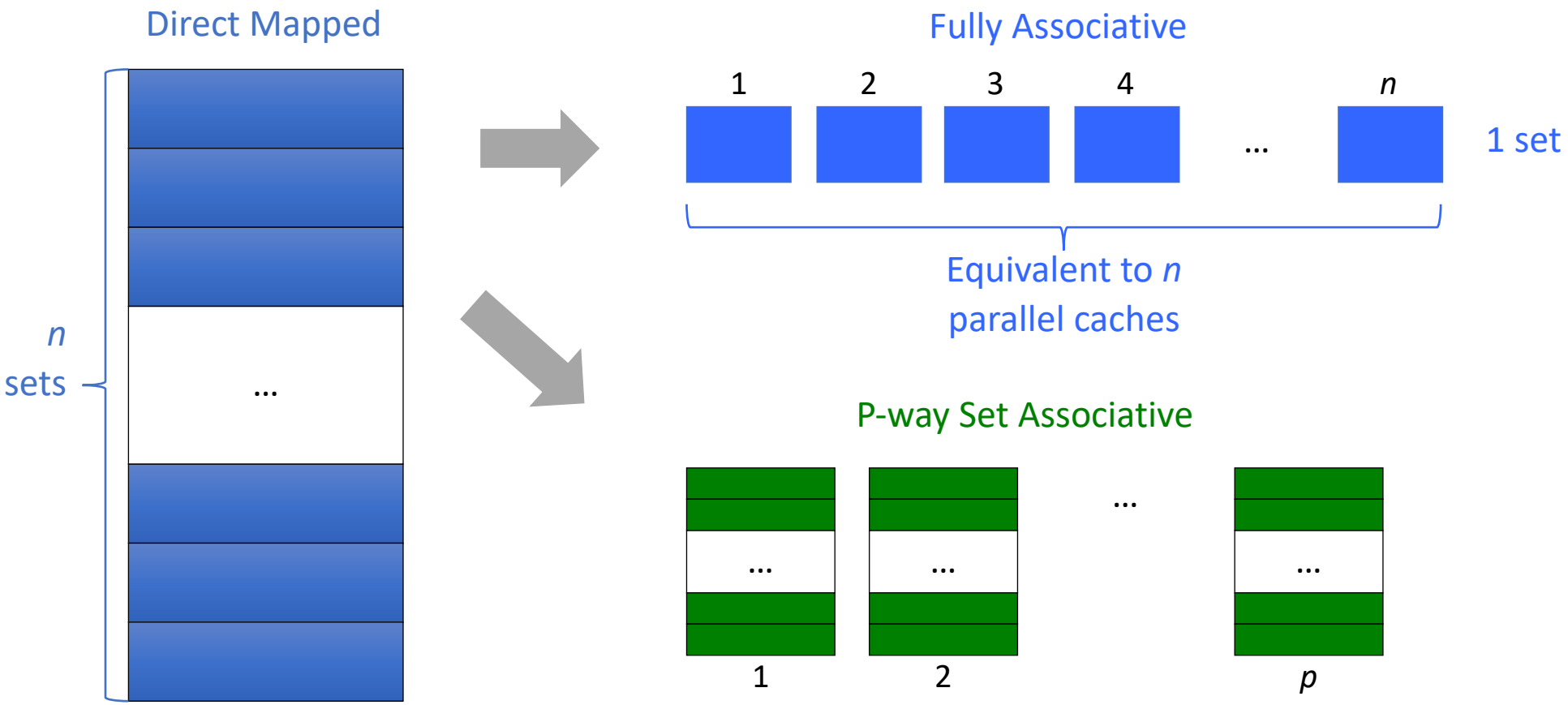
Generalization?



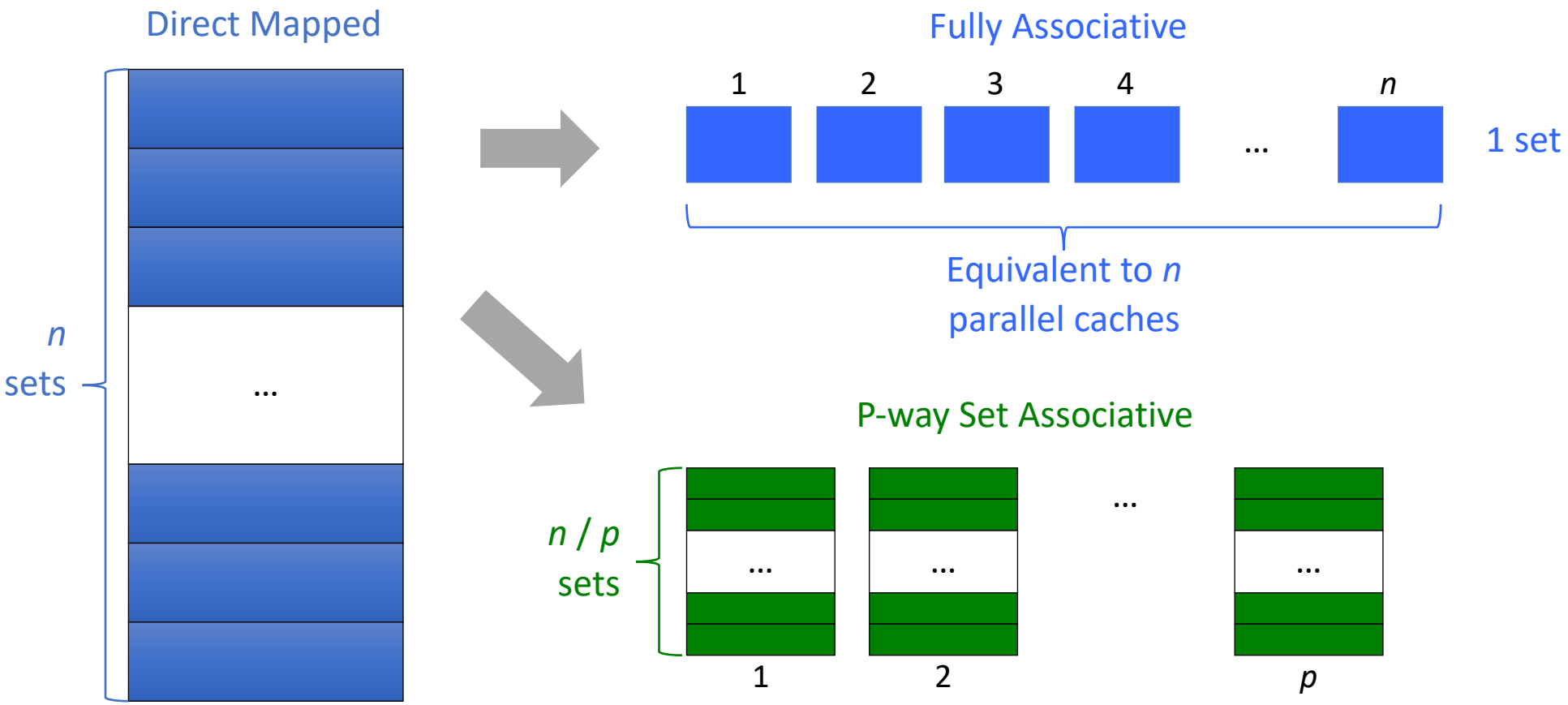
Generalization?



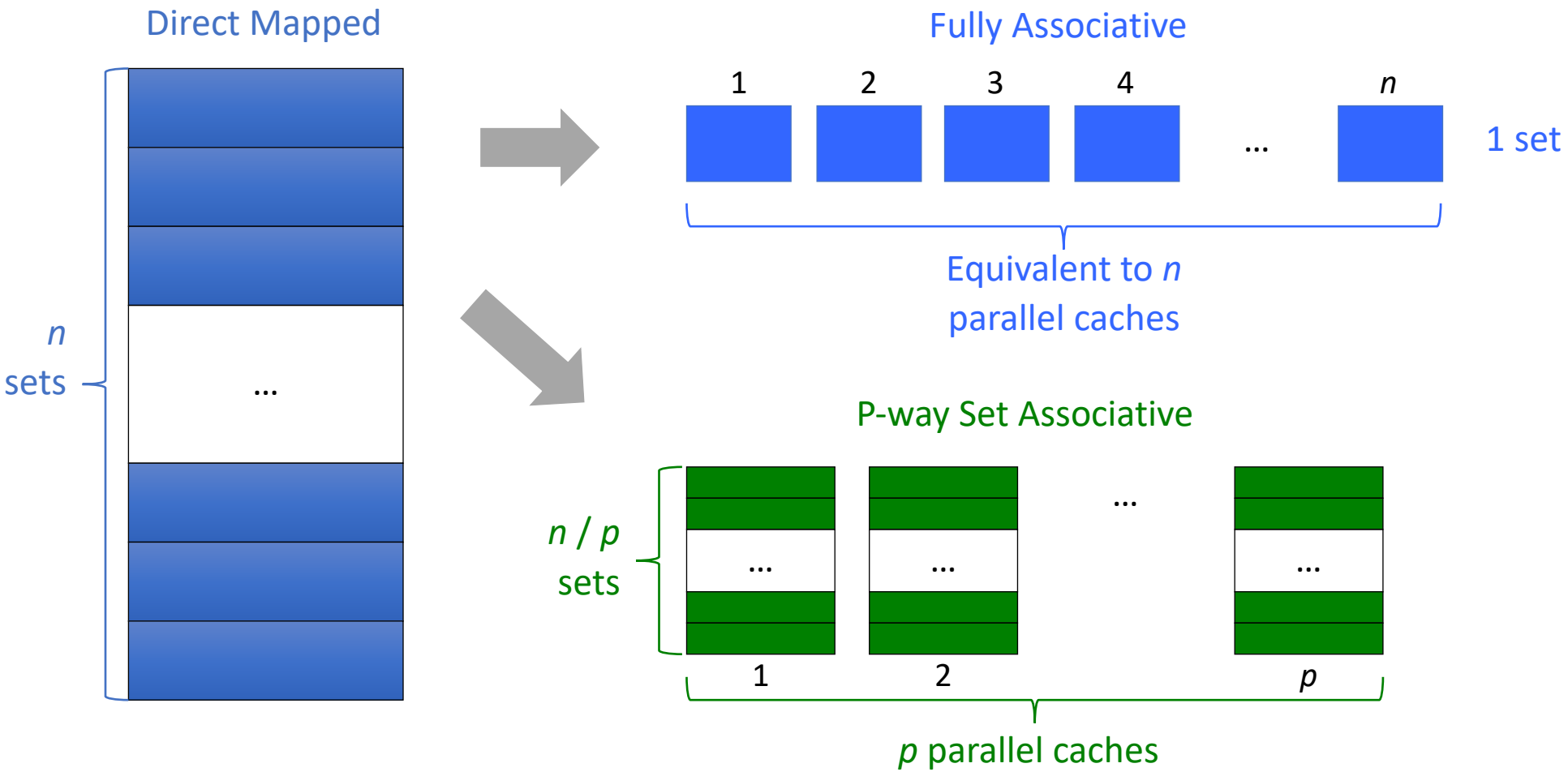
Generalization?



Generalization?

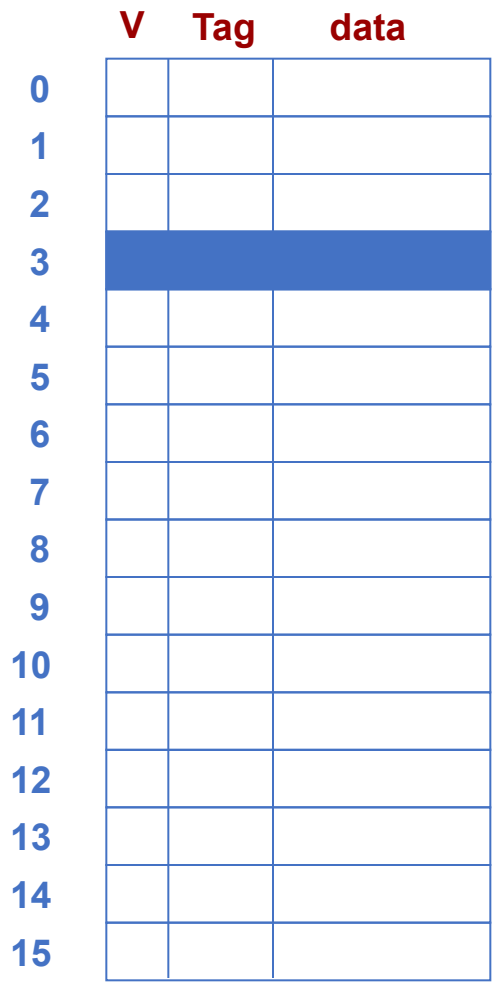


Generalization?



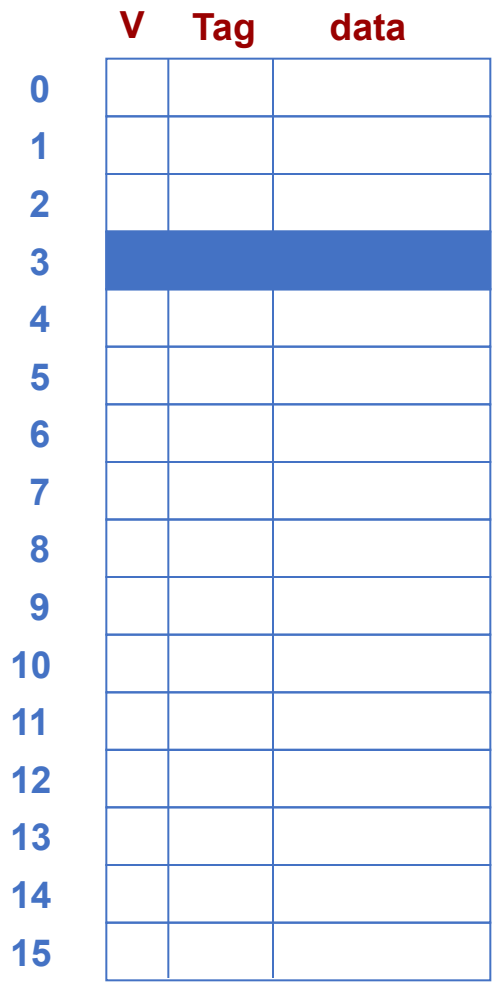
Iso-capacity cache comparison

Iso-capacity cache comparison



(a) Direct-mapped cache

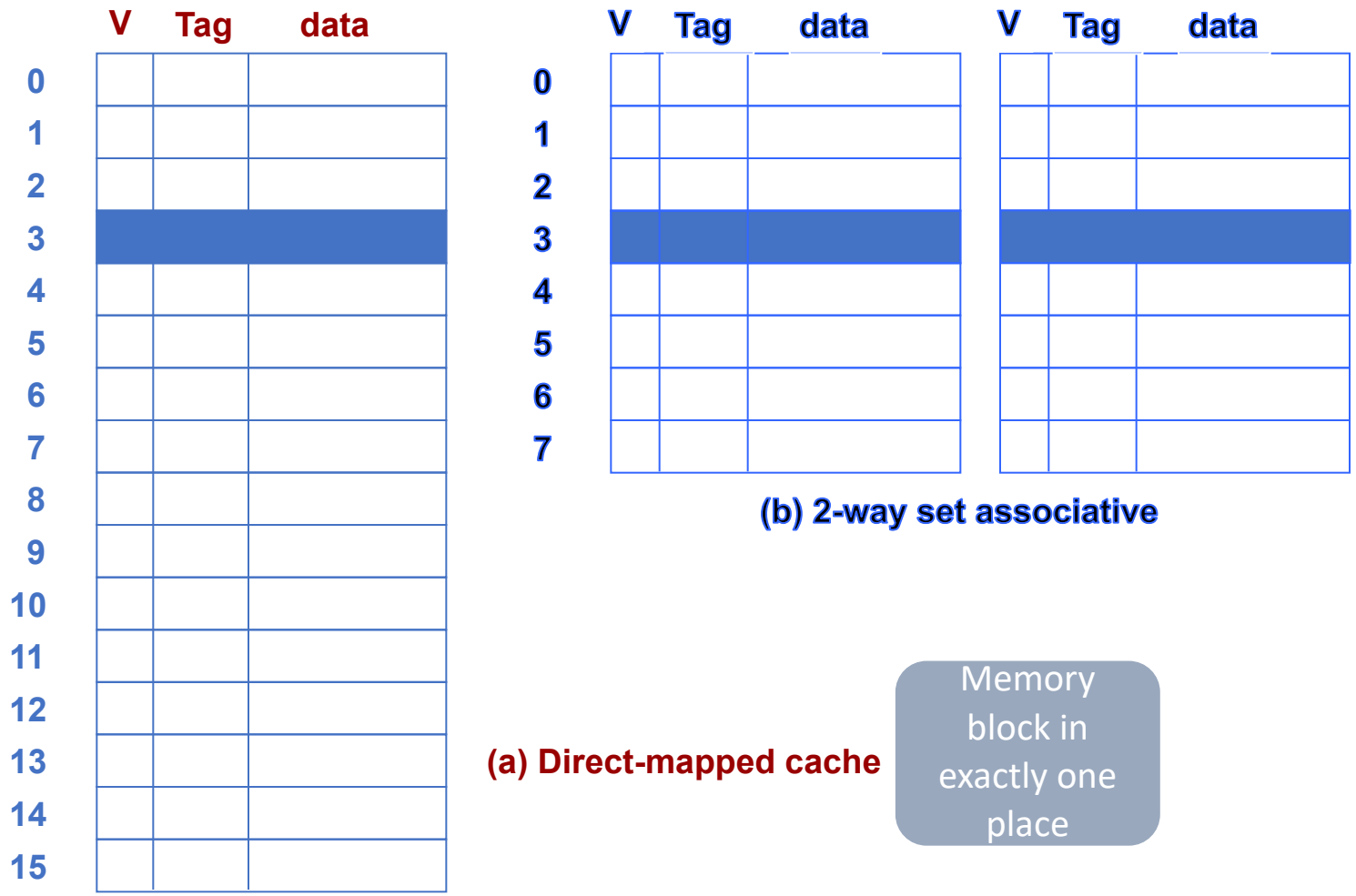
Iso-capacity cache comparison



(a) Direct-mapped cache

Memory block in exactly one place

Iso-capacity cache comparison

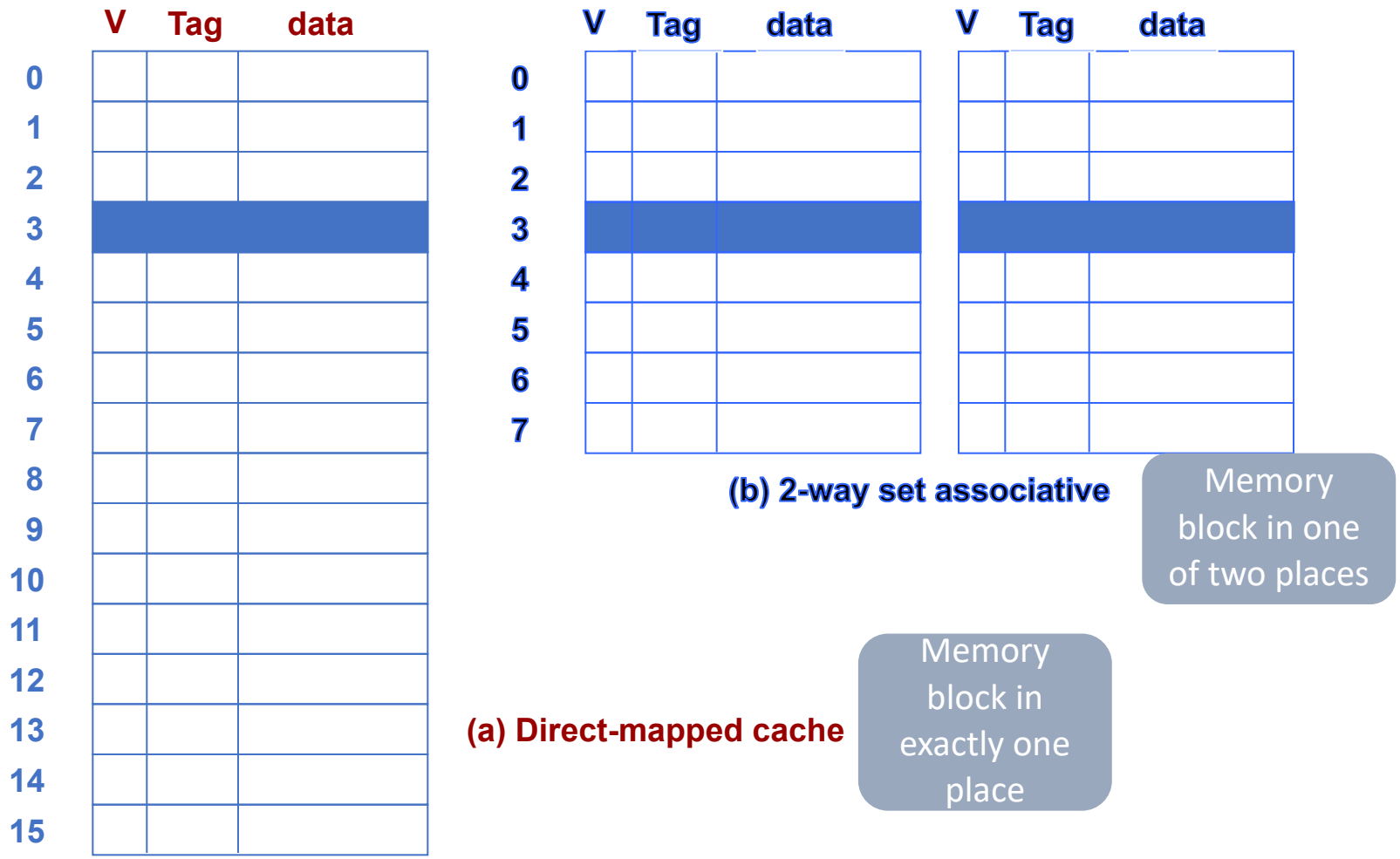


(b) 2-way set associative

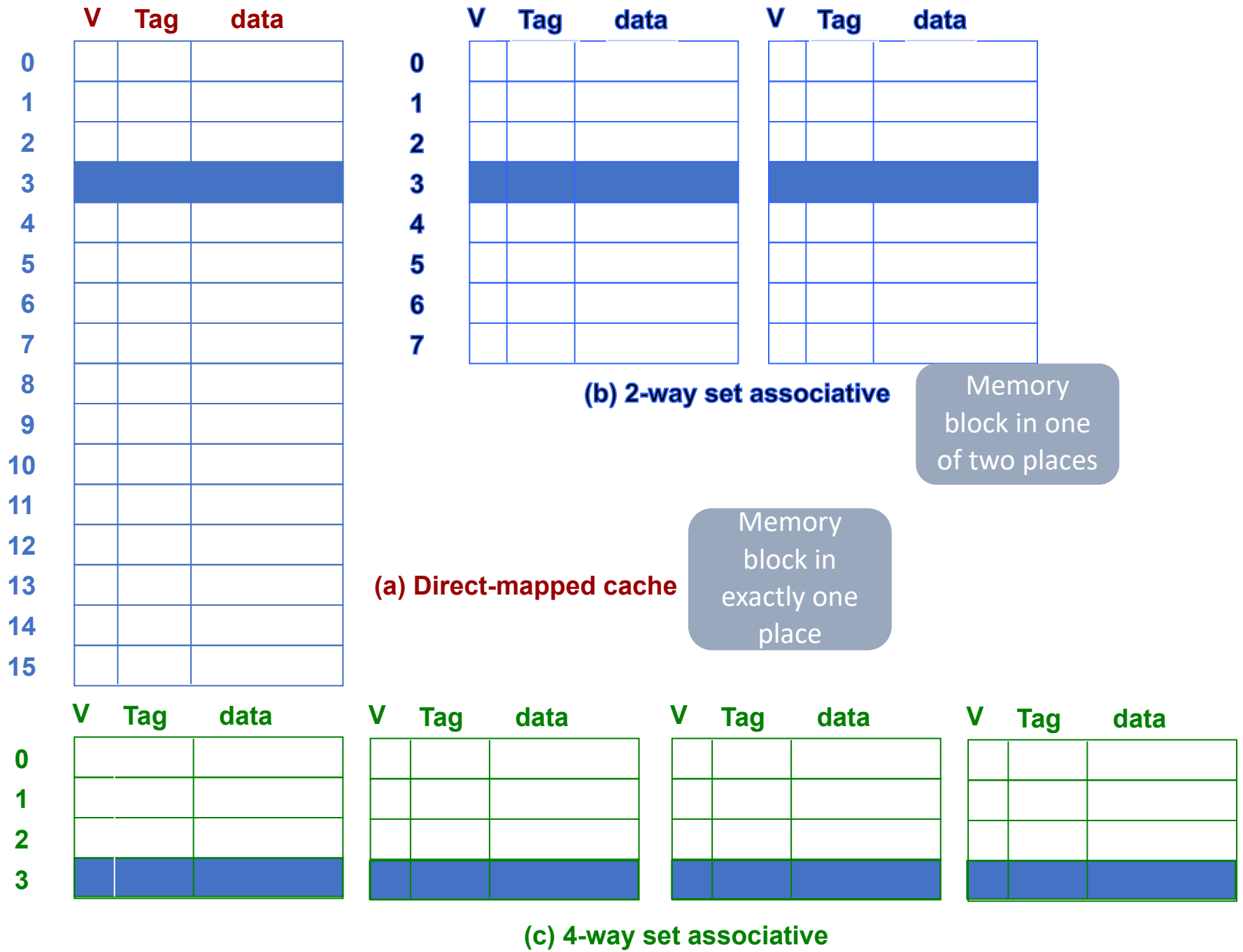
(a) Direct-mapped cache

Memory block in exactly one place

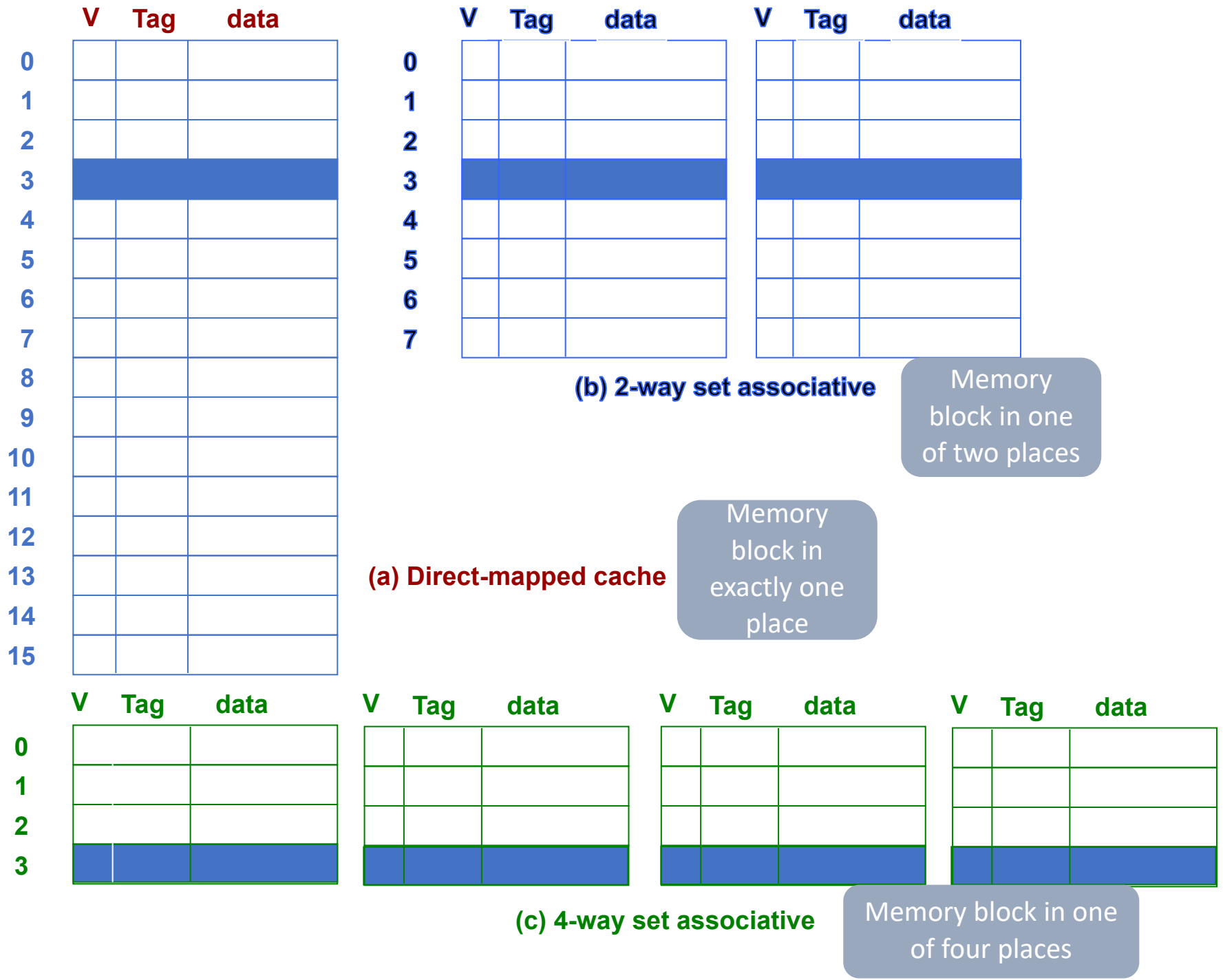
Iso-capacity cache comparison



Iso-capacity cache comparison



Iso-capacity cache comparison



Terminology clarification

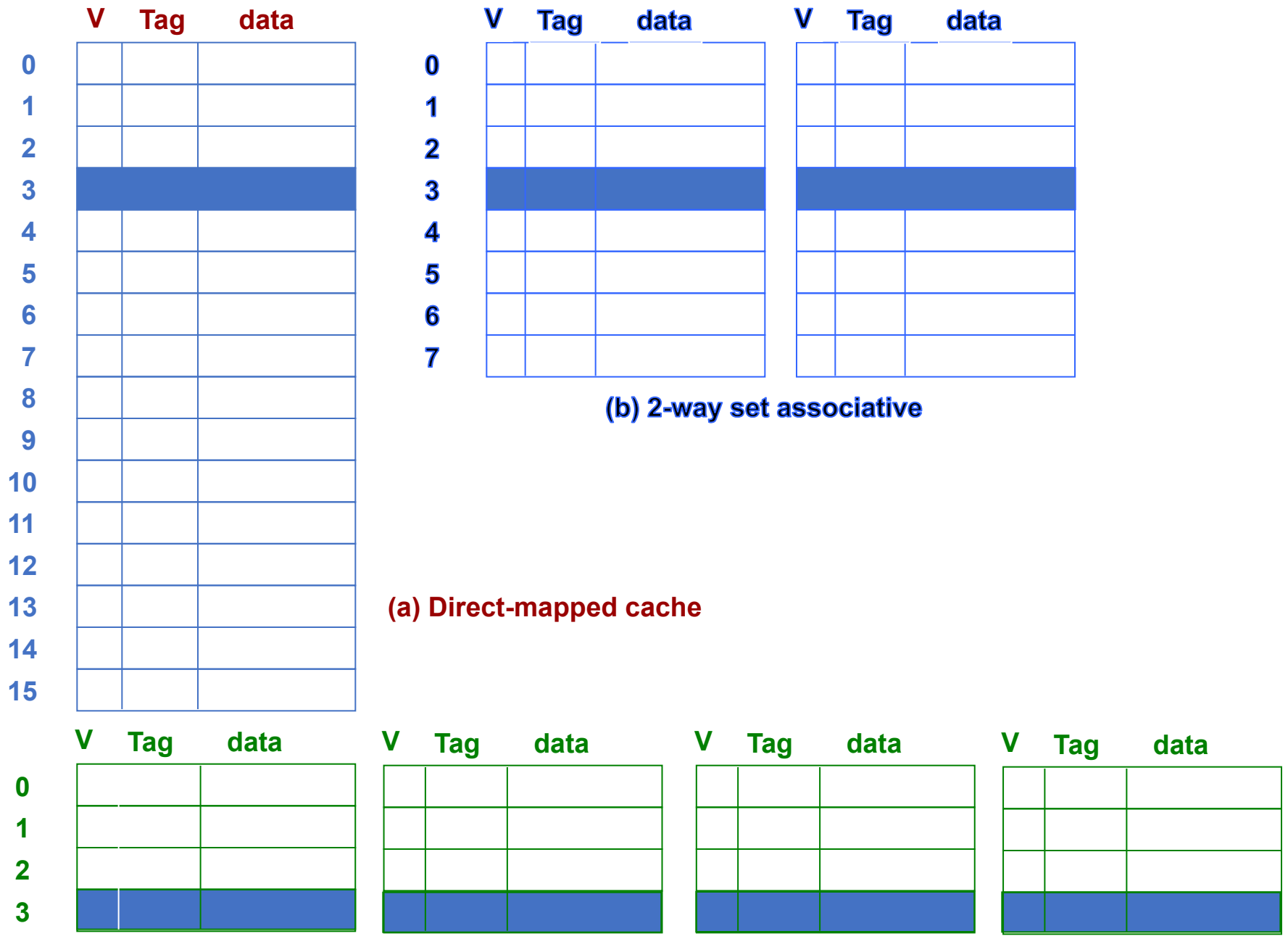
- Unfortunate but…
  - Four different ways of saying the same thing
    - Cache line
    - Cache block
    - Cache entry
    - Cache element

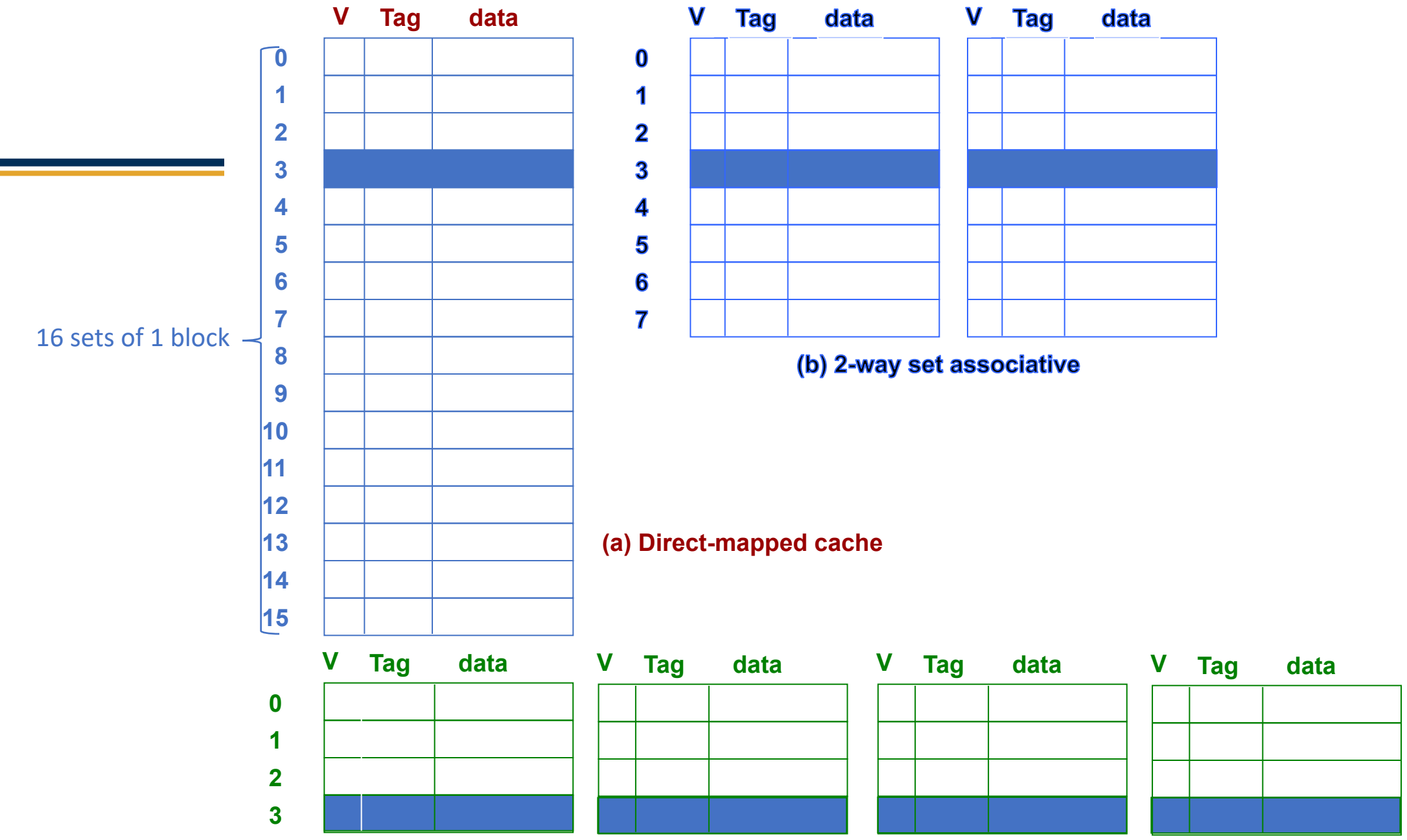
Terminology clarification

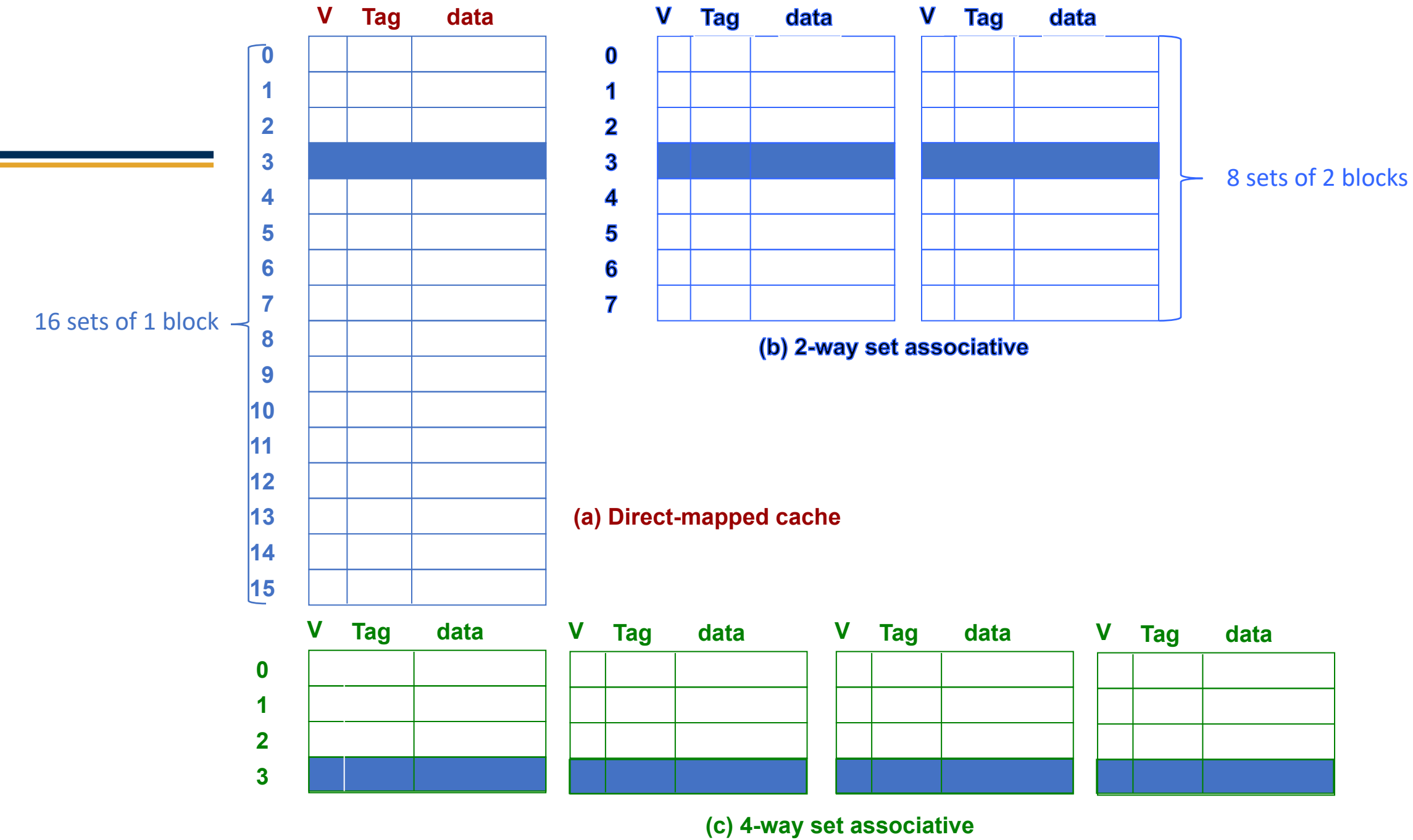
- Unfortunate but…
  - Four different ways of saying the same thing
    - Cache line
    - Cache block
    - Cache entry
    - Cache element
  - All mean the same thing…the basic unit of data transferred into/out of the cache at a time
  - The textbook has several typos in which it erroneously implies cache set is also synonymous to cache block. In addition, from chapter 9.11.3 onwards, almost every occurrence of the term "cache line" should have been "cache set". A cache line is synonymous to a cache block.
    A cache set corresponds to a single cache line only in the case of direct-mapped caches!

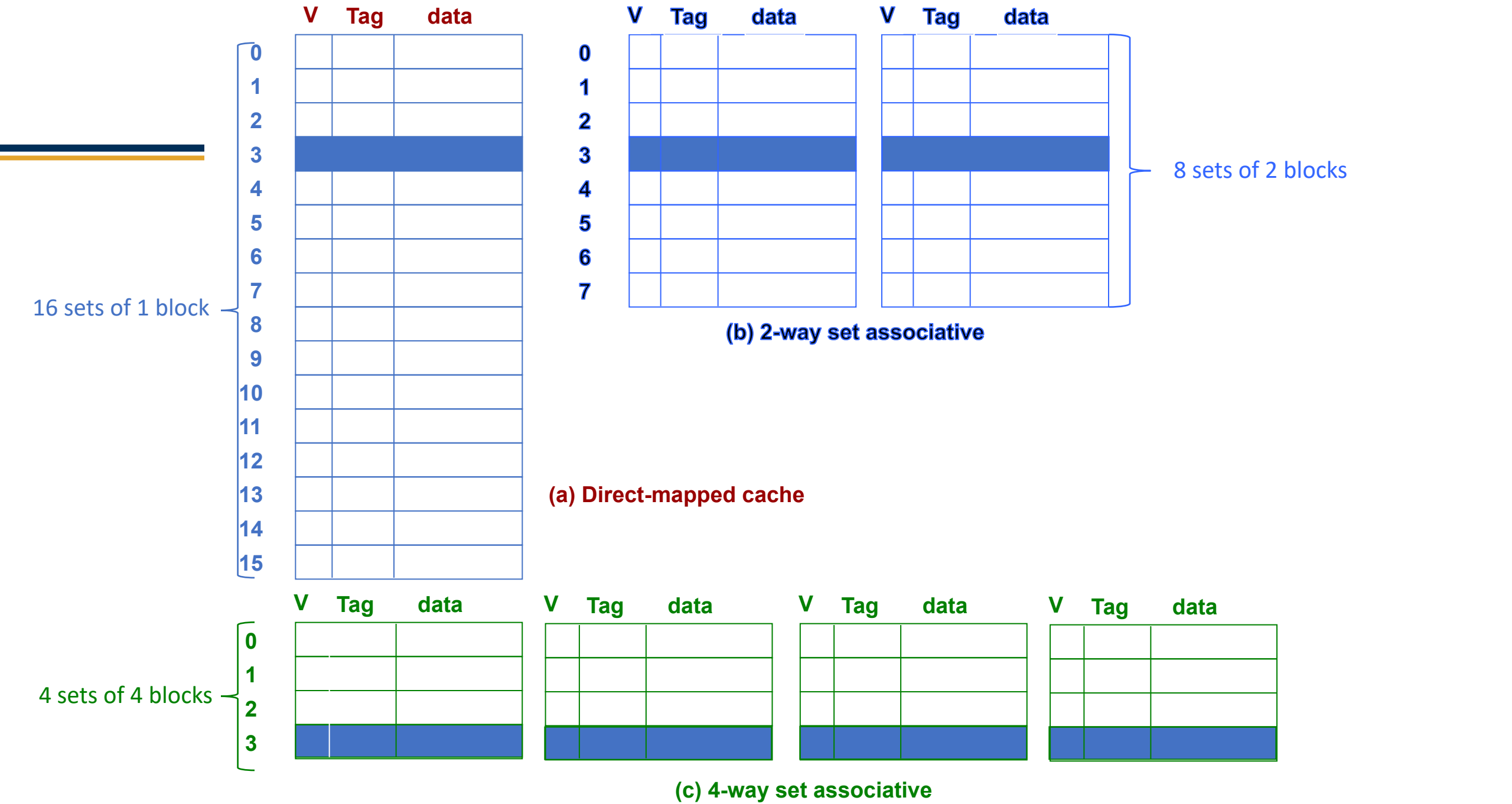
Terminology clarification

- Unfortunate but…
  - Four different ways of saying the same thing
    - Cache line
    - Cache block
    - Cache entry
    - Cache element
  - All mean the same thing…the basic unit of data transferred into/out of the cache at a time
  - The textbook has several typos in which it erroneously implies cache set is also synonymous to cache block. In addition, from chapter 9.11.3 onwards, almost every occurrence of the term "cache line" should have been "cache set". A cache line is synonymous to a cache block.
    A cache set corresponds to a single cache line only in the case of direct-mapped caches!
- A cache set is a "row" in the cache. The number of blocks per set is determined by the type of the cache
  - Direct mapped: n sets, 1 block
  - P-way set associative: n/p sets, p blocks
  - Fully associative: 1 set, n blocks

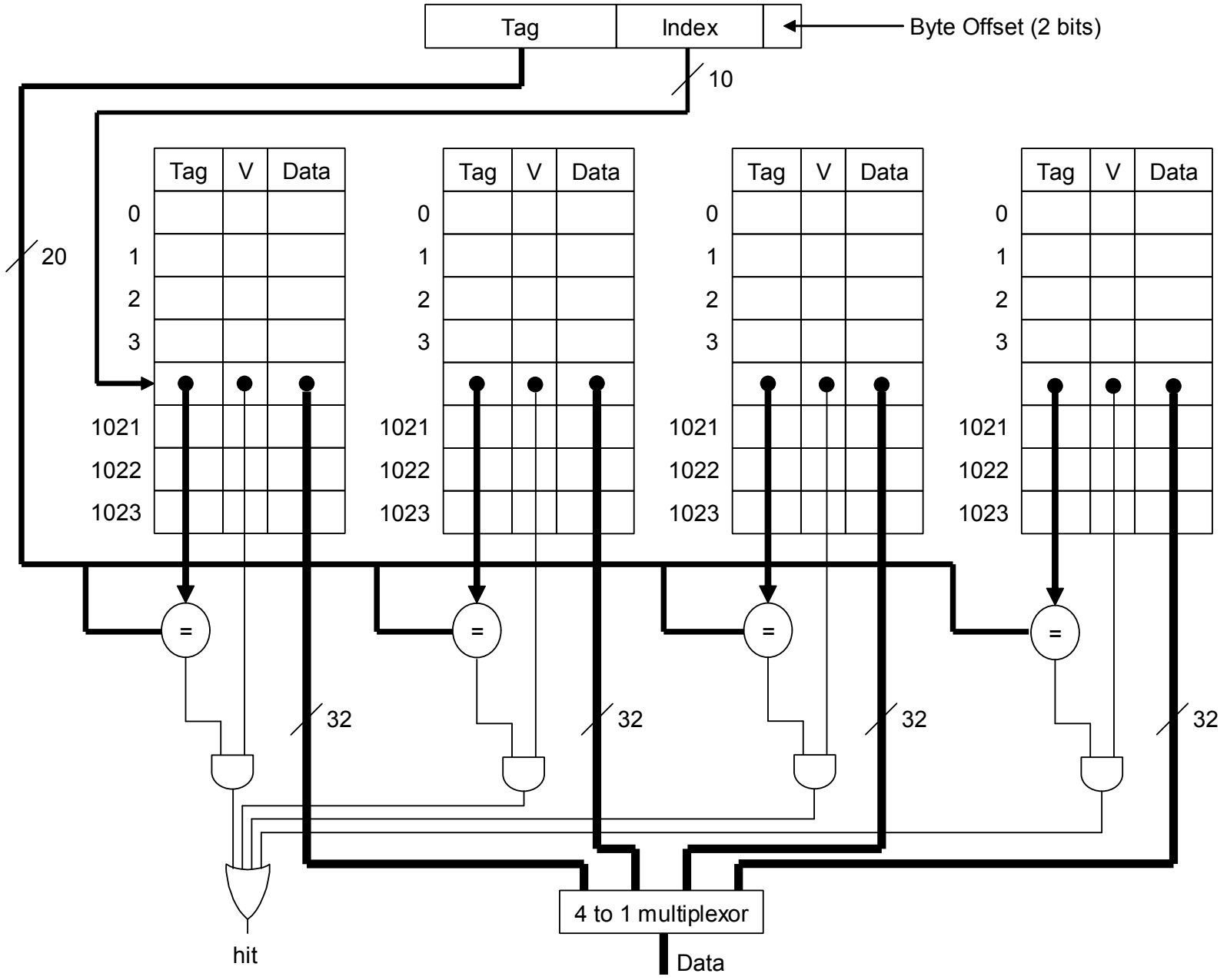

V
Tag
data
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
(a) Direct-mapped cache
(b) 2-way set associative
(c) 4-way set associative


V Tag data
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16 sets of 1 block
(a) Direct-mapped cache
V Tag data
0
1
2
3
4
5
6
7
(b) 2-way set associative
V Tag data
0
1
2
3
(c) 4-way set associative


V
Tag
data
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16 sets of 1 block
(a) Direct-mapped cache
8 sets of 2 blocks
(b) 2-way set associative
(c) 4-way set associative


V
Tag
data
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16 sets of 1 block
(a) Direct-mapped cache
8 sets of 2 blocks
(b) 2-way set associative
4 sets of 4 blocks
(c) 4-way set associative

4-way SA cache organization



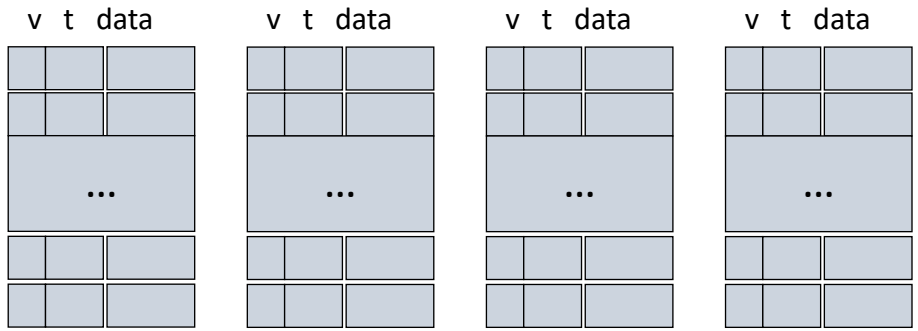
Example: 4-way set associative cache

Incoming – dag2200

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

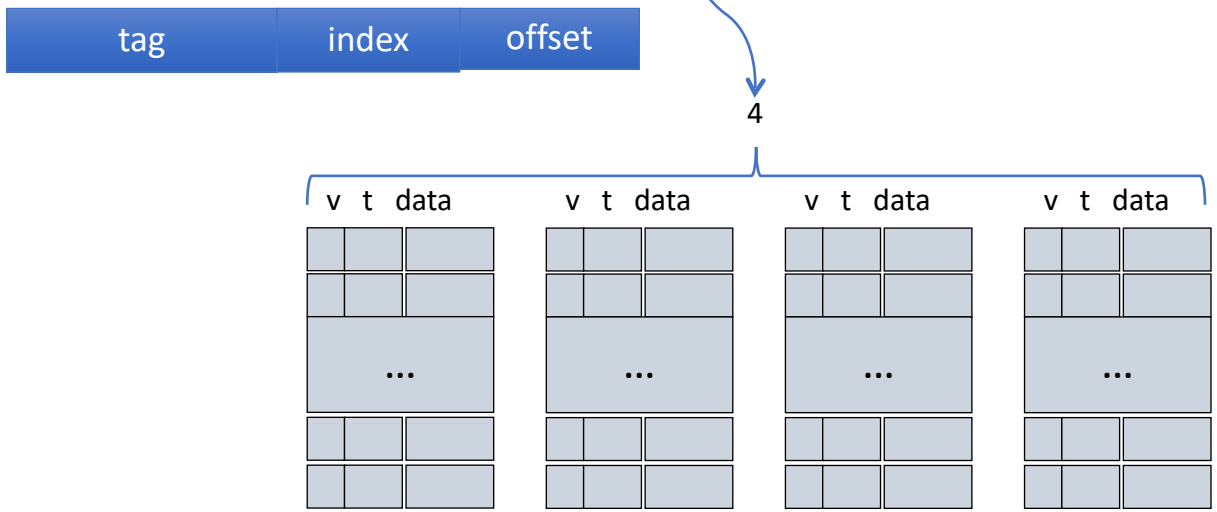


Example: 4-way set associative cache

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)



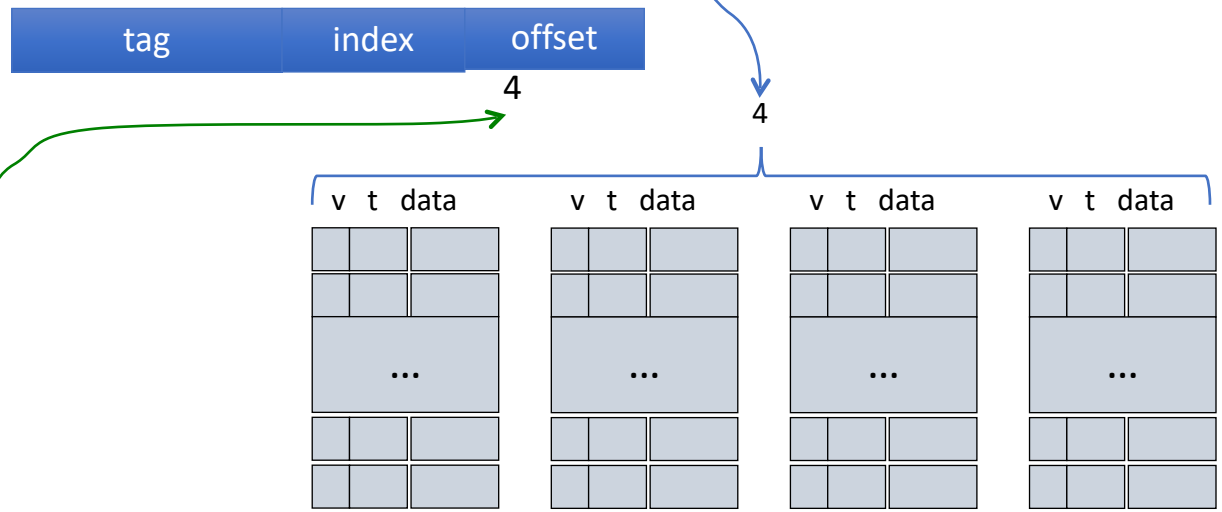
Example: 4-way set associative cache

Incoming – dag2200

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)



Example: 4-way set associative cache

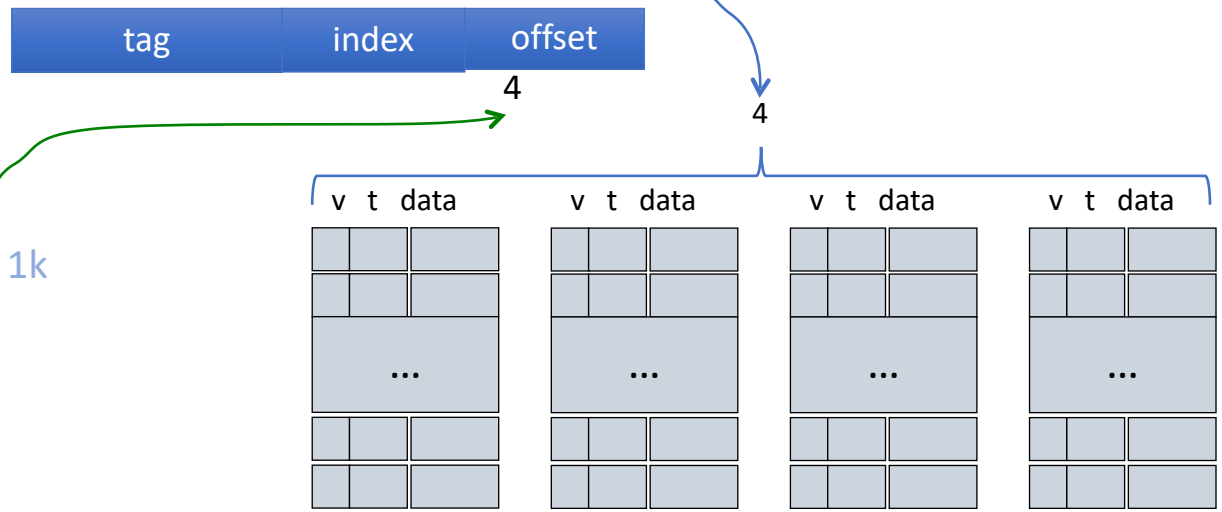
Incoming – dag2200

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

# of Sets? 64K/4/16 = 1k

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)



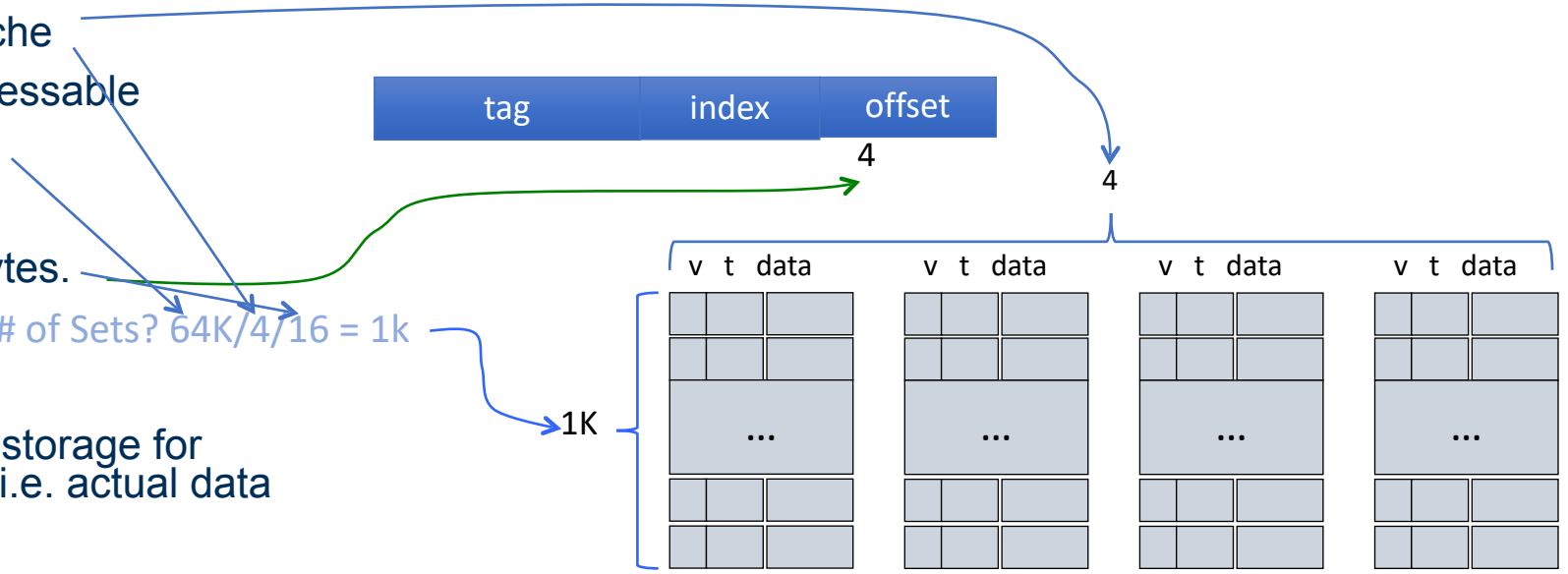
Example: 4-way set associative cache

Incoming – dag2200

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)



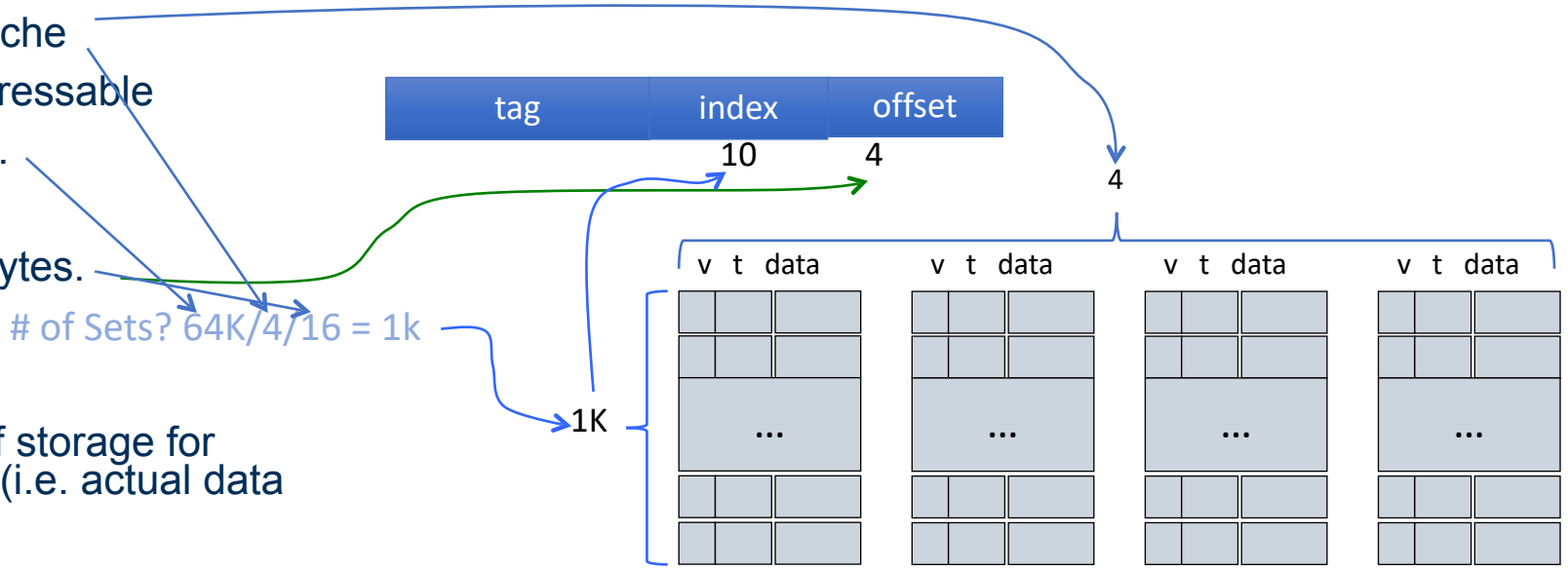
Example: 4-way set associative cache

Incoming – dag2200

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

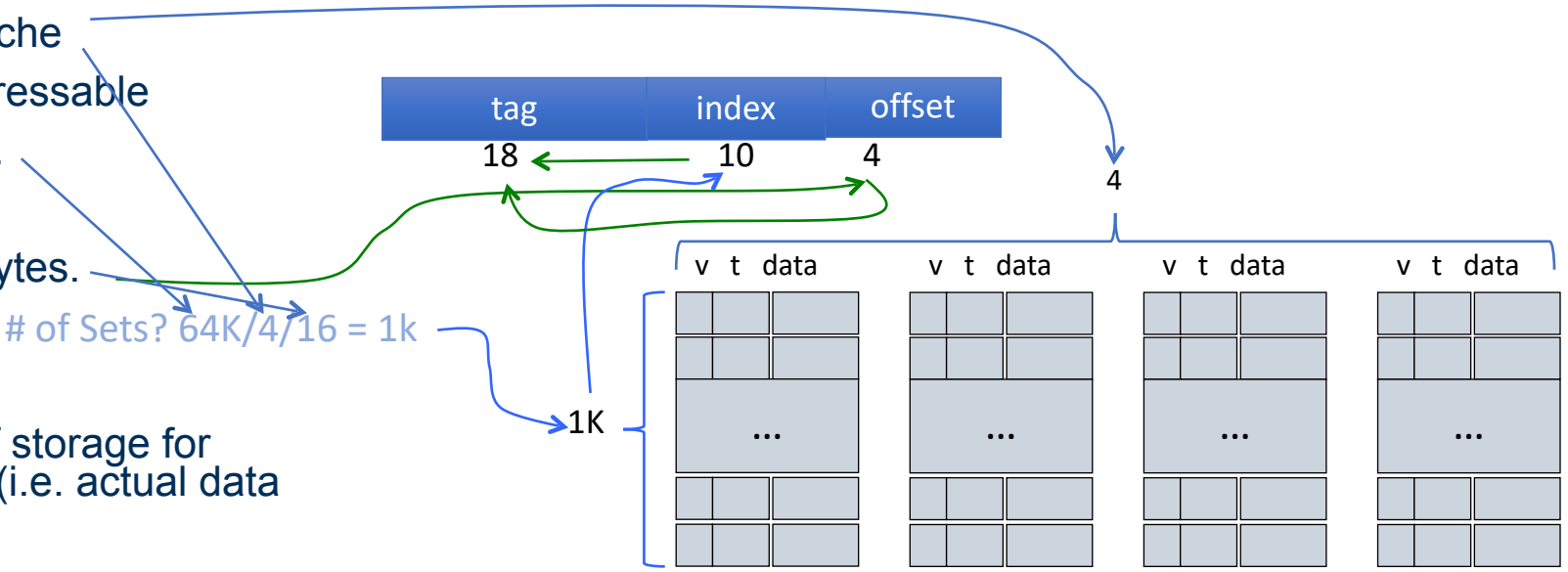


Example: 4-way set associative cache

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

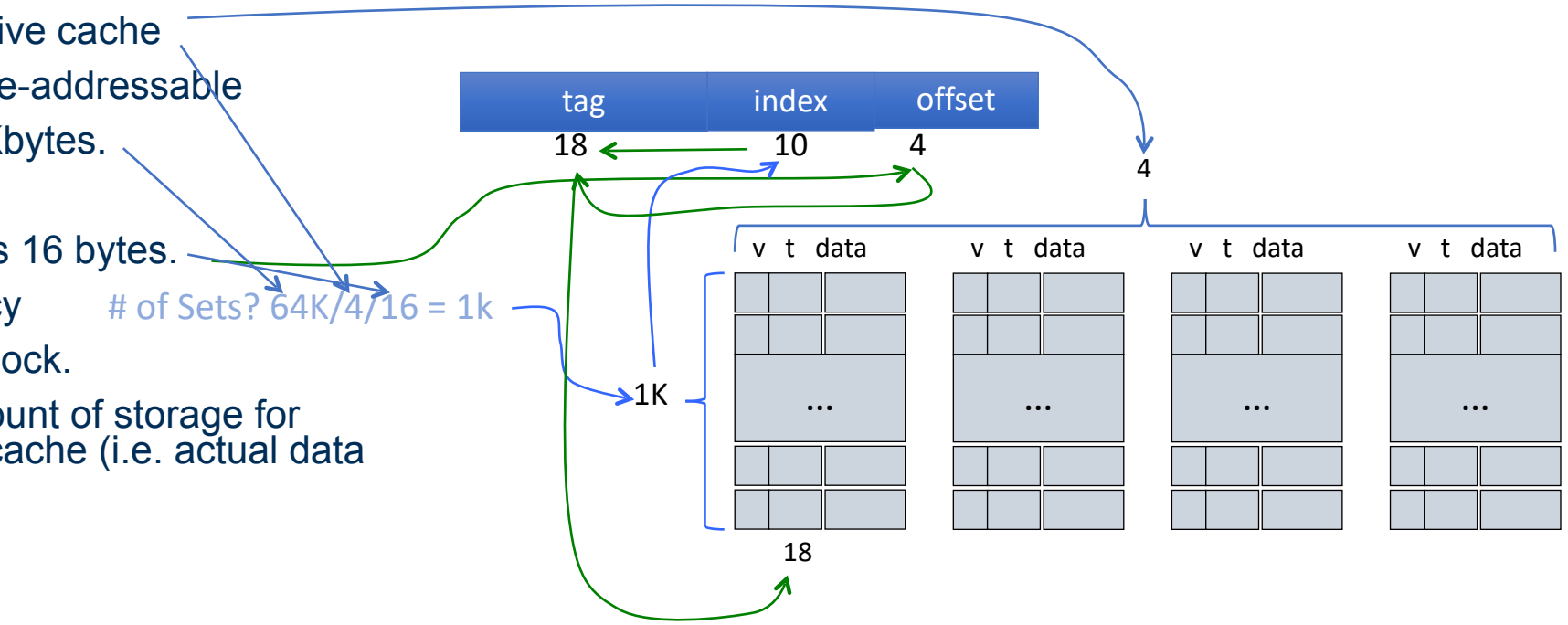


Example: 4-way set associative cache

- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)



Example: 4-way set associative cache

Incoming – dag2200

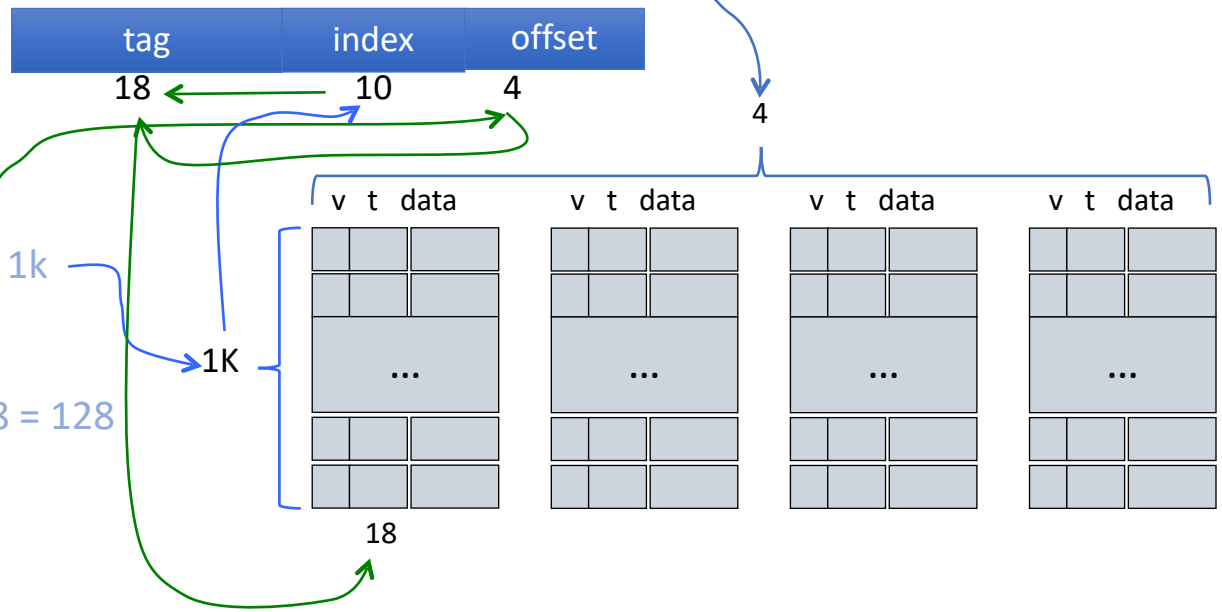
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

# of Sets? 64K/4/16 = 1k

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

16*8 = 128



Example: 4-way set associative cache

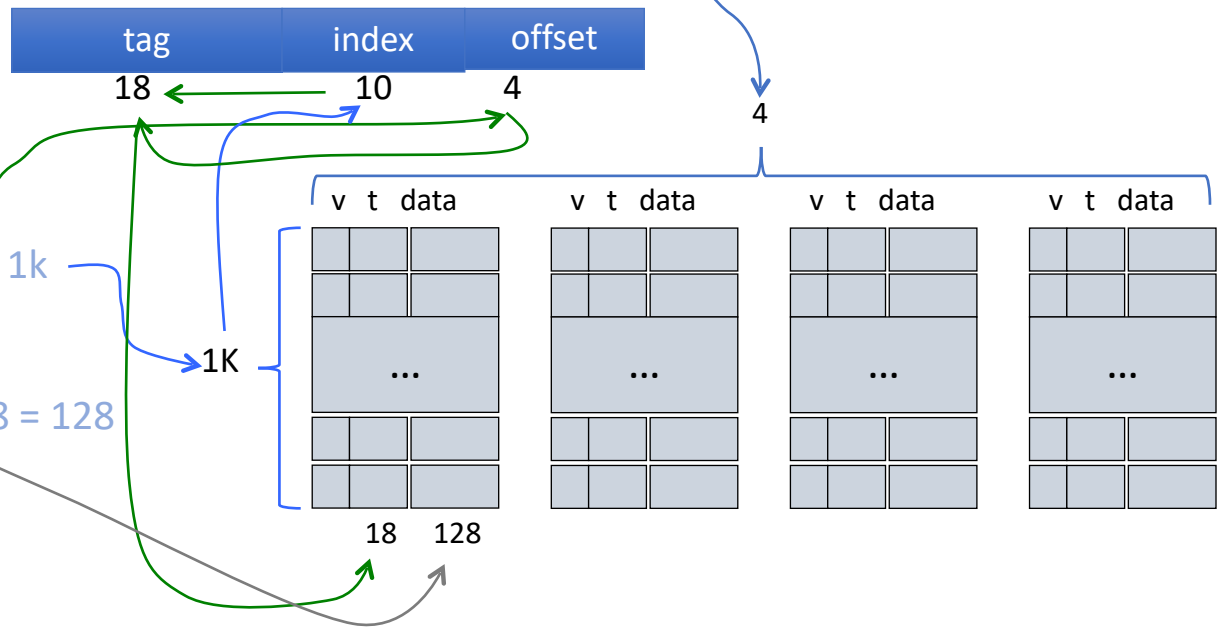
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

# of Sets? 64K/4/16 = 1k

16*8 = 128



Example: 4-way set associative cache

Incoming – dag2200

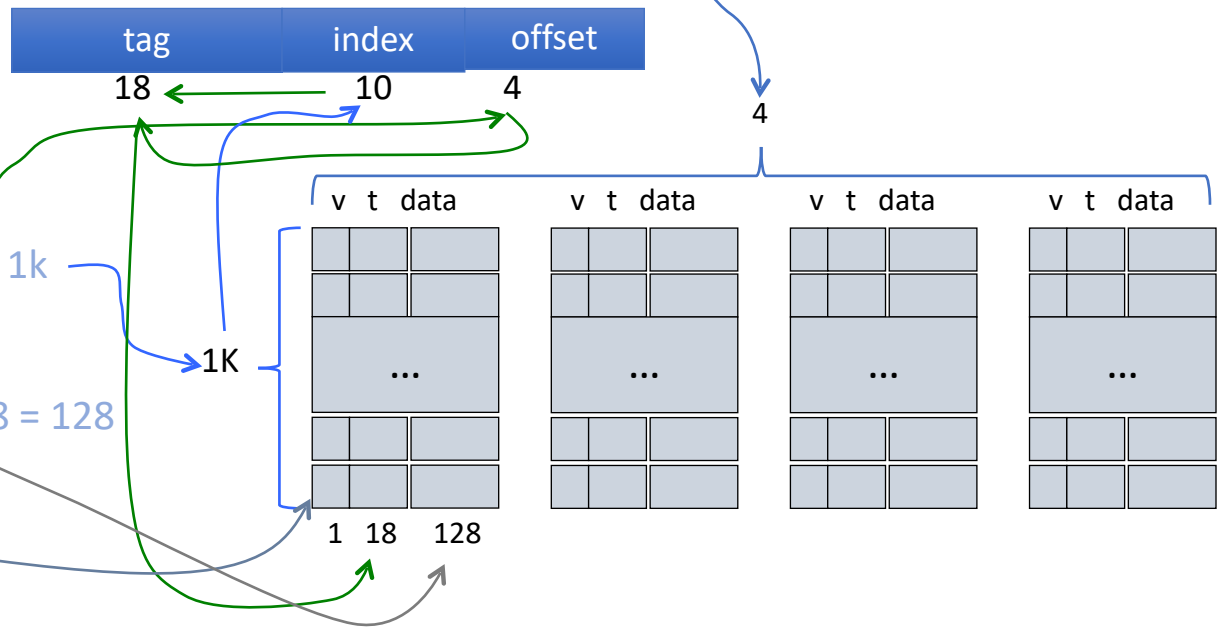
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

# of Sets? 64K/4/16 = 1k

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

16*8 = 128



Example: 4-way set associative cache

Incoming – dag2200

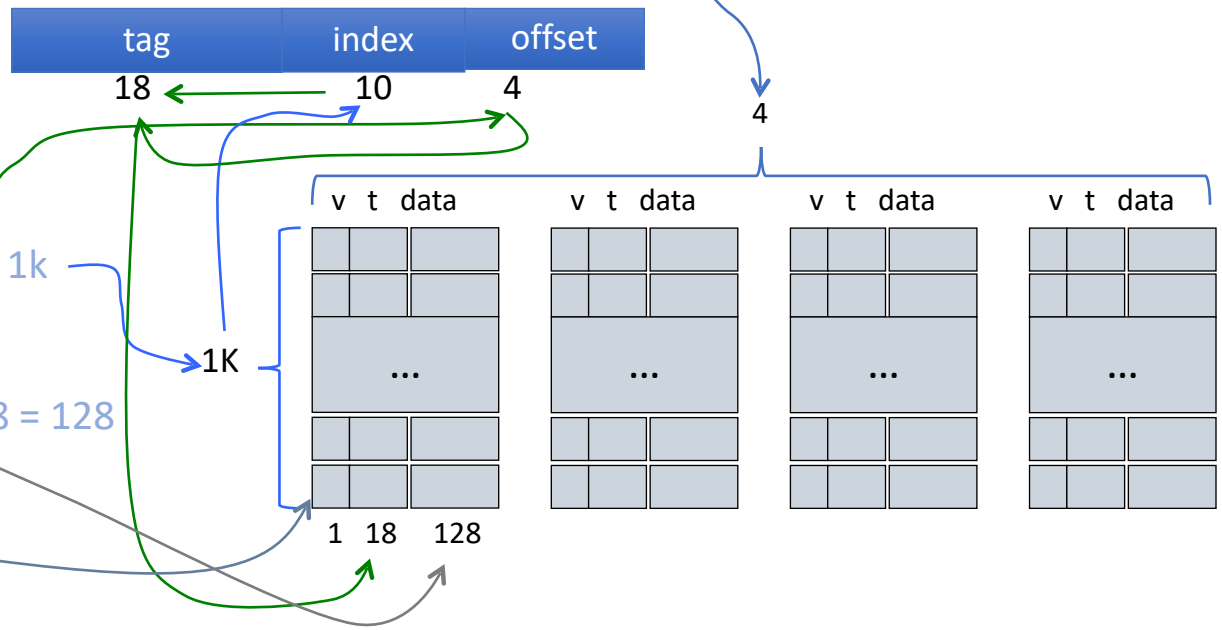
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

# of Sets? 64K/4/16 = 1k

16*8 = 128



(1+18+128) * 1K * 4 / 8 = 75,264 bytes

Example: 4-way set associative cache

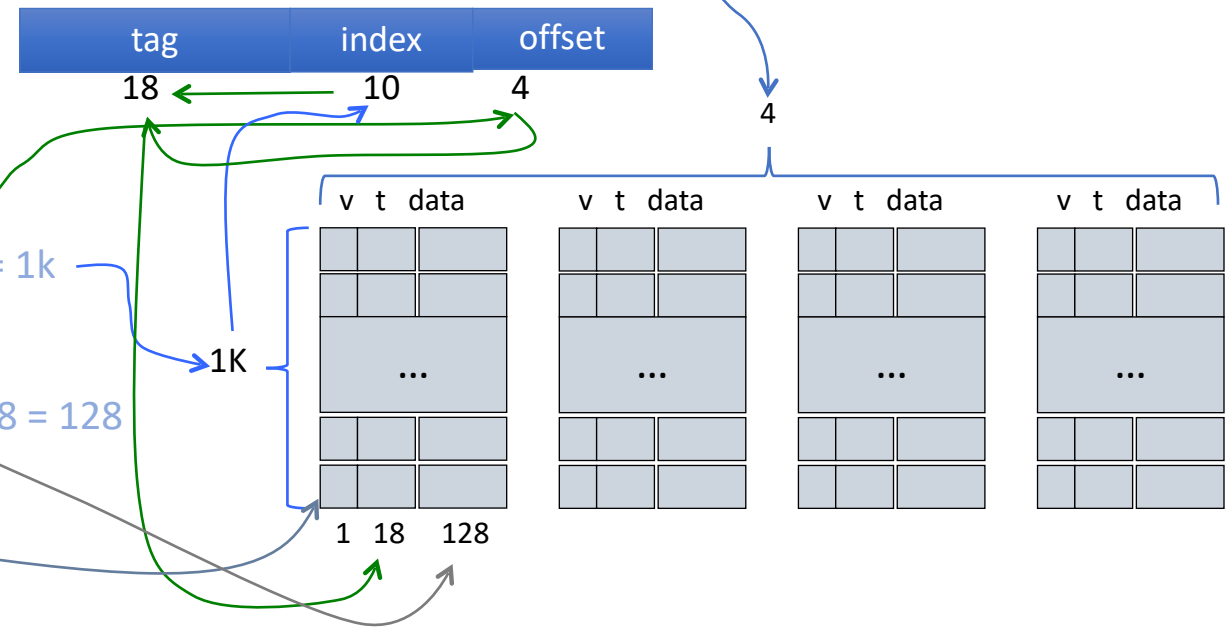
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

# of Sets? 64K/4/16 = 1k

16*8 = 128



(1+18+128) * 1K * 4 / 8 = 75,264 bytes

Hardware complexity?

Example: 4-way set associative cache

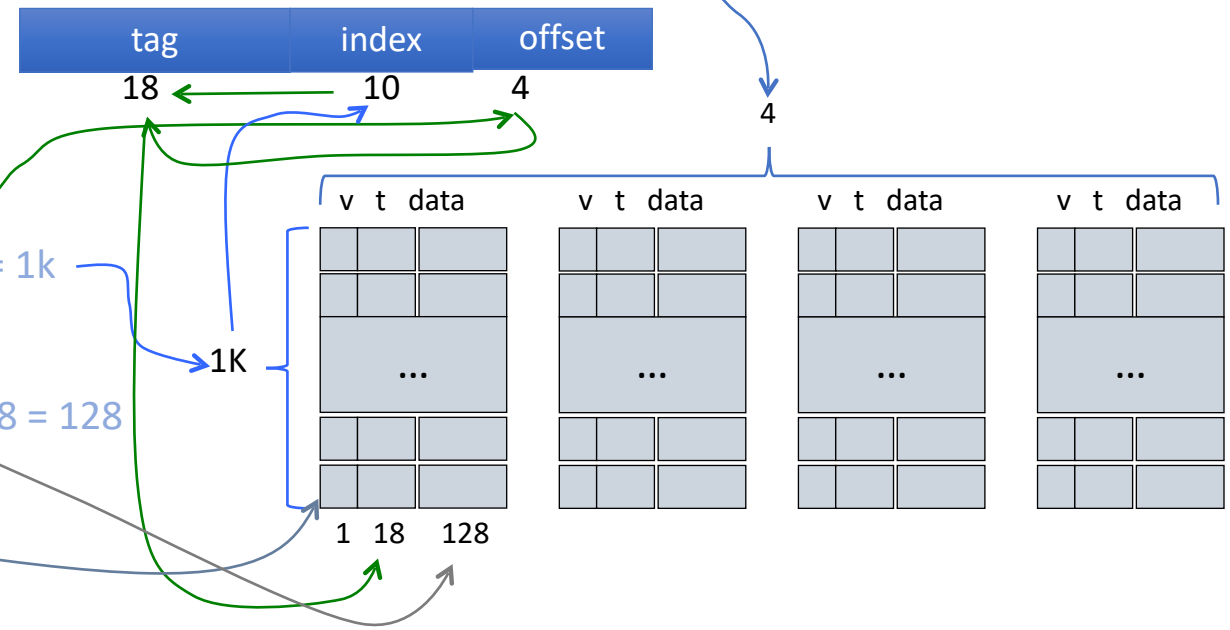
- 4-way set associative cache
- 32-bit memory, byte-addressable
- Cache size of 64 Kbytes.

- Cache block size is 16 bytes.
- Write-through policy
- One valid bit per block.

Compute the total amount of storage for implementing the cache (i.e. actual data plus the metadata)

# of Sets? 64K/4/16 = 1k

16*8 = 128



(1+18+128) * 1K * 4 / 8 = 75,264 bytes

Hardware complexity?

4 18-bit comparators

Session ID: dag2200

In a fully associative cache …

…with 64K bytes of data, 64 bytes / block and a *t*-bit tag

19% A. There are four *t*-bit tag comparators

30% B. There are 64 *t*-bit tag comparators

43% C. There are 1k *t*-bit tag comparators

8% D. There is one *t*-bit tag comparator for the entire cache

Session ID: dag2200

In a fully associative cache …

…with 64K bytes of data, 64 bytes / block and a *t*-bit tag

19% A. There are four *t*-bit tag comparators

30% B. There are 64 *t*-bit tag comparators

43% C. There are 1k *t*-bit tag comparators

8% D. There is one *t*-bit tag comparator for the entire cache

FA caches have one comparator for each way in the cache.
That means 64K/64 = 1K is the number of comparators.
(In FA cache, # of ways also happens to be equal to # of cache blocks)

Session ID: dag2200

In a 4-way set associative cache, …

…with 64K bytes of data, 64 bytes / block, with a *t*-bit tag

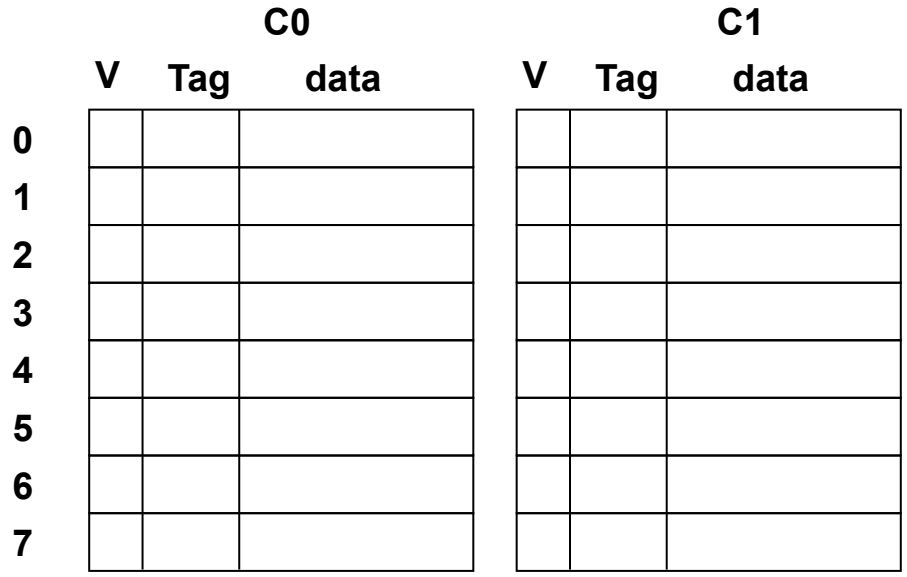
49% A. There are four *t*-bit tag comparators

29% B. There are 64 *t*-bit tag comparators

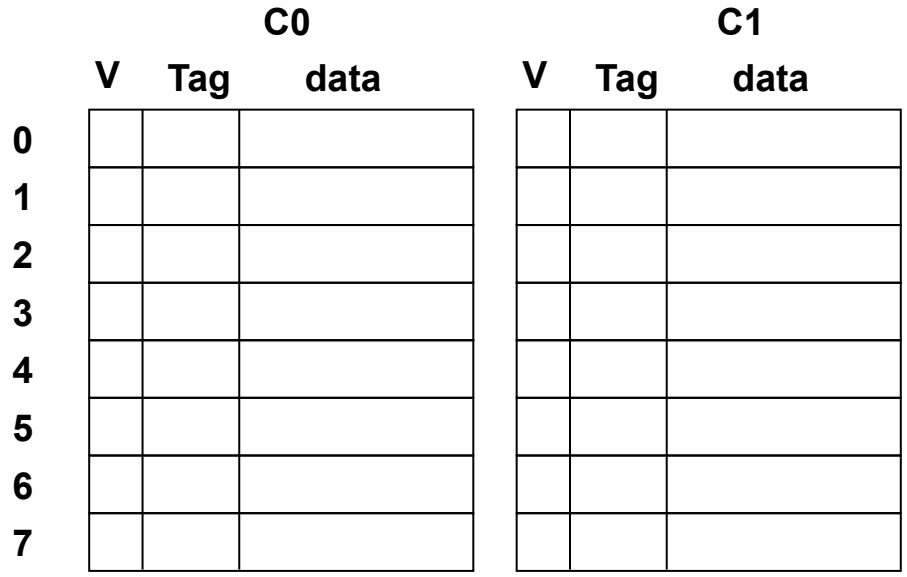
17% C. There are 1k *t*-bit tag comparators

4% D. There is one *t*-bit tag comparator for the entire cache

What about cache replacement policy?

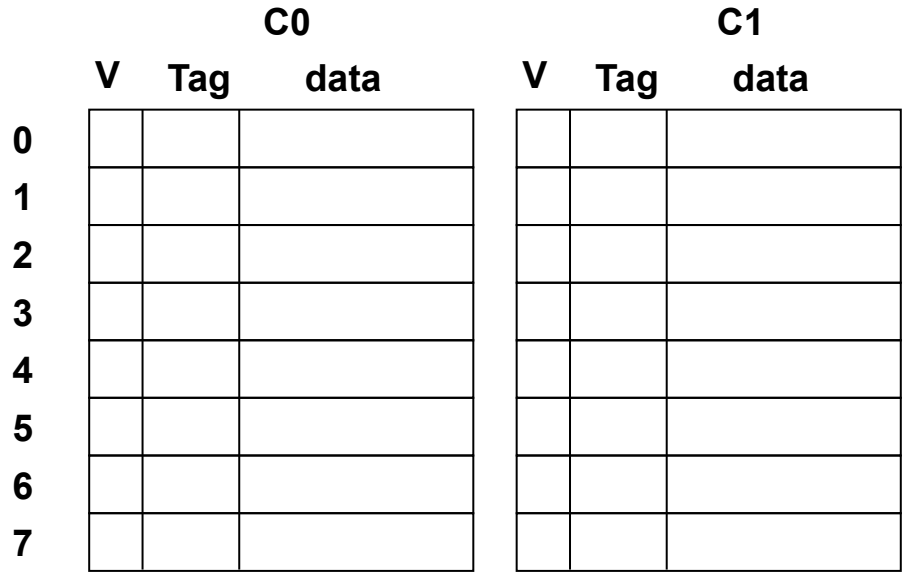


What about cache replacement policy?



- What kind of cache is this?

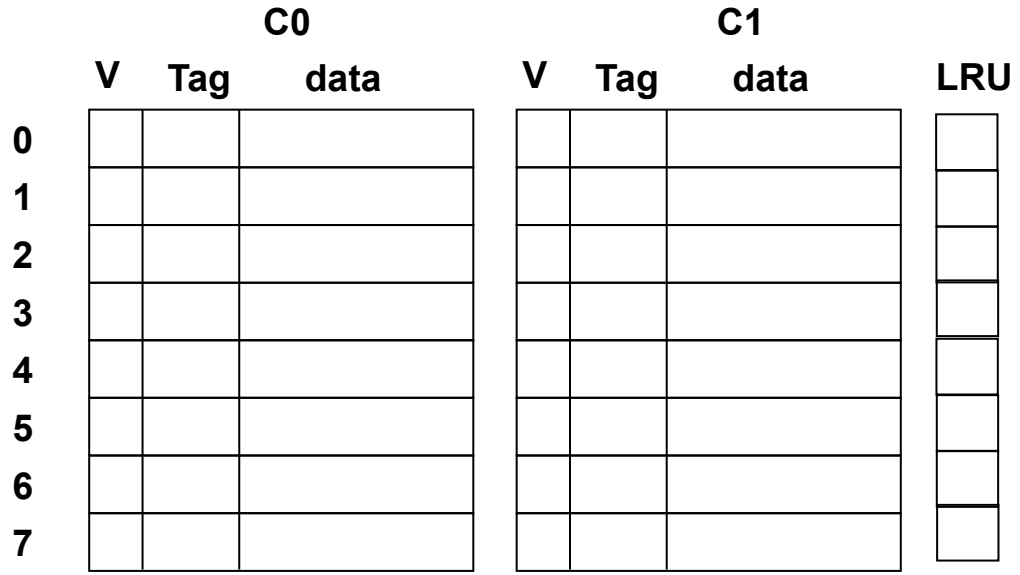
What about cache replacement policy?



- What kind of cache is this?

2-way set associative

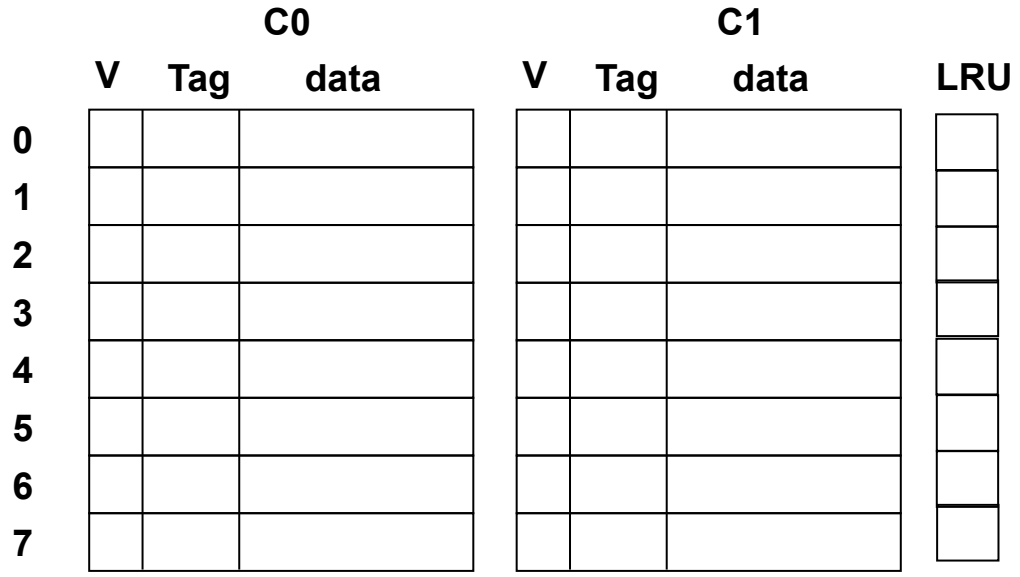
What about cache replacement policy?



- What kind of cache is this?

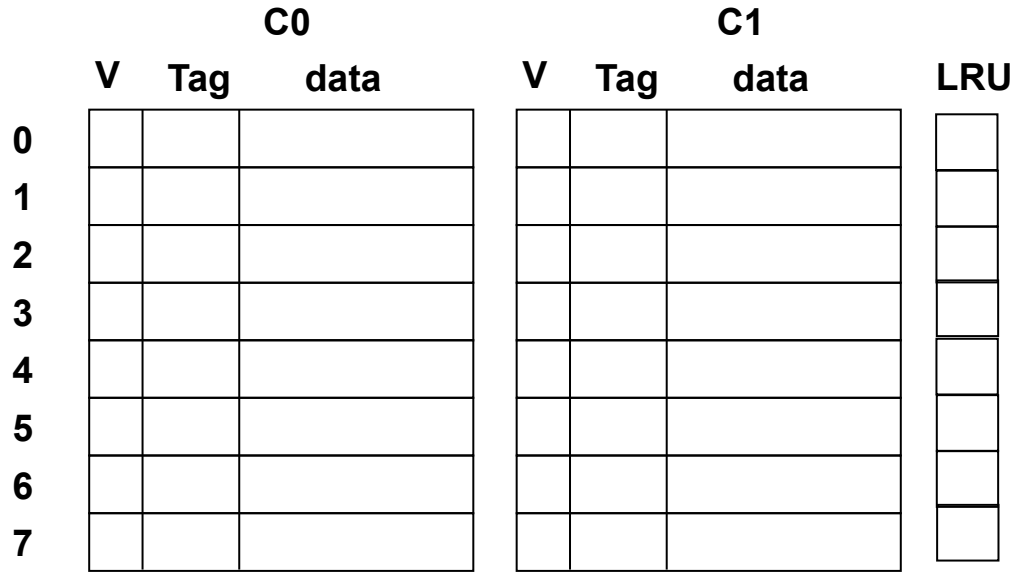
2-way set associative

What about cache replacement policy?



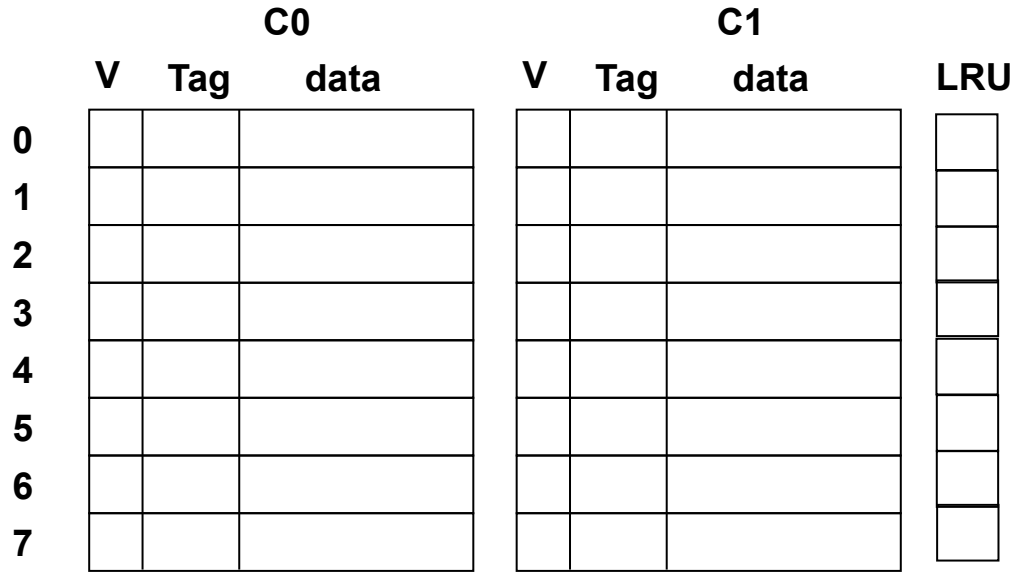
- What kind of cache is this? 2-way set associative
- How many LRU bits do we need?

What about cache replacement policy?



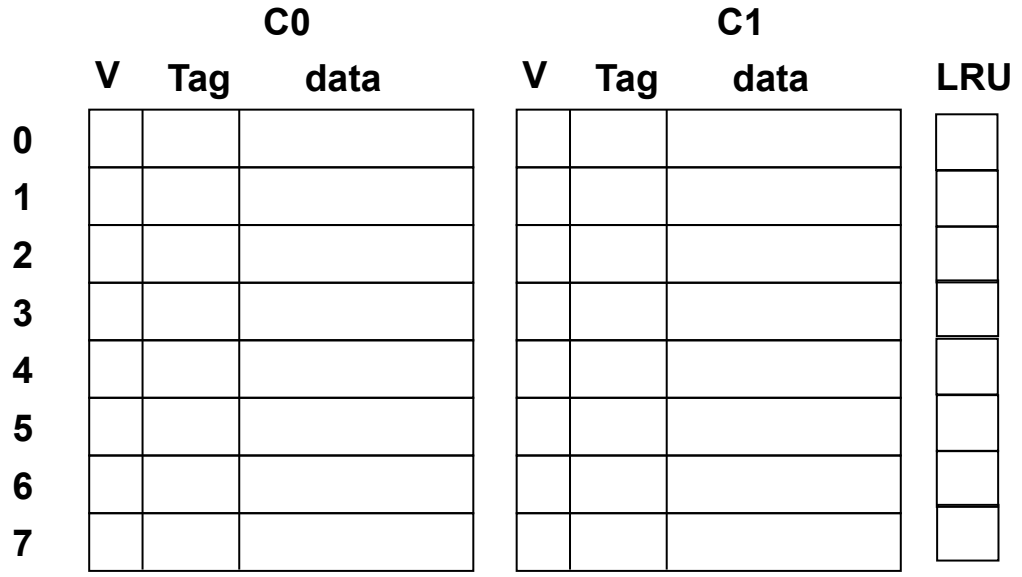
- What kind of cache is this? 2-way set associative
- How many LRU bits do we need? Just one.

What about cache replacement policy?



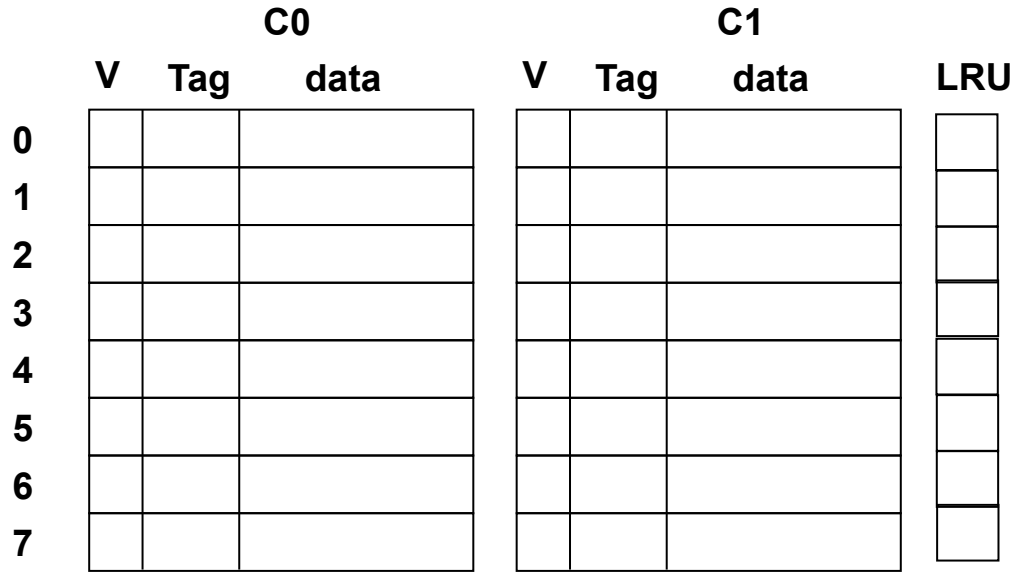
- What kind of cache is this? 2-way set associative
- How many LRU bits do we need? Just one.
- What happens on every memory access?

What about cache replacement policy?



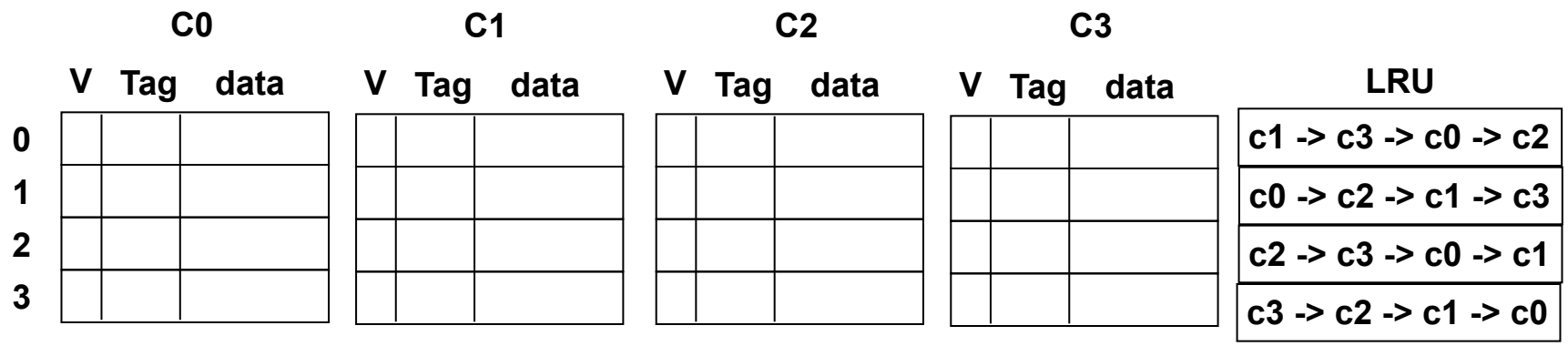
- What kind of cache is this? 2-way set associative
- How many LRU bits do we need? Just one.
- What happens on every memory access? Set LRU to 0/1 if we read from/store in C0/C1

What about cache replacement policy?

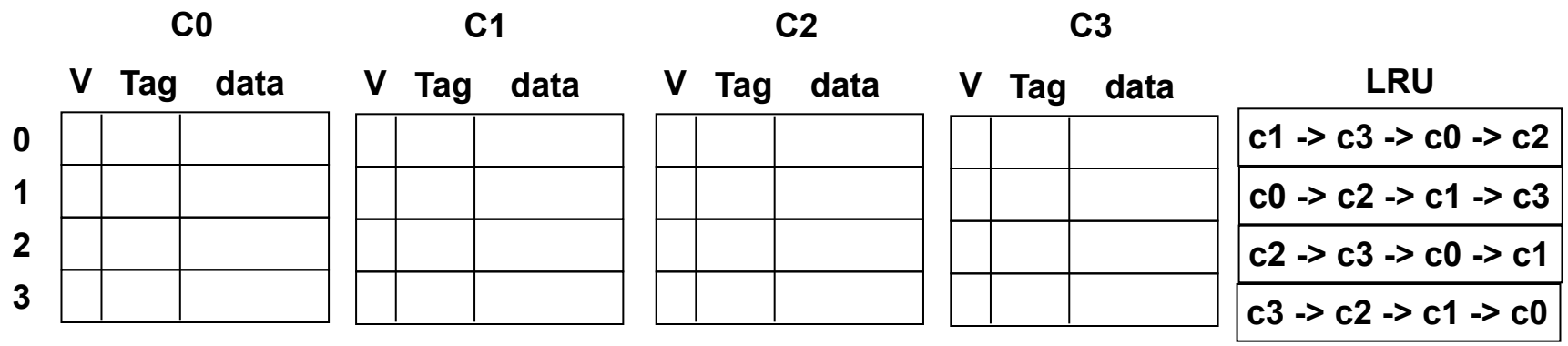


- What kind of cache is this? 2-way set associative
- How many LRU bits do we need? Just one.
- What happens on every memory access? Set LRU to 0/1 if we read from/store in C0/C1
- So what do we do with a 4-way set associative cache?

LRU replacement in a 4-way cache

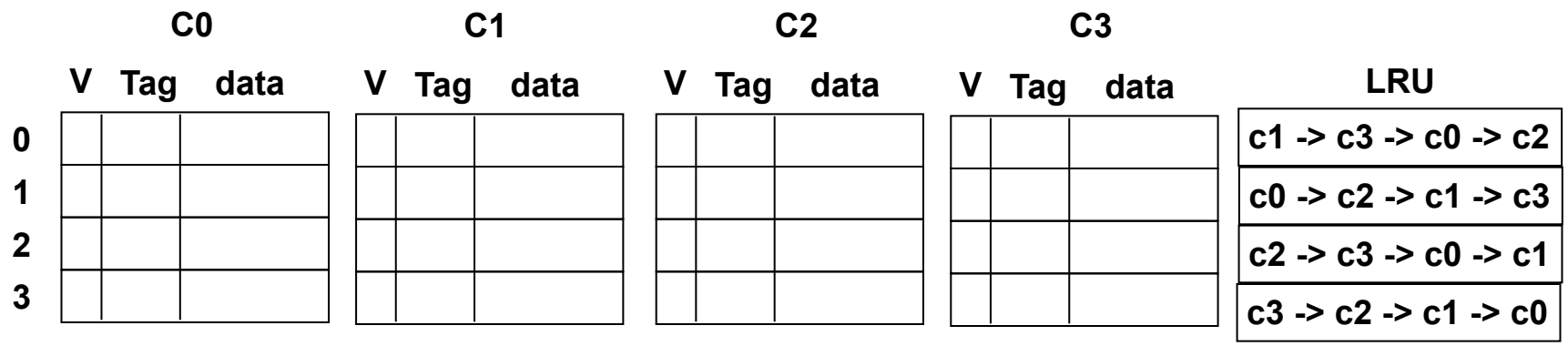


LRU replacement in a 4-way cache



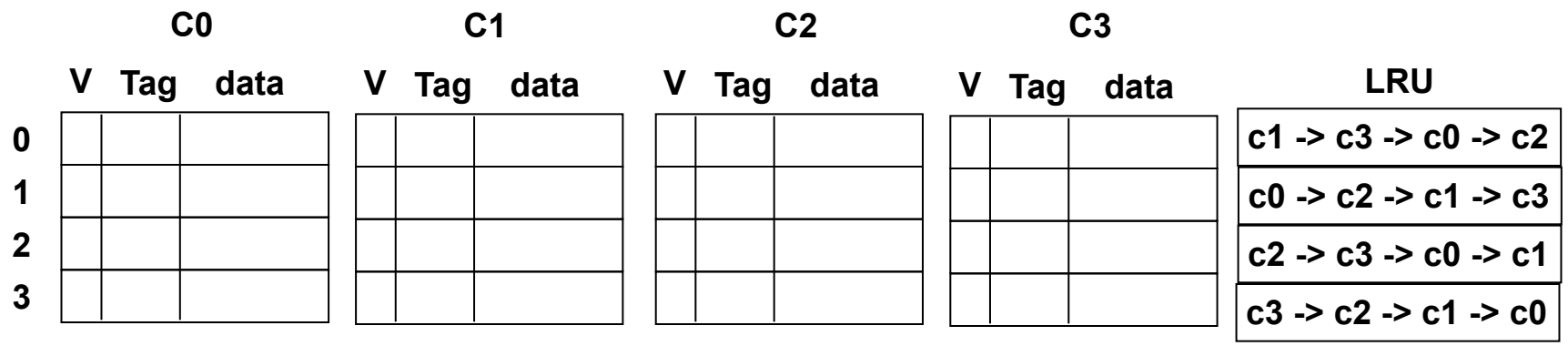
- Do we need a state machine for each cache line?

LRU replacement in a 4-way cache



- Do we need a state machine for each cache line?
- Using as many state machines as the number of rows in the cache is a lot of hardware

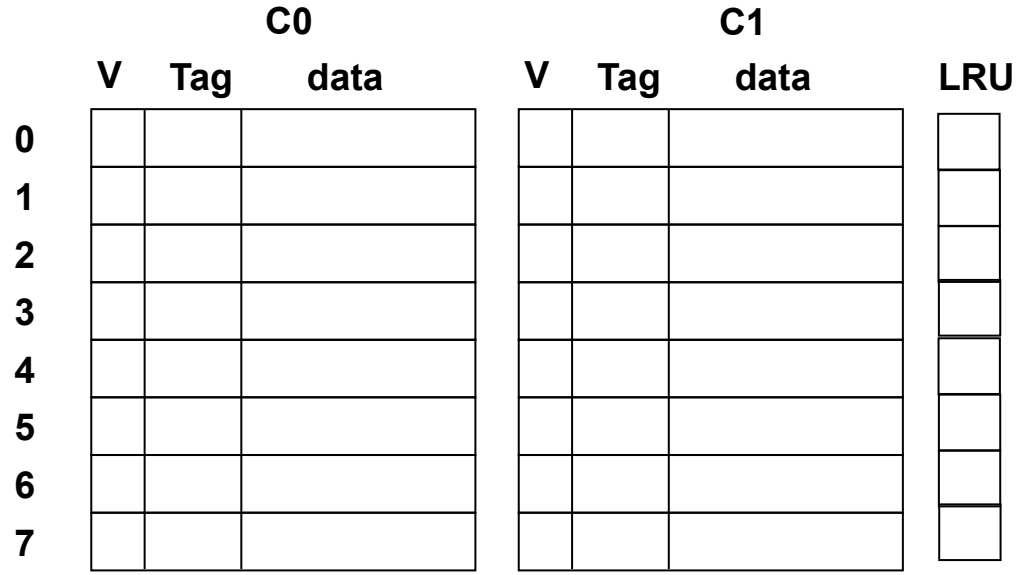
LRU replacement in a 4-way cache



- Do we need a state machine for each cache line?
- Using as many state machines as the number of rows in the cache is a lot of hardware
- Each state machine ➔ 4! States ➔ 5 bit state register

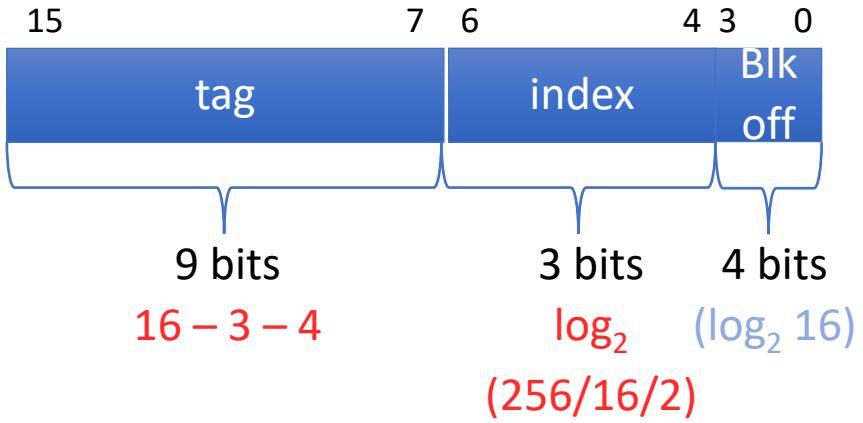
Example

- 16-bit byte-addressable memory
- 2-way 256-byte cache
- 16-byte cache blocks



Example

- 16-bit byte-addressable memory
- 2-way 256-byte cache
- 16-byte cache blocks



	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2							
3							
4							
5							
6							
7							

Example

Access stream from CPU:

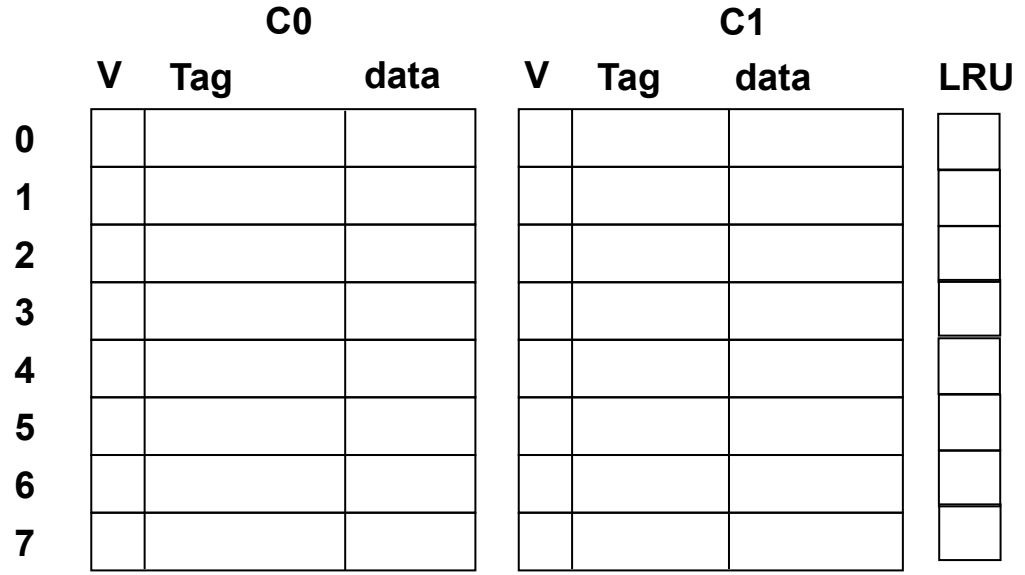
0xF123

0x0252

0x11A0

0xF120

0xB020



0xF123 → 1111 0001 0010 0011

Offset: 0011

Index: 010

Tag: 1111 0001 0 → 0x1E2

Example

Access stream from CPU:

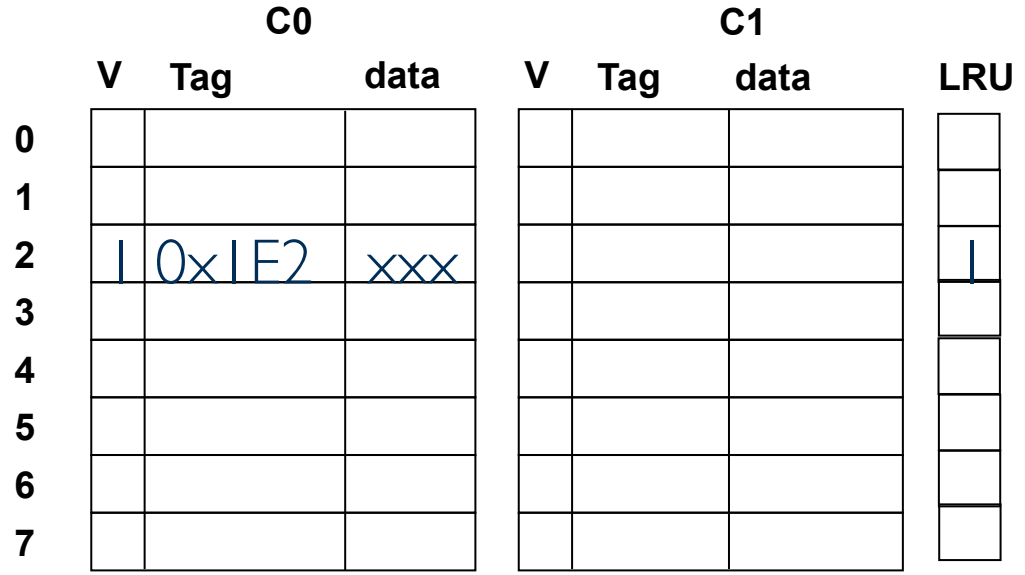
0xF123

0x0252

0x11A0

0xF120

0xB020



0xF123 → 1111 0001 0010 0011

Offset: 0011

Index: 010

Tag: 1111 0001 0 → 0x1E2

Example

Access stream from CPU:

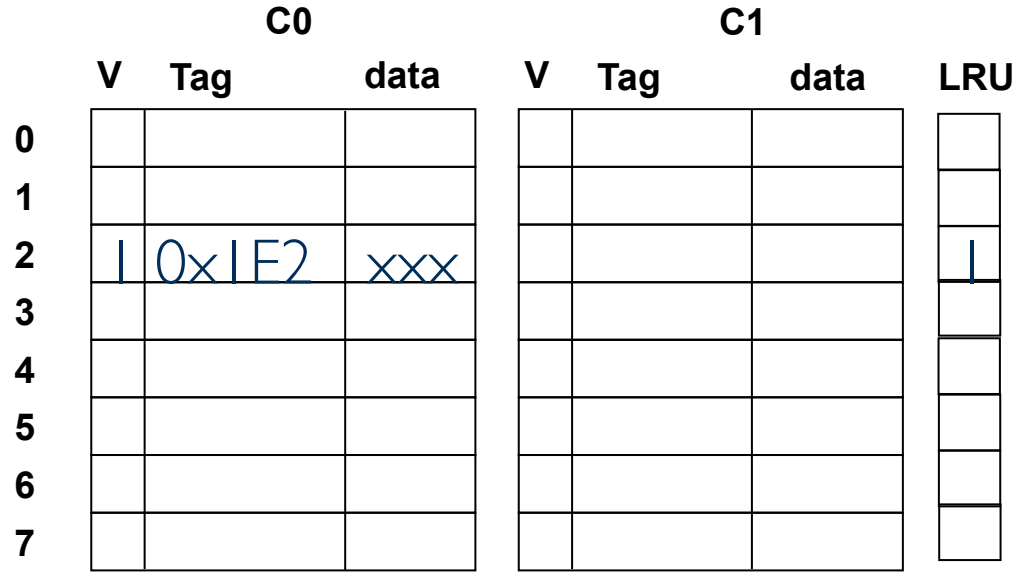
0xF123

0x0252

0x11A0

0xF120

0xB020



0x0252 → 0000 0010 0101 0010

Offset: 0010

Index: 101

Tag: 0000 0010 0 → 0x004

Example

Access stream from CPU:

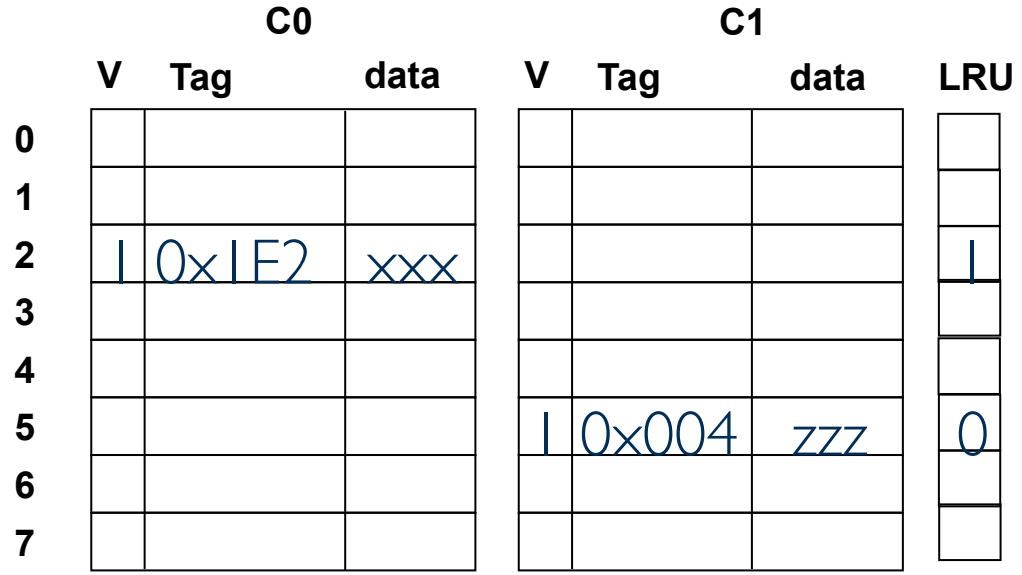
0xF123

0x0252

0x11A0

0xF120

0xB020



0x0252 → 0000 0010 0101 0010

Offset: 0010

Index: 101

Tag: 0000 0010 0 → 0x004

Example

Access stream from CPU:

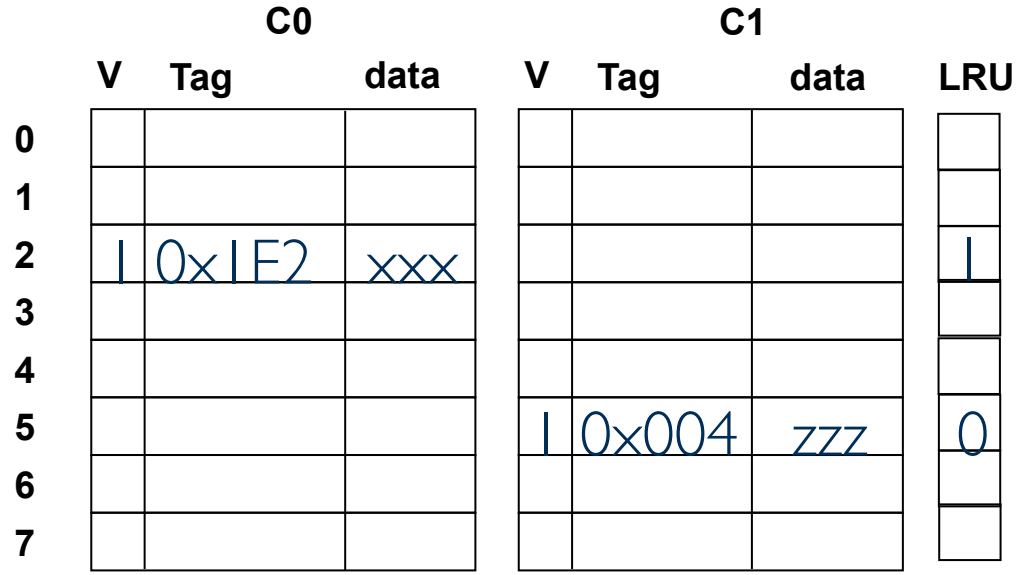
0xF123

0x0252

0x11A0

0xF120

0xB020



0x11A0 → 0001 0001 1010 0000

Offset: 0000

Index: 010

Tag: 0001 0001 1 → 0x023

Example

Access stream from CPU:

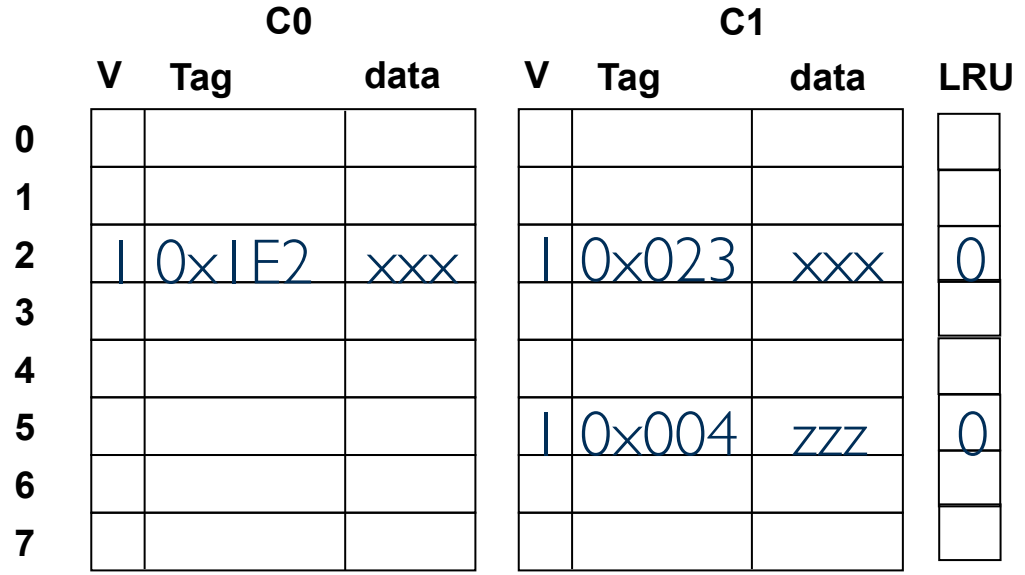
0xF123

0x0252

0x11A0

0xF120

0xB020



0x11A0 → 0001 0001 1010 0000

Offset: 0000

Index: 010

Tag: 0001 0001 1 → 0x023

Example

Access stream from CPU:

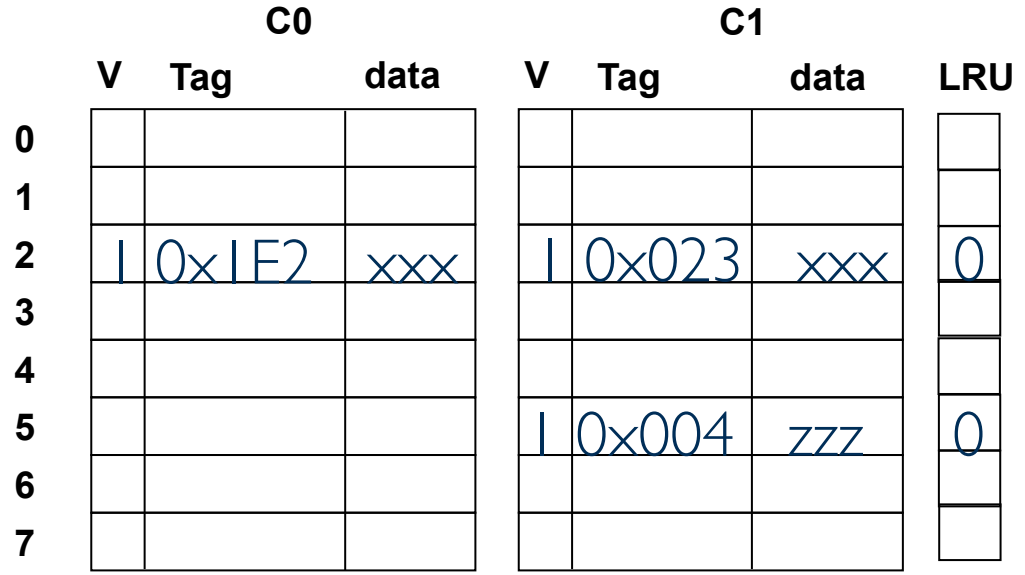
0xF123

0x0252

0x11A0

0xF120

0xB020



	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2	1	0x1E2	xxx	1	0x023	xxx	0
3							
4							
5				1	0x004	zzz	0
6							
7							

0xF120 → 1111 0001 0010 0000

Offset: 0000

Index: 010

Tag: 1111 0001 0 → 0x1E2

Example

Access stream from CPU:

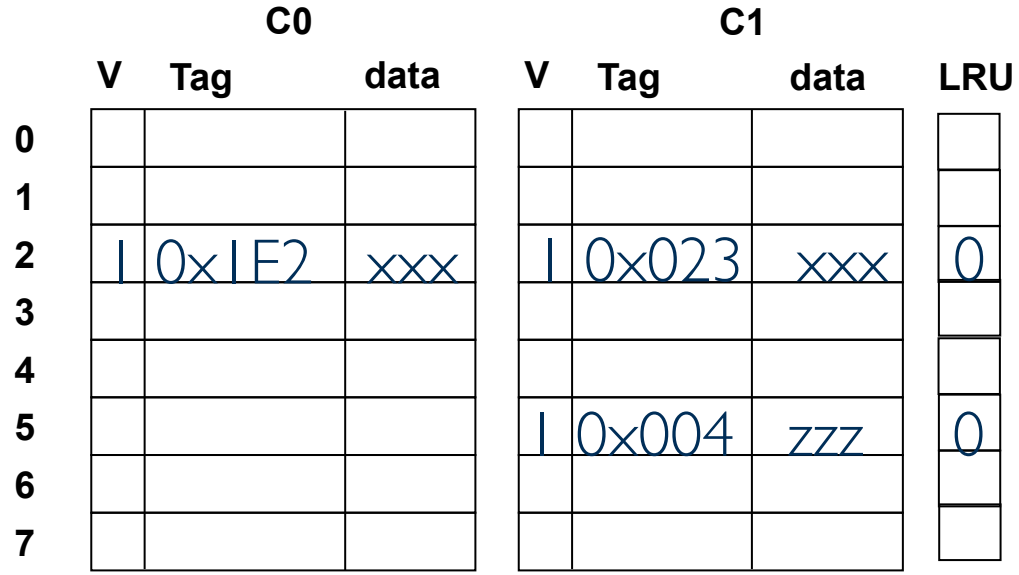
0xF123

0x0252

0x11A0

0xF120

0xB020



0xF120 → 1111 0001 0010 0000

Offset: 0000

Index: 010

Tag: 1111 0001 0 → 0x1E2

Cache Hit!

Example

Access stream from CPU:

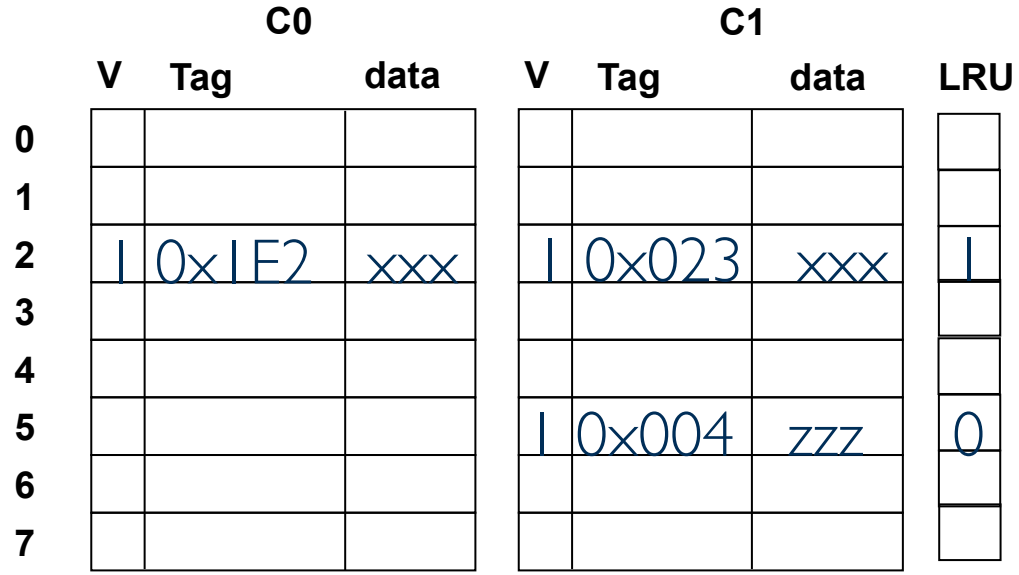
0xF123

0x0252

0x11A0

0xF120

0xB020



0xF120 → 1111 0001 0010 0000

Offset: 0000

Index: 010

Tag: 1111 0001 0 → 0x1E2

Cache Hit!

Example

Access stream from CPU:

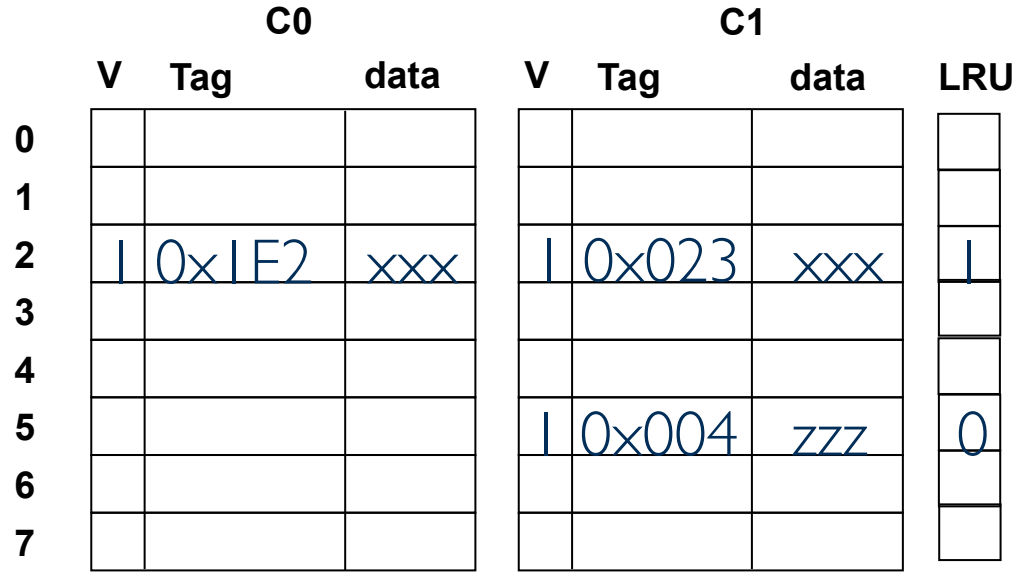
0xF123

0x0252

0x11A0

0xF120

0xB020



	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2	1	0x1E2	xxx	1	0x023	xxx	1
3							
4							
5				1	0x004	zzz	0
6							
7							

0xB020 → 1011 0000 0010 0000

Offset: 0000

Index: 010

Tag: 1011 0000 0 → 0x160

Example

Access stream from CPU:

0xF123

0x0252

0x11A0

0xF120

0xB020

	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2	1	0x1E2	xxx	1	0x023	xxx	1
3							
4							
5				1	0x004	zzz	0
6							
7							

Replace way 1's block!

0xB020 → 1011 0000 0010 0000

Offset: 0000

Index: 010

Tag: 1011 0000 0 → 0x160

Example

Access stream from CPU:

0xF123

0x0252

0x11A0

0xF120

0xB020

	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2	1	0x1F2	xxx	1	0x160	yyy	1
3							
4							
5				1	0x004	zzz	0
6							
7							

Replace way 1's block!

0xB020 → 1011 0000 0010 0000

Offset: 0000

Index: 010

Tag: 1011 0000 0 → 0x160

Example

Access stream from CPU:

0xF123

0x0252

0x11A0

0xF120

0xB020

	C0			C1			
	V	Tag	data	V	Tag	data	LRU
0							
1							
2	1	0x1E2	xxx	1	0x160	yyy	0
3							
4							
5				1	0x004	zzz	0
6							
7							

Replace way 1's block!

0xB020 → 1011 0000 0010 0000

Offset: 0000

Index: 010

Tag: 1011 0000 0 → 0x160

What happens on a context switch?

What happens on a context switch?

Flush user portion

What happens on a context switch?

Flush user portion

Nothing! We're using physical addresses

What happens on a context switch?

- TLB? Flush user portion
- Cache? Nothing! We're using physical addresses

One caveat

Note: If the OS brings in a page from disk directly to a physical page frame (i.e., page replacement) and bypasses the cache, it must flush any cache locations for the previous contents. This isn't a context switch issue, it's an I/O issue: if I/O bypasses the cache, then any cache entries referencing the page that the OS swapped out are definitely invalid.

Session ID: dag2200

On a process context switch

- 0% A. The cache entries for the process being suspended are flushed.
- 0% B. The cache and TLB entries for the process being suspended are flushed
- 0% C. The TLB entries for the process being suspended are flushed
- 0% D. None of the above

Session ID: dag2200

On a process context switch

- 0% A. The cache entries for the process being suspended are flushed.
- 0% B. The cache and TLB entries for the process being suspended are flushed
- 0% C. The TLB entries for the process being suspended are flushed
- 0% D. None of the above

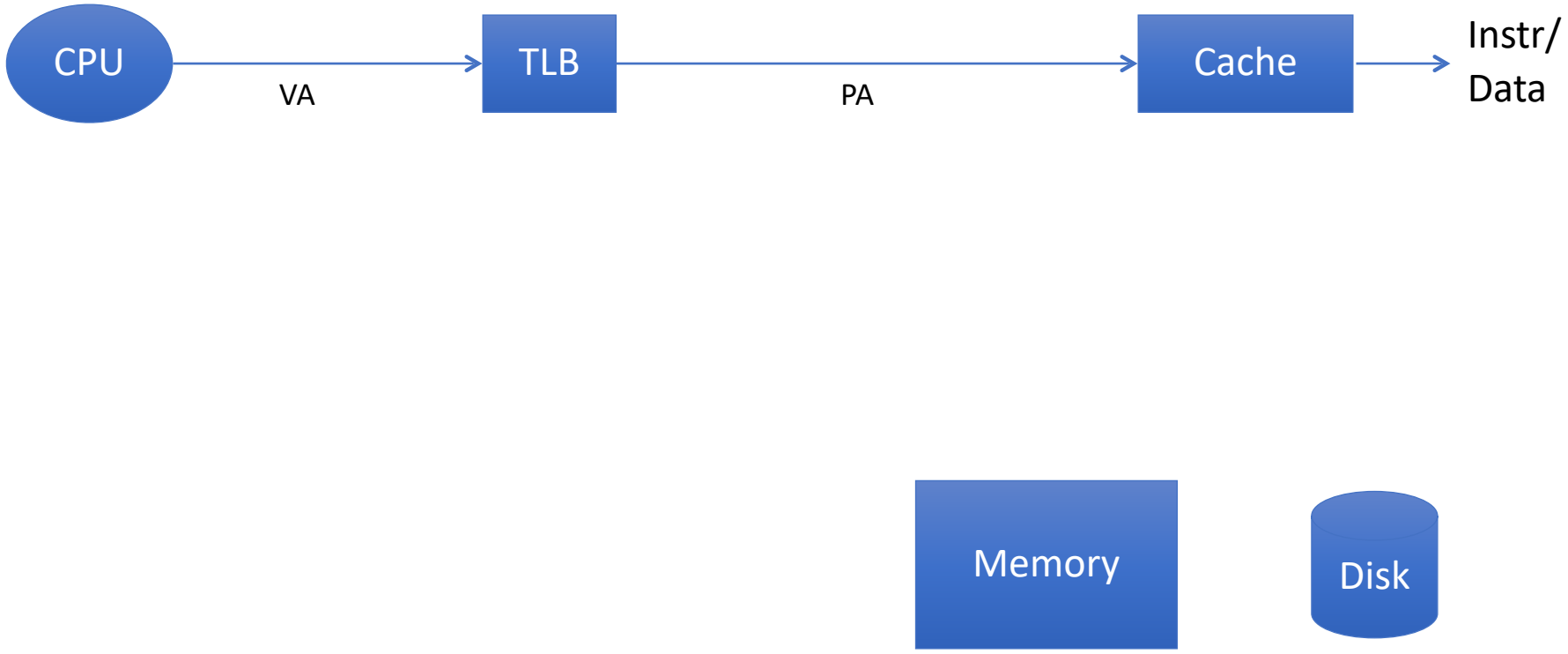
The caches we've studied so far hold physical addresses (address translation has already occurred before the cache gets involved), then no flush is needed on context switch.
If the cache holds virtual addresses, then flushing cache entries from the page is required.

Session ID: dag2200

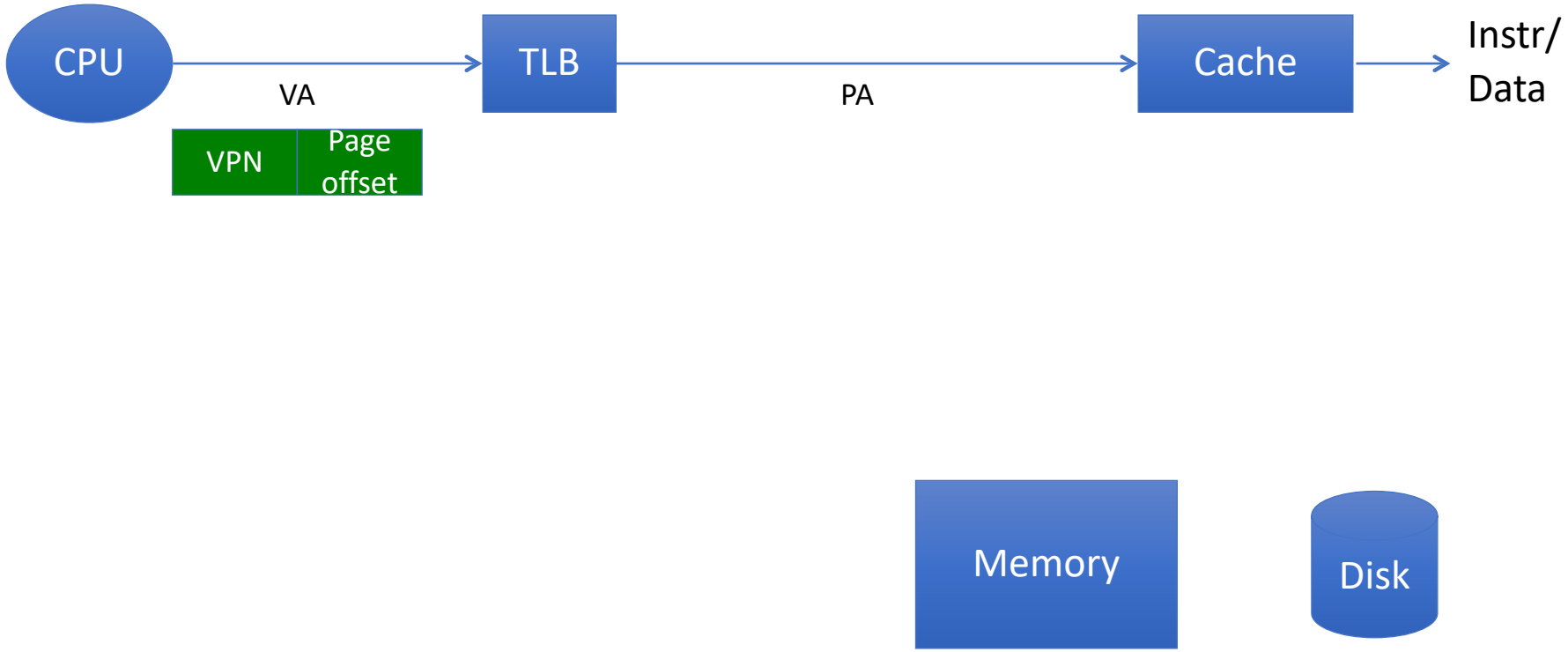
When a page is evicted from a page frame in memory

A. The cache entries for the evicted page are flushed.
B. The cache and TLB entries for the evicted page are flushed
C. The TLB entries for the evicted page are flushed
D. None of the above

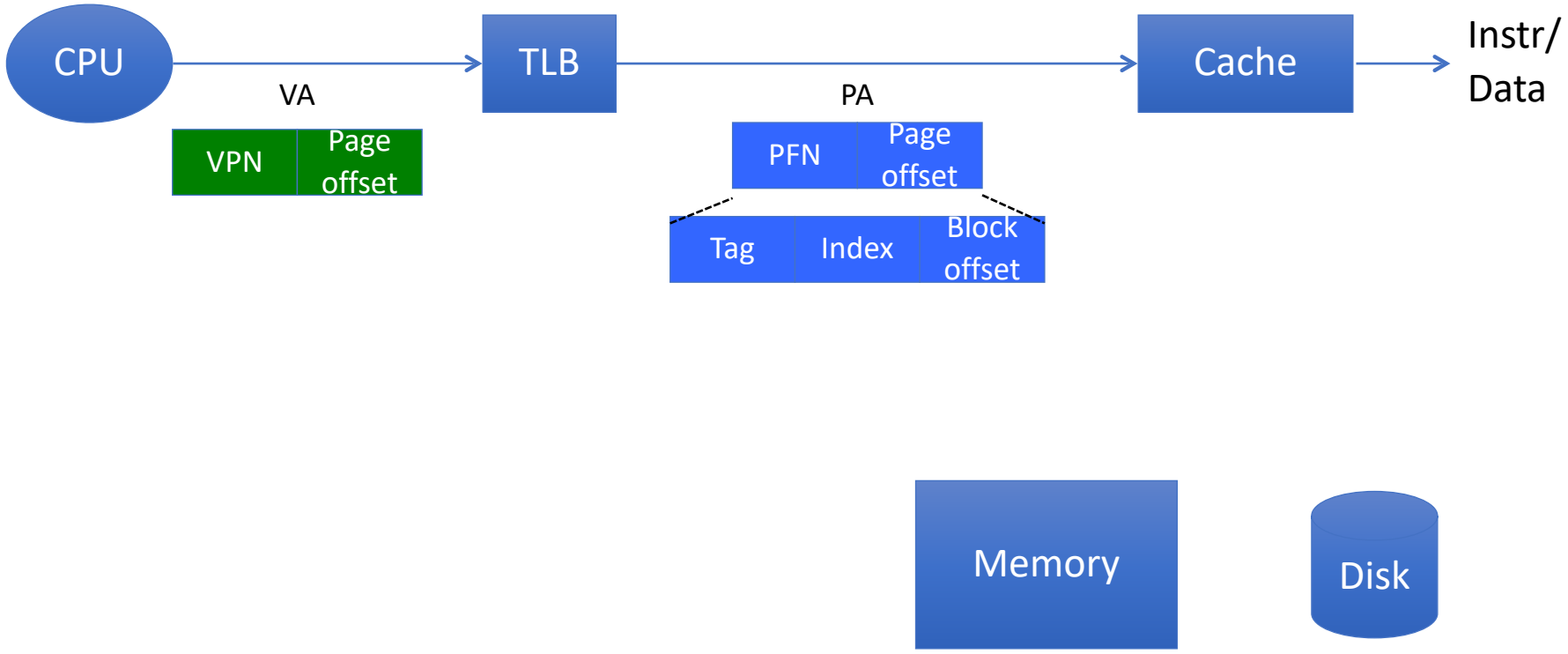
Putting cache and VM together



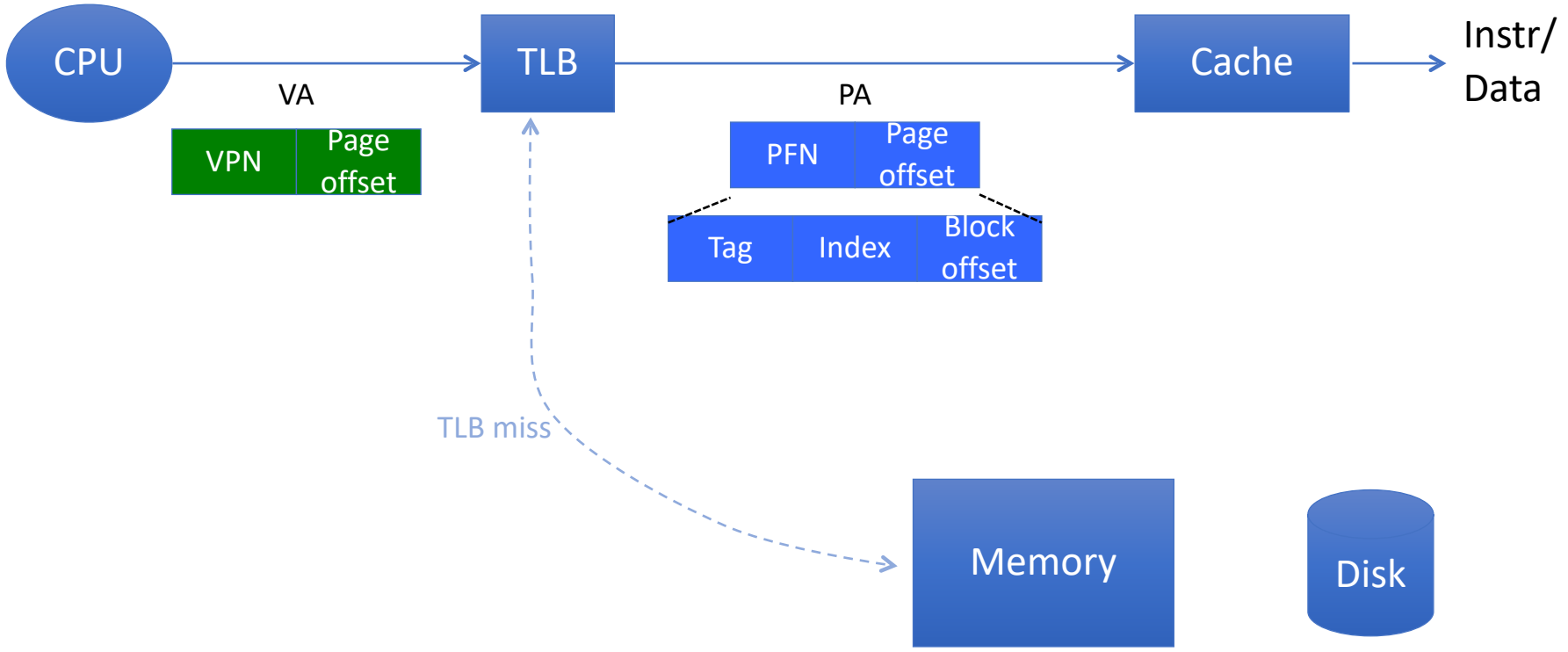
Putting cache and VM together



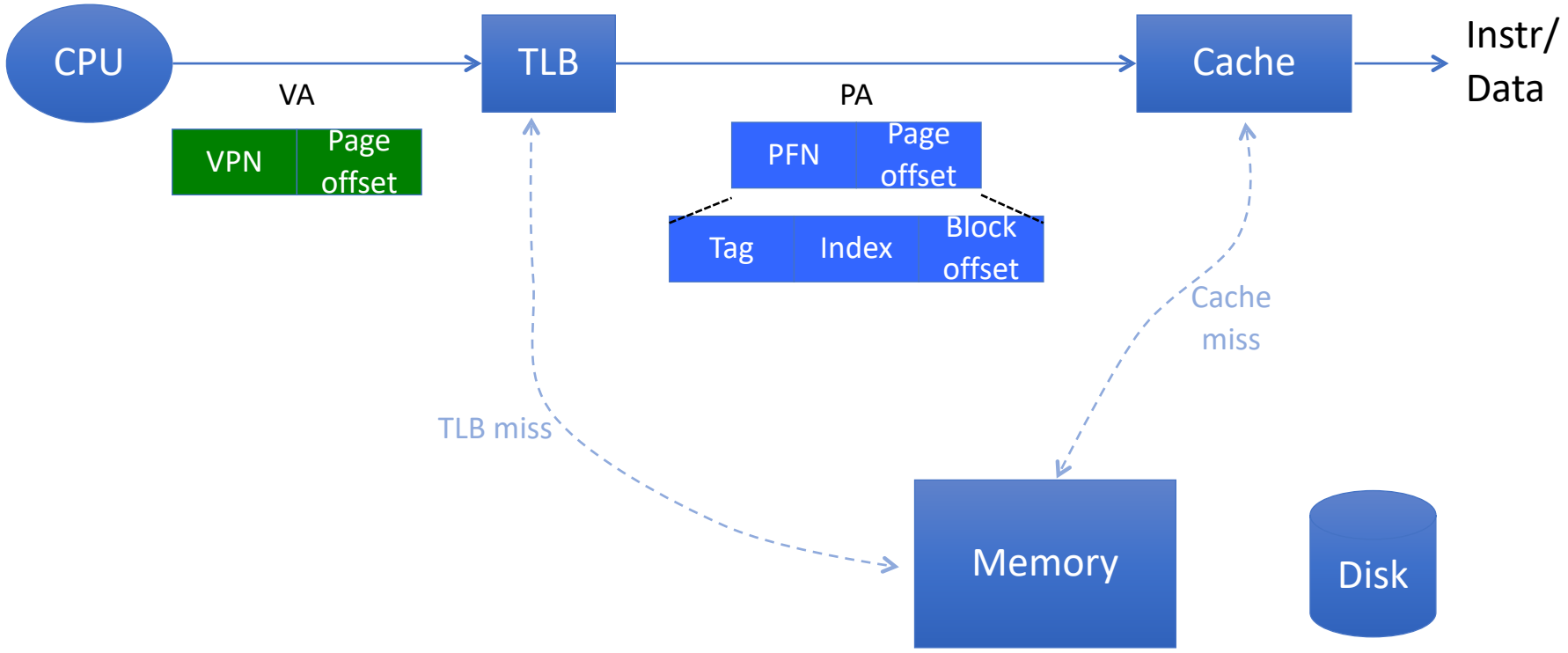
Putting cache and VM together



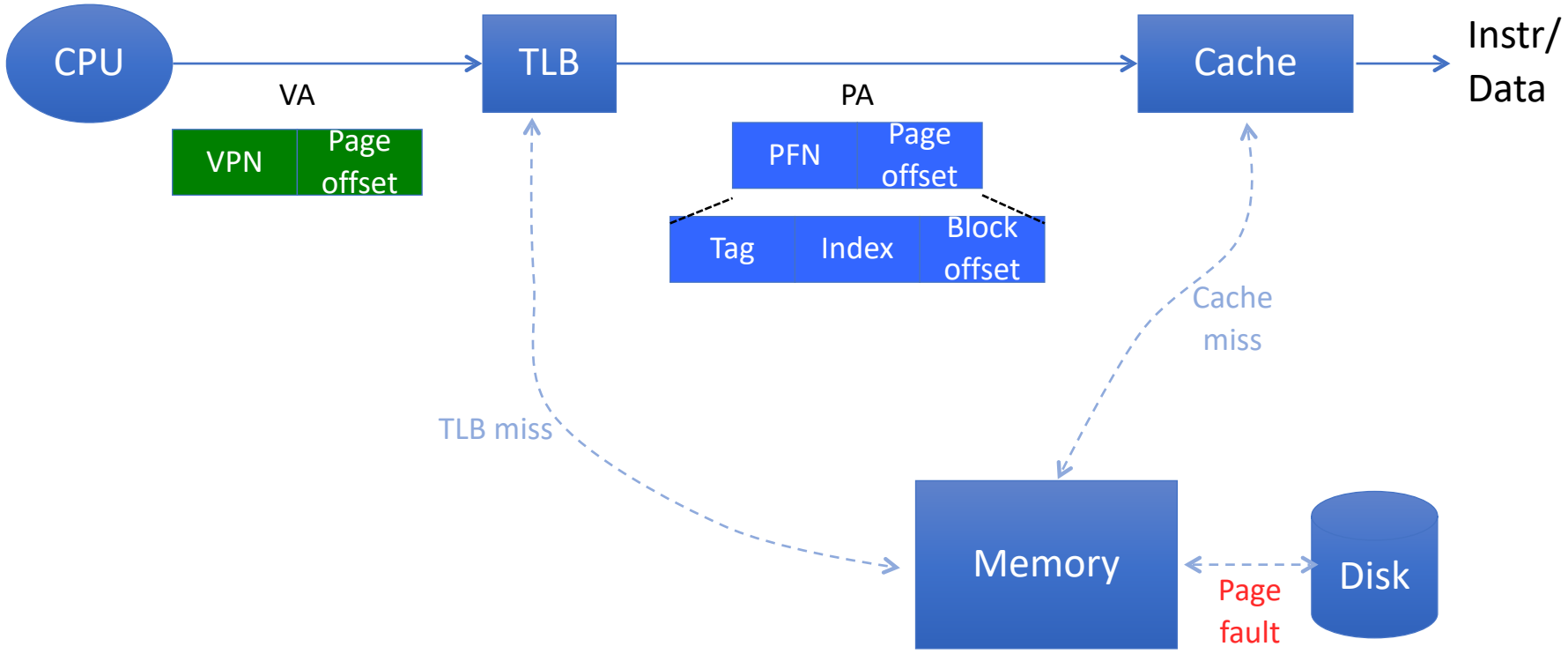
Putting cache and VM together



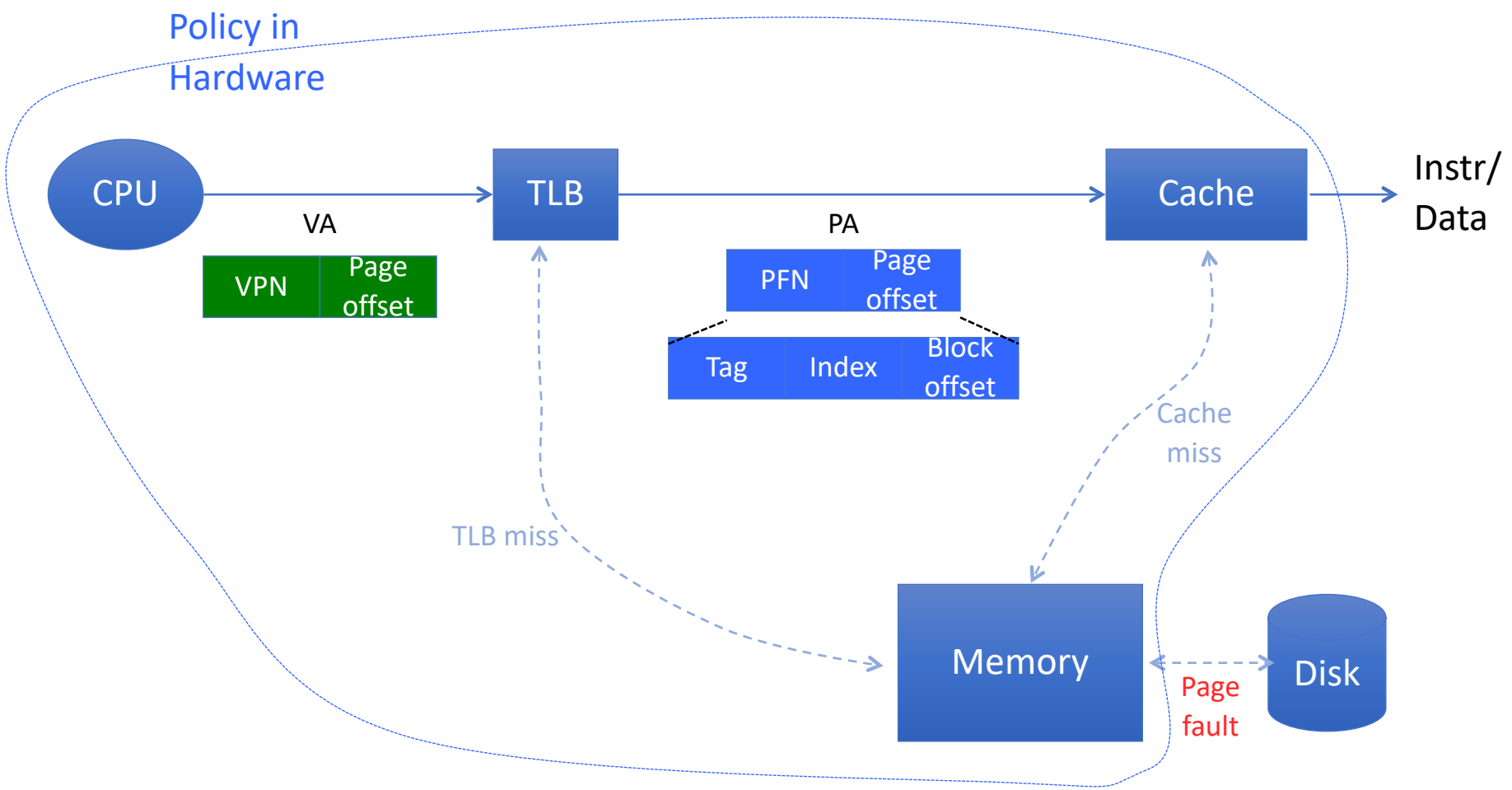
Putting cache and VM together



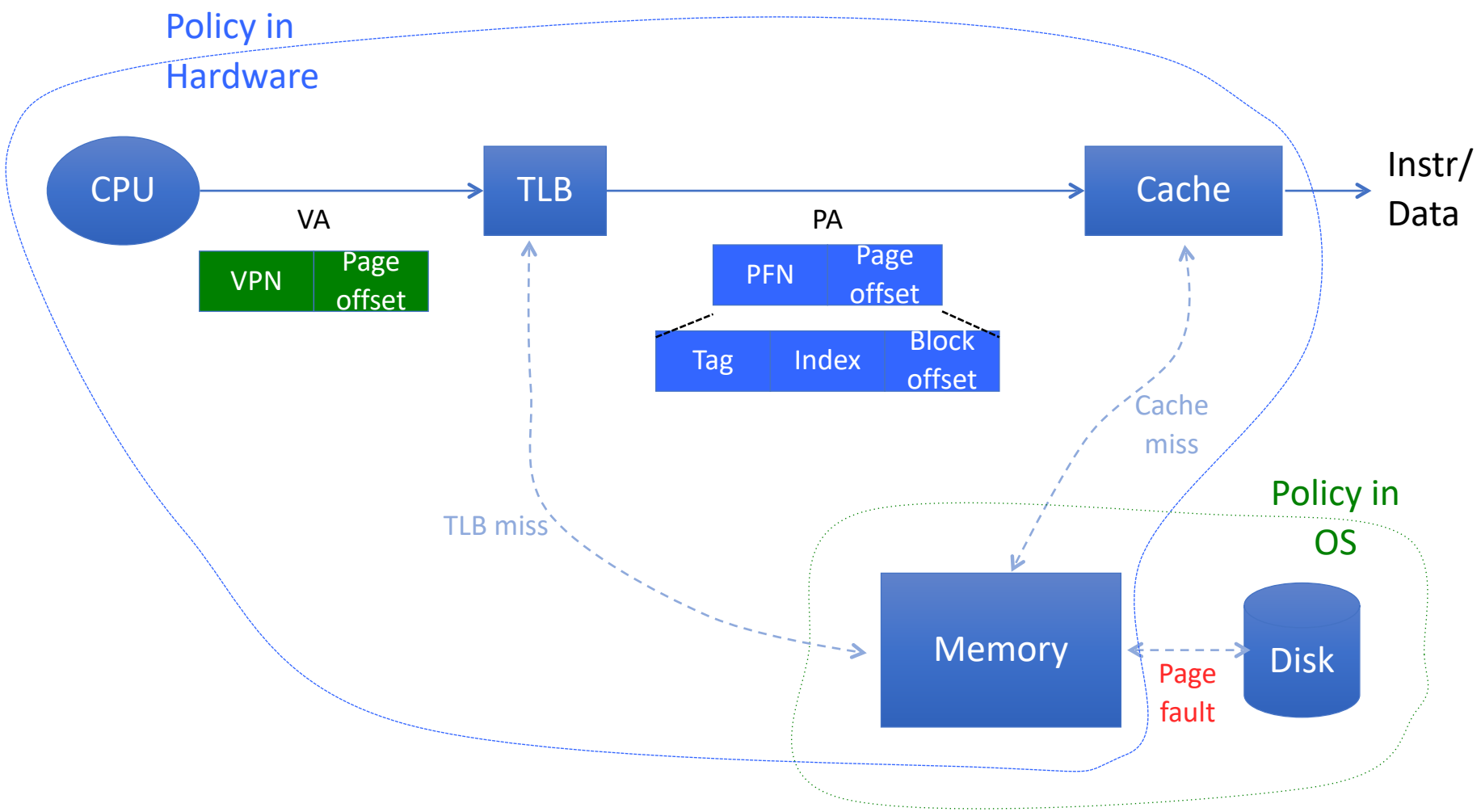
Putting cache and VM together



Putting cache and VM together

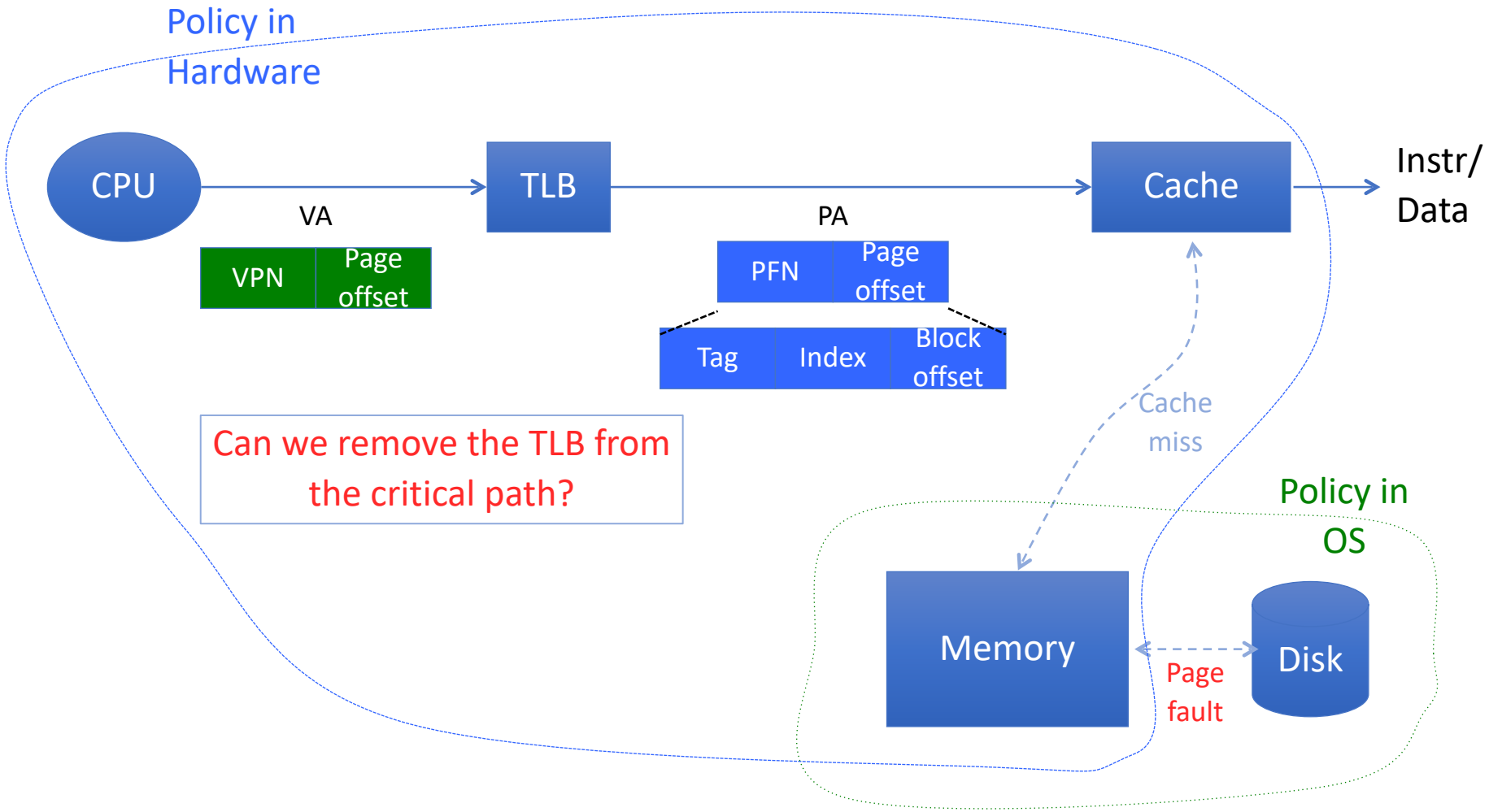


Putting cache and VM together

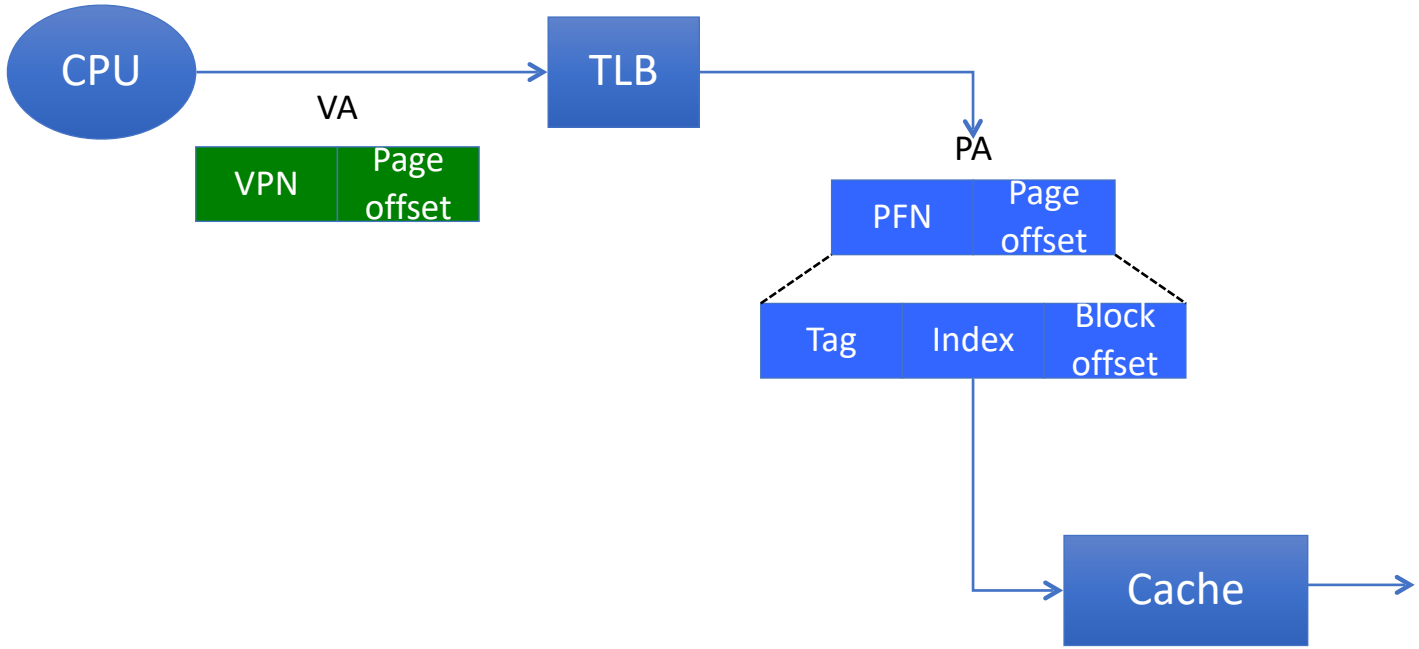


Adding some speed

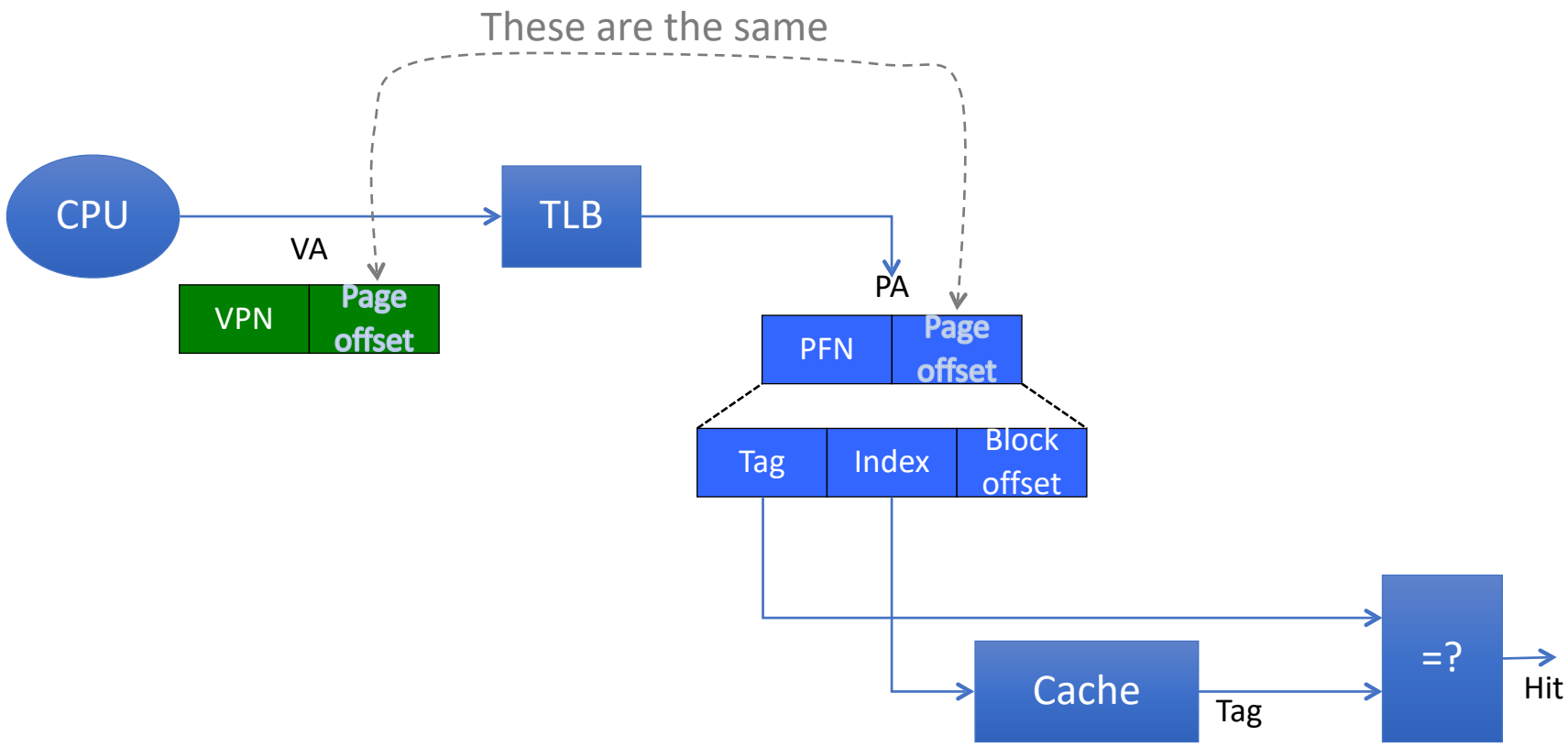
- TLB access is on the critical path of cache access ➔ increased clock cycle time



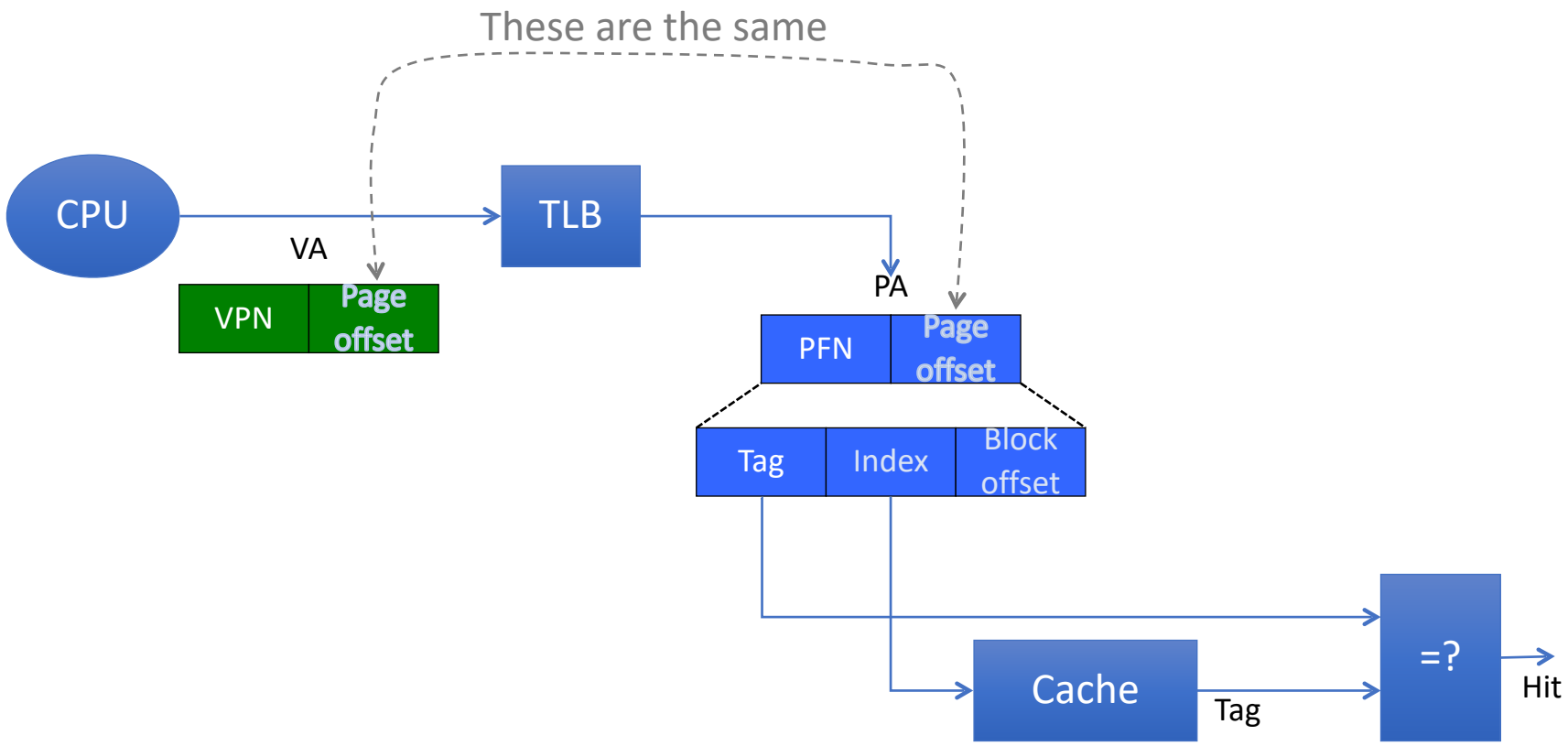
Recall how TLB and Cache work together



Recall


These are the same
CPU
VA
TLB
VPN
Page offset
PA
PFN
Page offset
Tag
Index
Block offset
Cache
Tag
=?
Hit

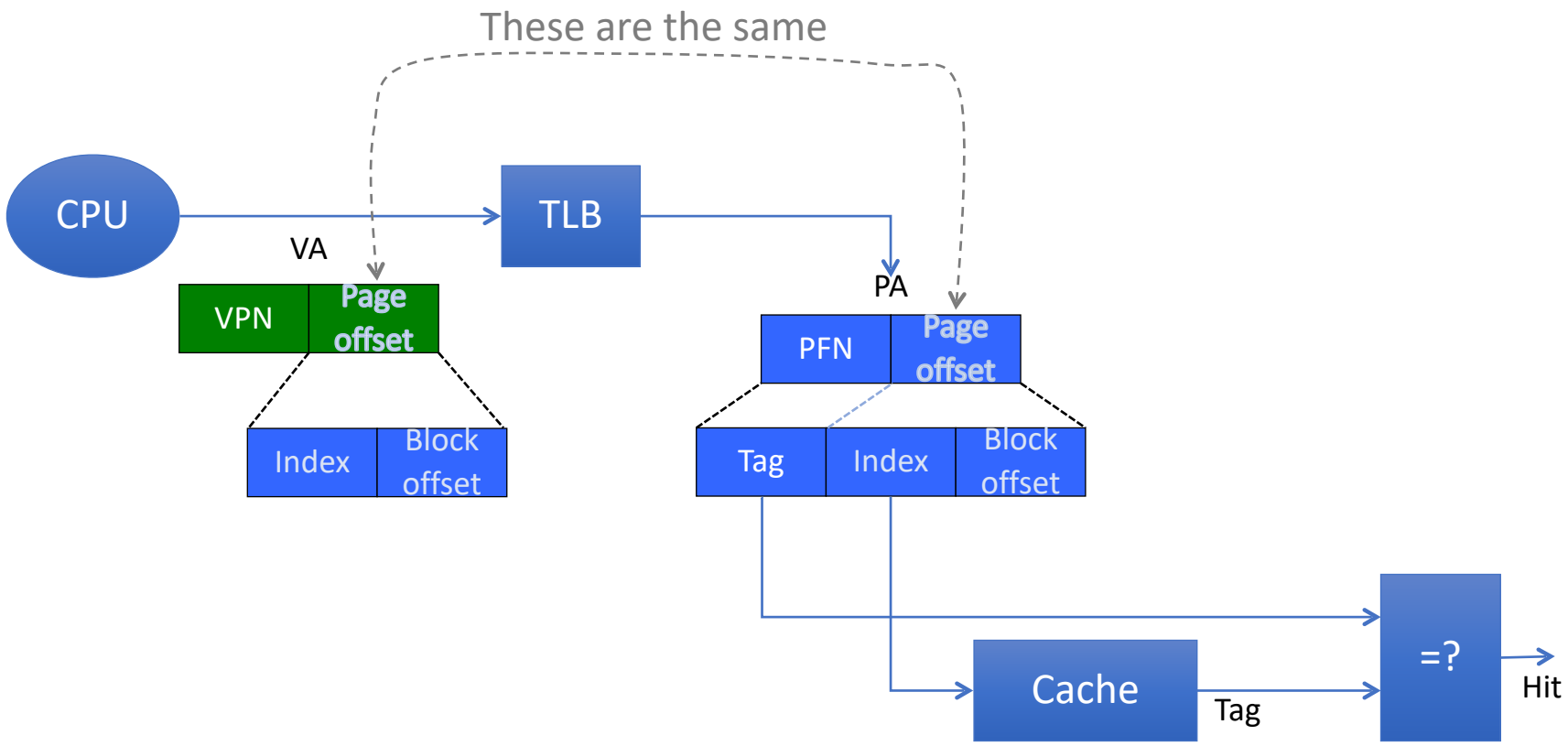
Make the cache index ≤ page offset?



What if we arrange to make the Index + Block offset bits fall within Page offset?

Then we can get the cache index from the VA to start the cache read without waiting for the TLB!

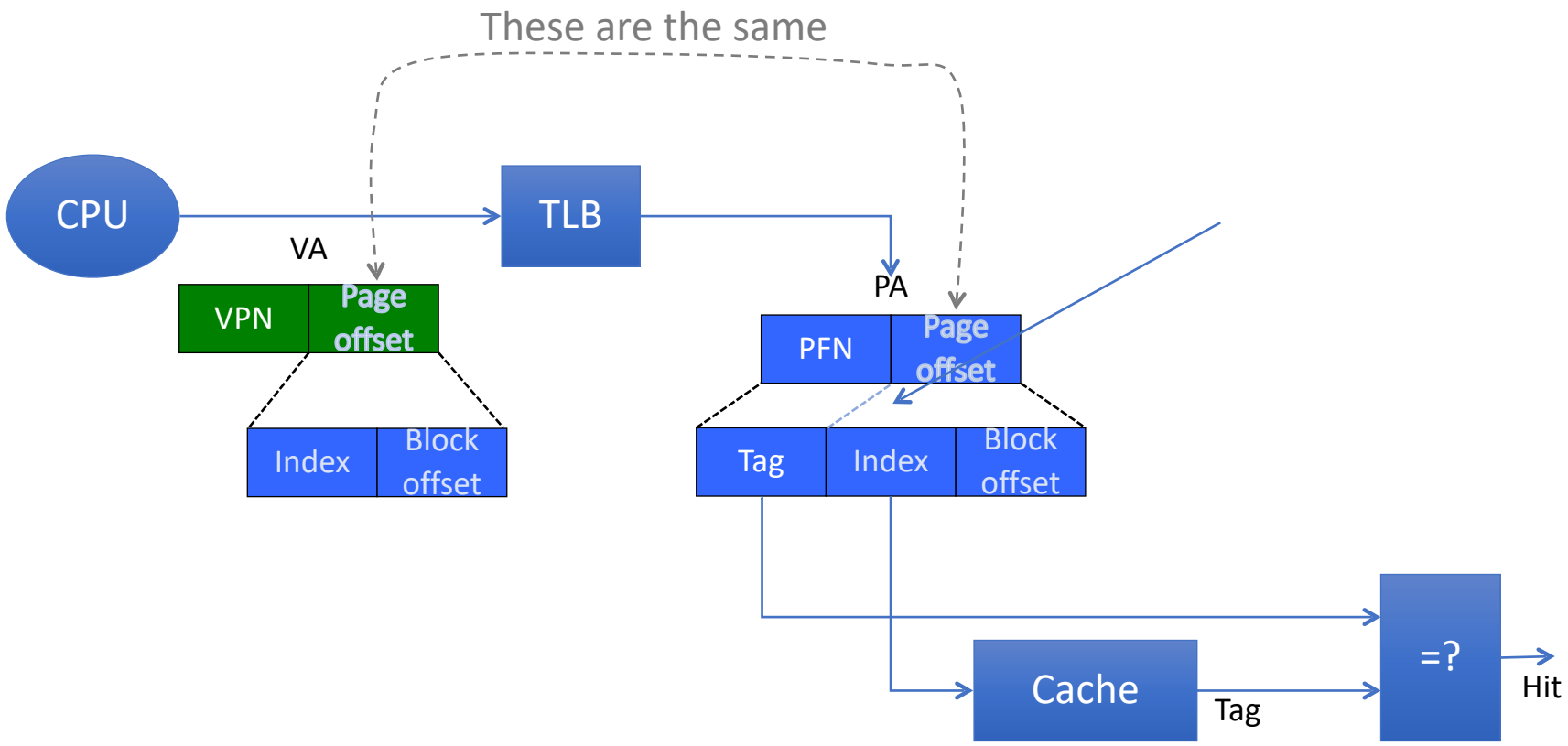
Make the cache index ≤ page offset?



What if we arrange to make the Index + Block offset bits fall within Page offset?

Then we can get the cache index from the VA to start the cache read without waiting for the TLB!

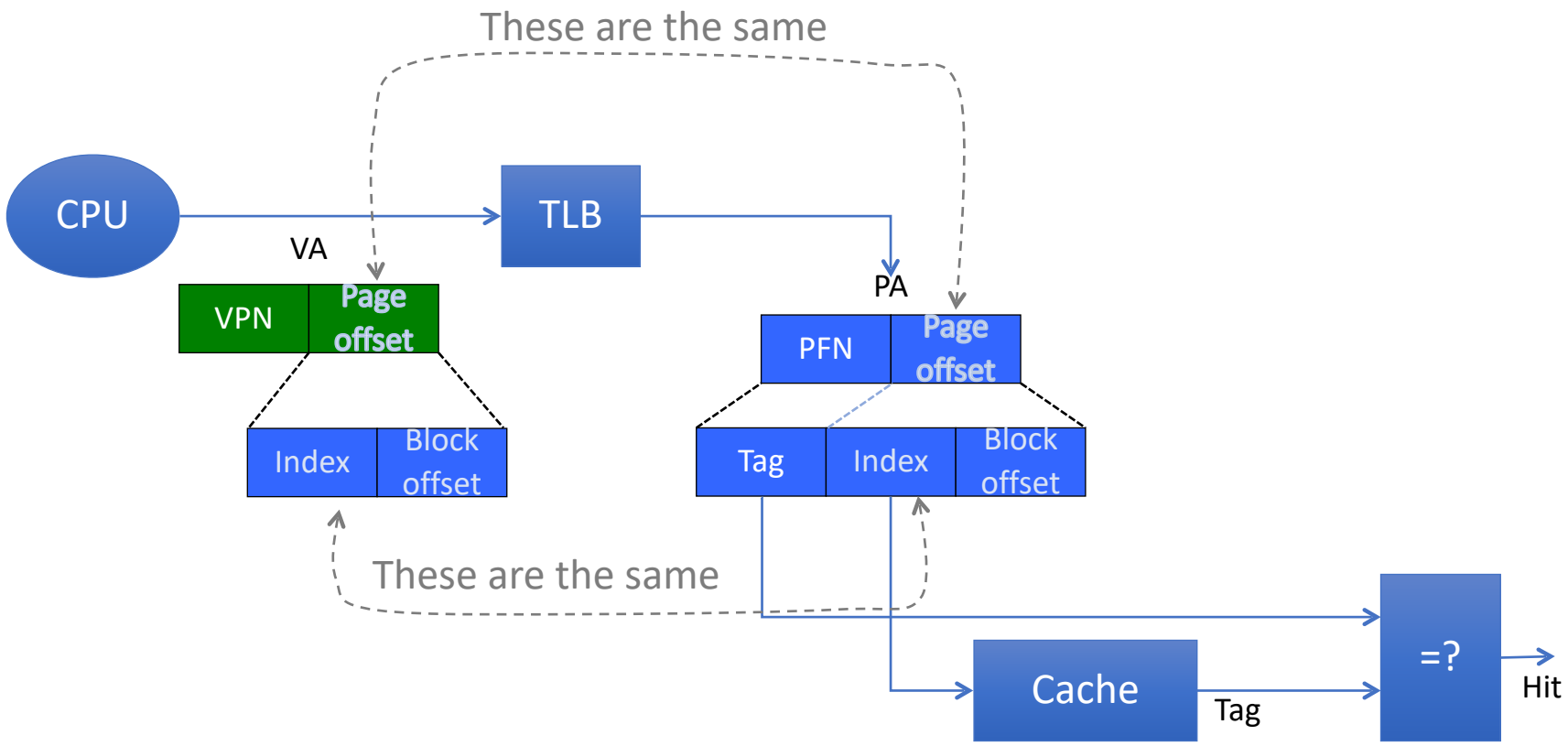
Make the cache index ≤ page offset?



What if we arrange to make the Index + Block offset bits fall within Page offset?

Then we can get the cache index from the VA to start the cache read without waiting for the TLB!

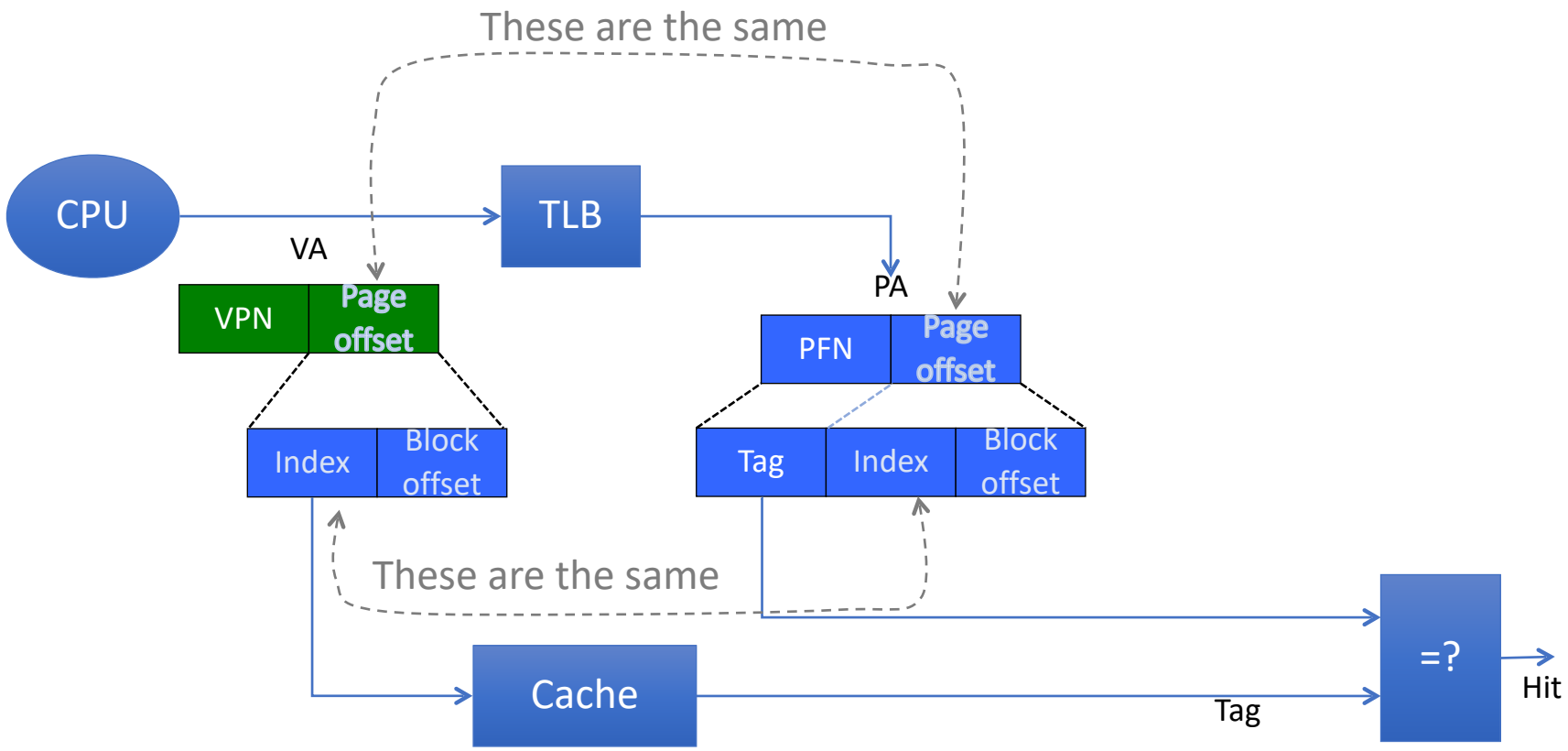
Now Index is the same for VA and PA!



What if we arrange to make the Index + Block offset bits fall within Page offset?

Then we can get the cache index from the VA to start the cache read without waiting for the TLB!

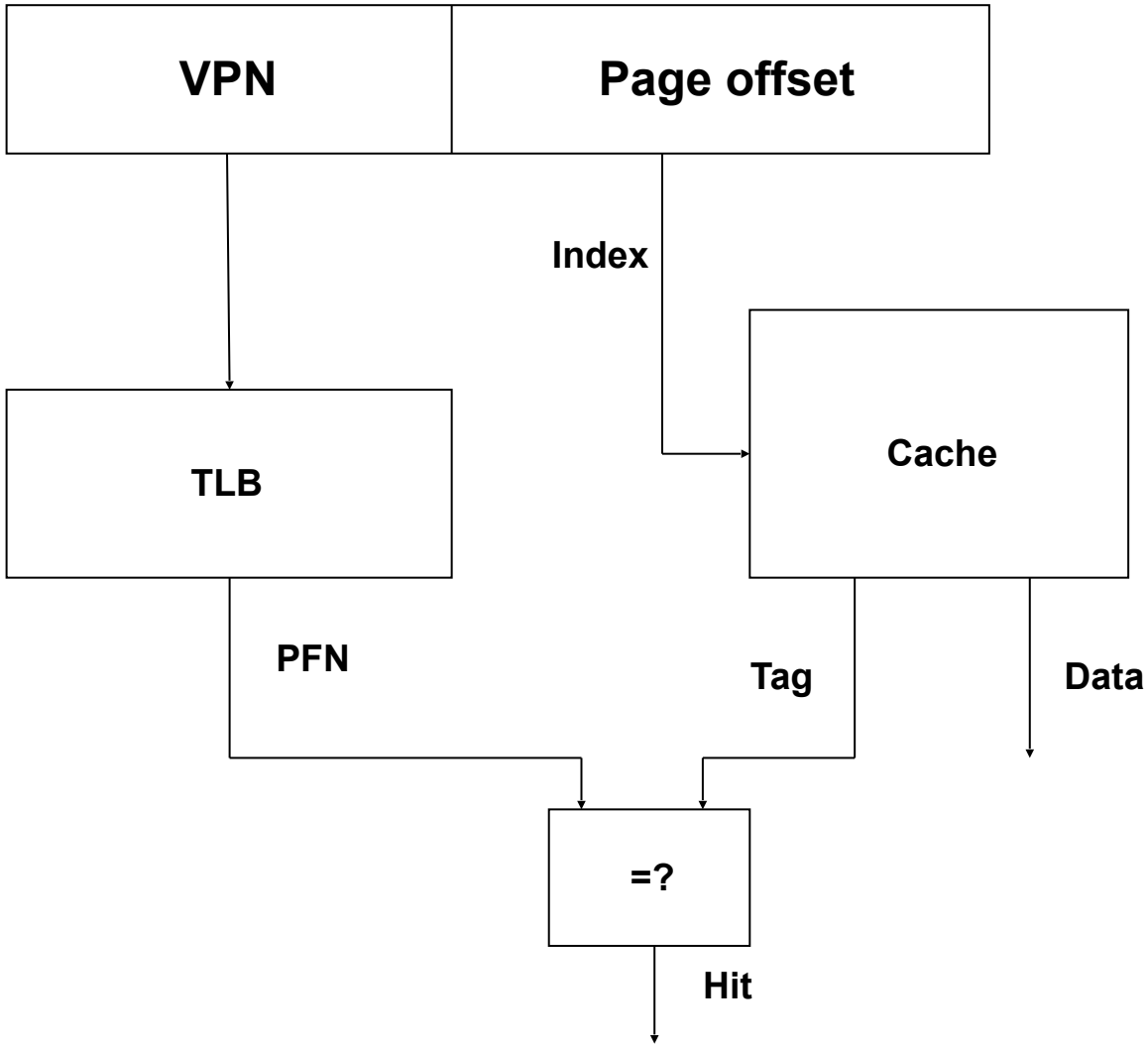
Cache and TLB access can start in parallel!



What if we arrange to make the Index + Block offset bits fall within Page offset?

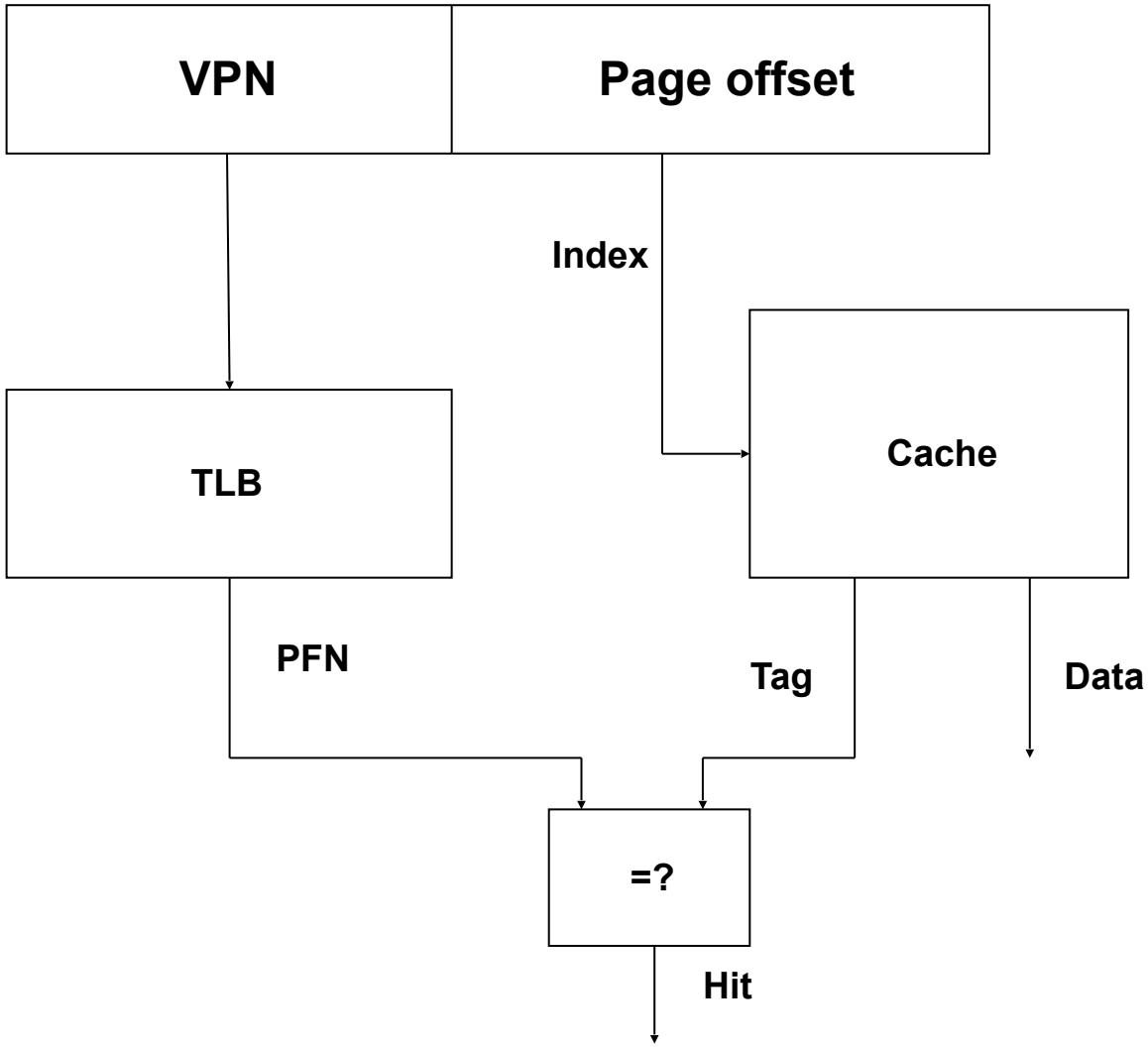
Then we can get the cache index from the VA to start the cache read without waiting for the TLB!

Virtually indexed physically tagged (VIPT) cache



Virtually indexed physically tagged (VIPT) cache

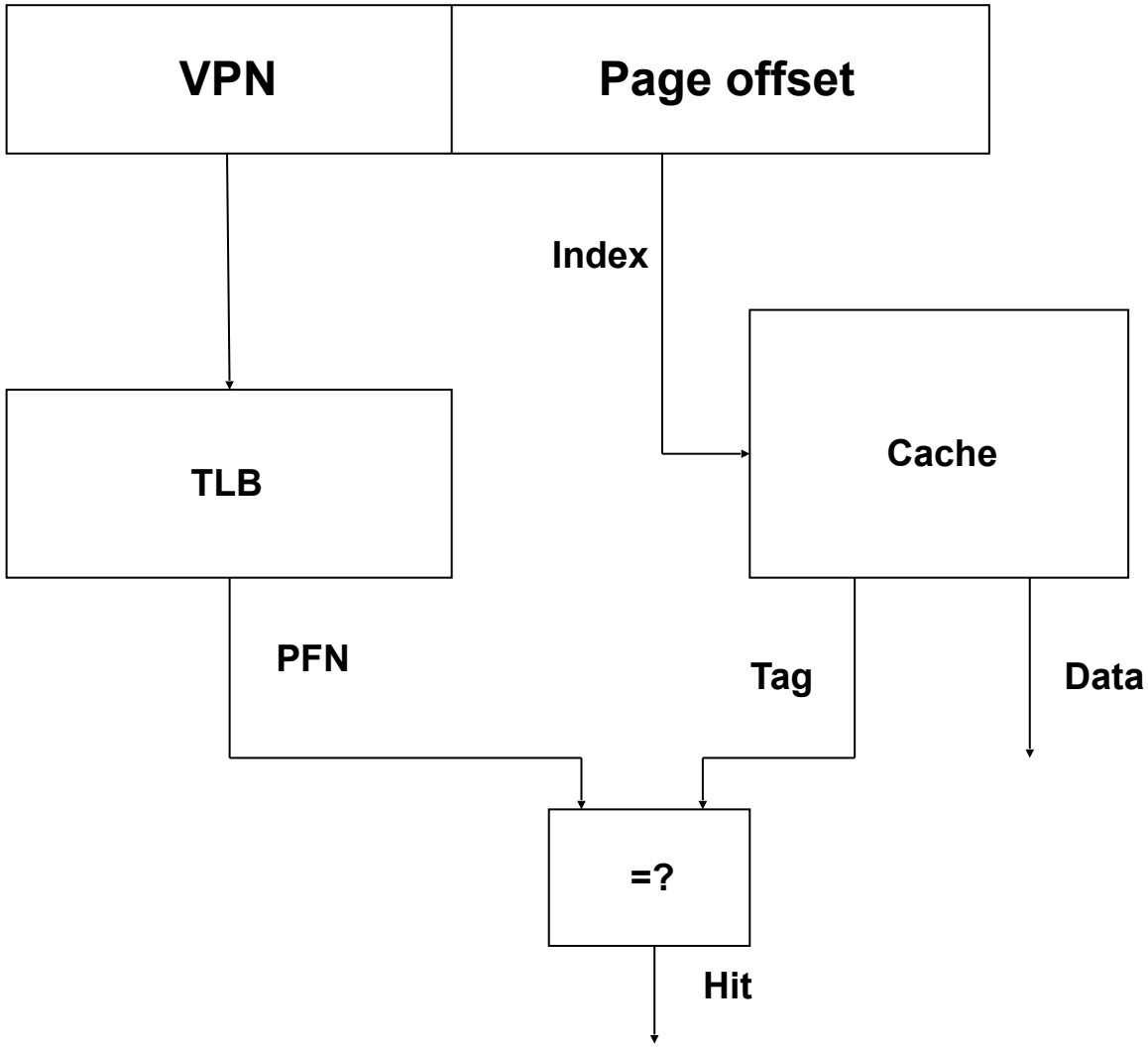
How to get TLB out of the critical path?



Virtually indexed physically tagged (VIPT) cache

How to get TLB out of the critical path?

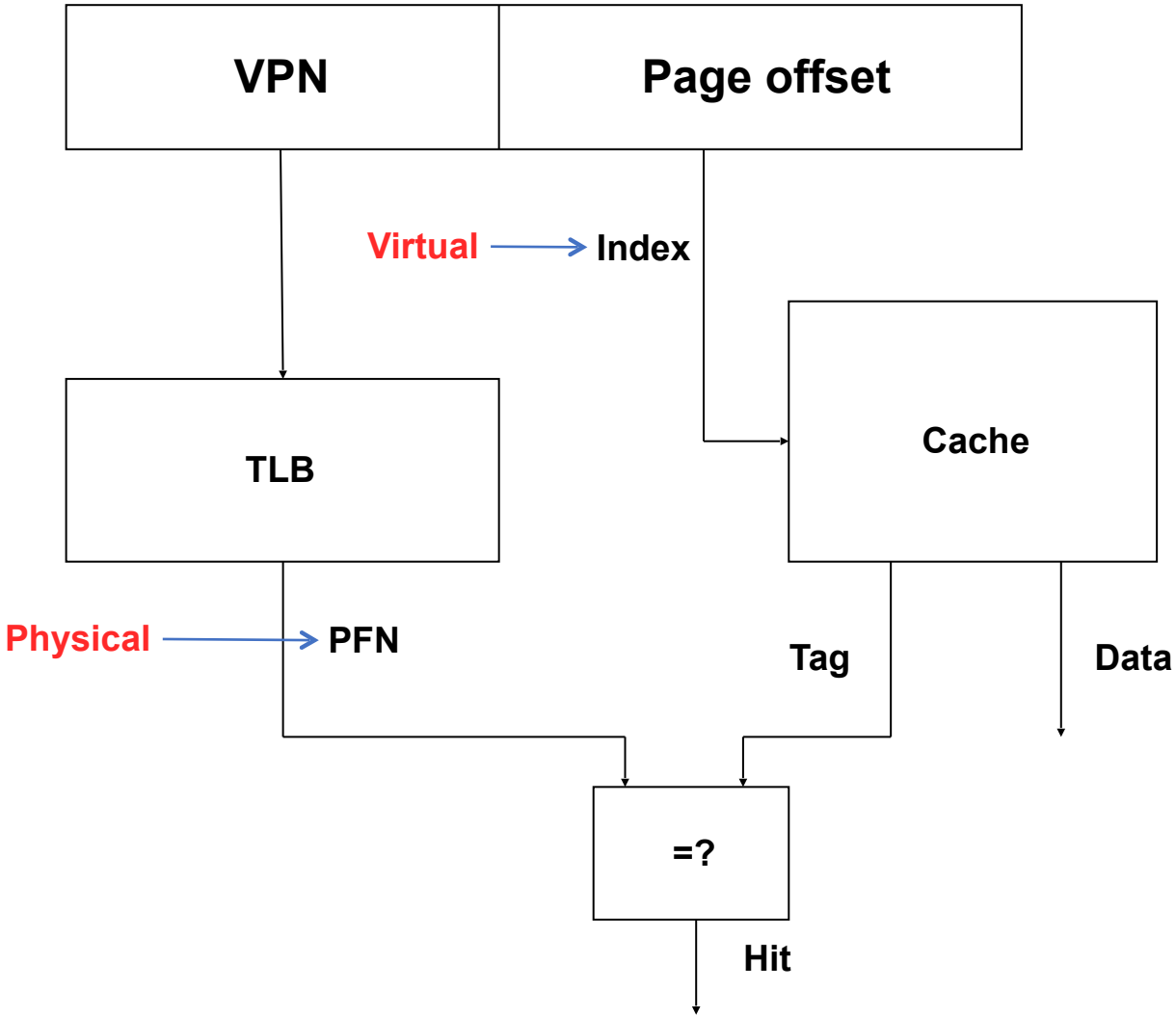
TLB & cache access in parallel



Virtually indexed physically tagged (VIPT) cache

How to get TLB out of the critical path?

TLB & cache access in parallel



Session ID: dag2200

In a virtually indexed physically tagged cache

16% A. I just want the participation credit

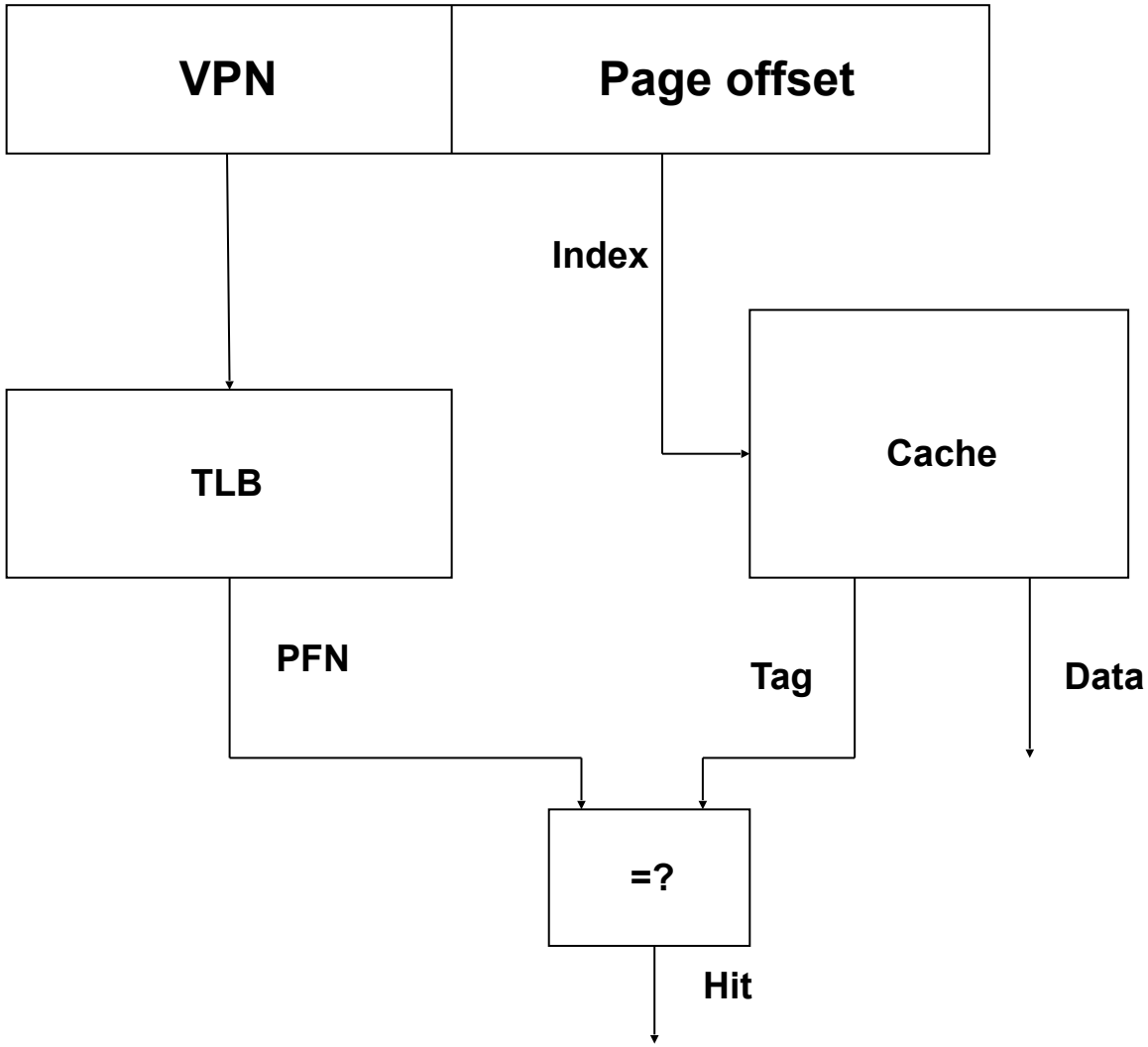
74% B. The TLB and cache are accessed in parallel

9% C. The TLB and cache are accessed sequentially

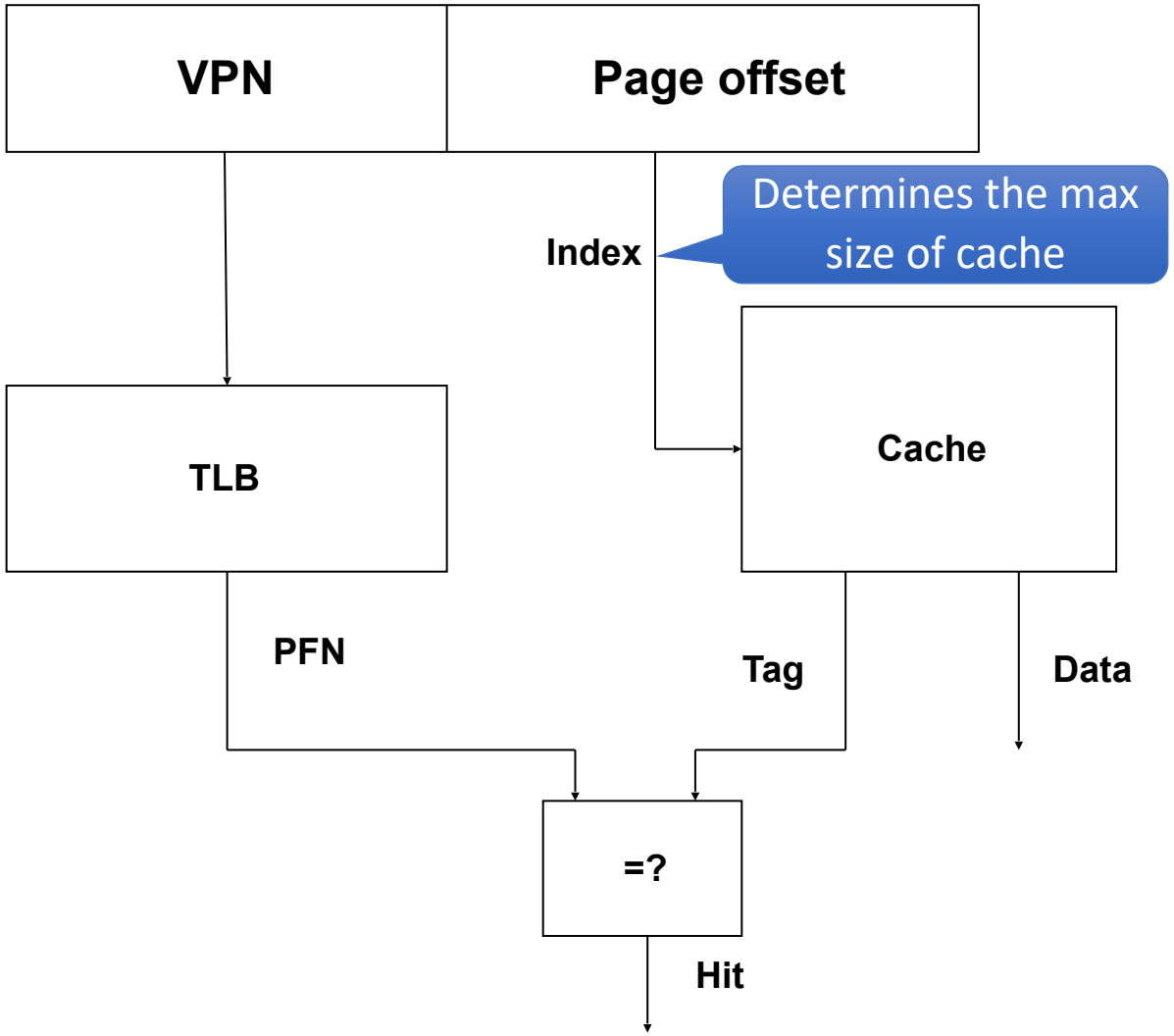
0% D. The TLB and cache are accessed at random

1% E. None of the above

Is there a limitation in VIPT cache size?

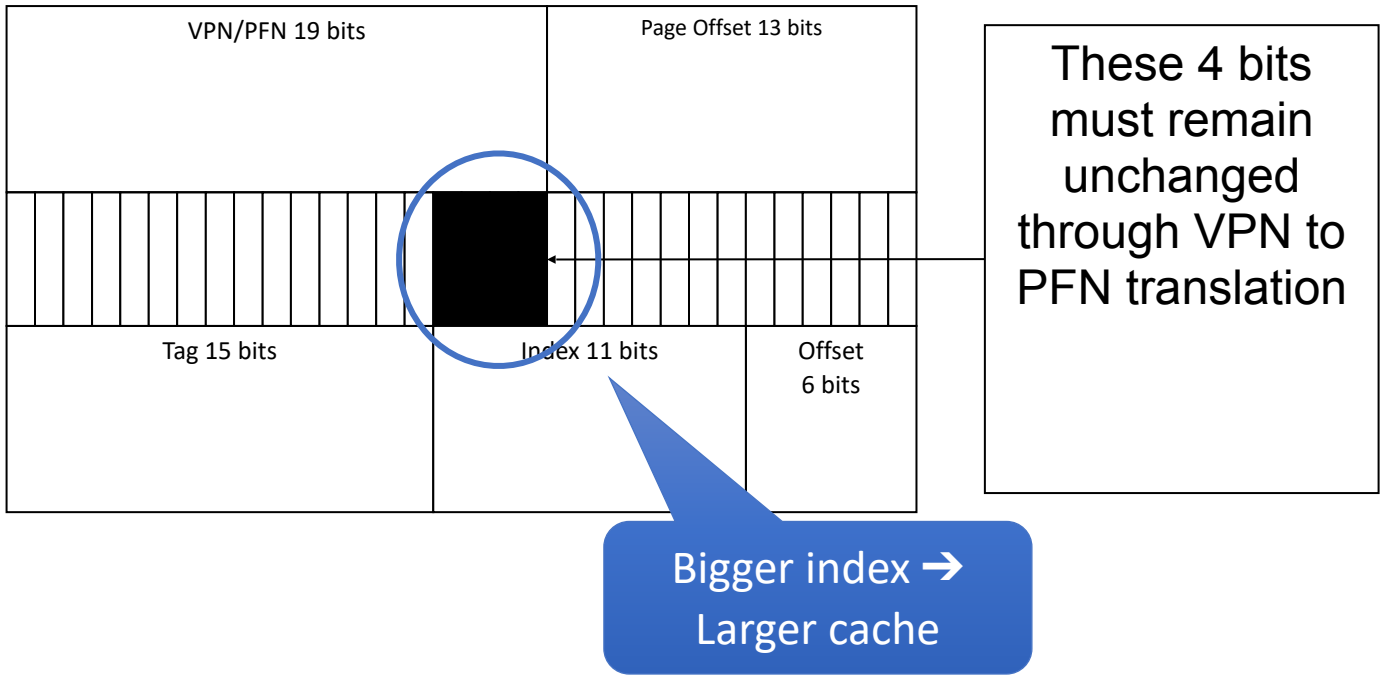


Is there a limitation in VIPT cache size?



Page coloring example

OS guarantees some bits of VPN will remain unchanged



Work out Example 10 in the book on your own and check the solution

Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)

Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)
- What does that impact?

Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)
- What does that impact?
  - A virtual page can only occupy a subset of page frames in which the low bits of the VPN match the low bits of the PFN (i.e., limit the *fully associative* nature of Virtual Memory Management).

Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)
- What does that impact?
  - A virtual page can only occupy a subset of page frames in which the low bits of the VPN match the low bits of the PFN (i.e., limit the *fully associative* nature of Virtual Memory Management).
  - We refer to this as page coloring which means the page replacement algorithm must keep track of the "color" of the VPNs and PFNs (namely those low bits) to ensure virtual pages are only loaded into like-colored page frames.

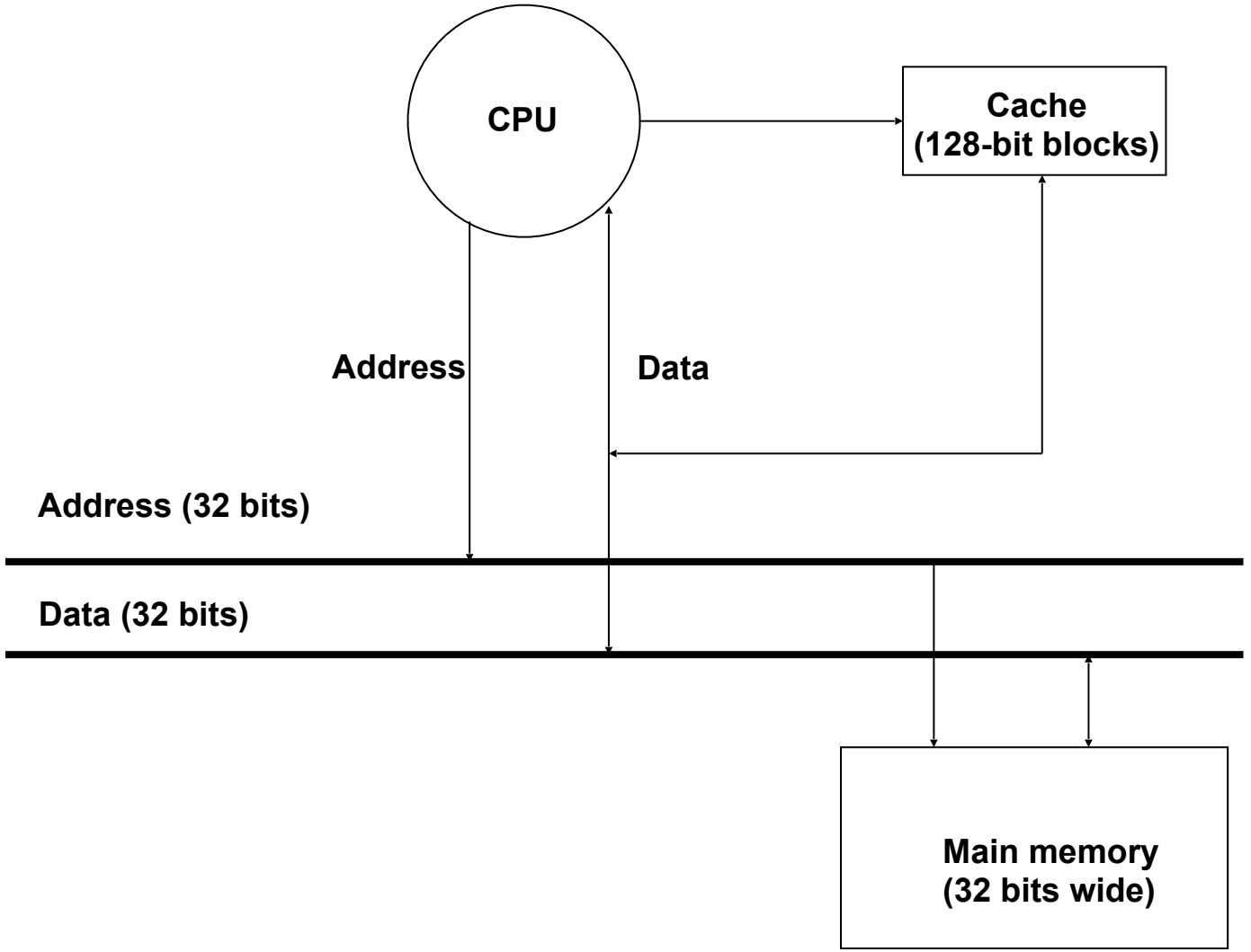
Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)
- What does that impact?
  - A virtual page can only occupy a subset of page frames in which the low bits of the VPN match the low bits of the PFN (i.e., limit the *fully associative* nature of Virtual Memory Management).
  - We refer to this as page coloring which means the page replacement algorithm must keep track of the "color" of the VPNs and PFNs (namely those low bits) to ensure virtual pages are only loaded into like-colored page frames.
  - It could make it harder to fit a working set into physical memory, but it's often workable because processes tend to use contiguous pages so the VPNs of the pages are spread evenly among the colors

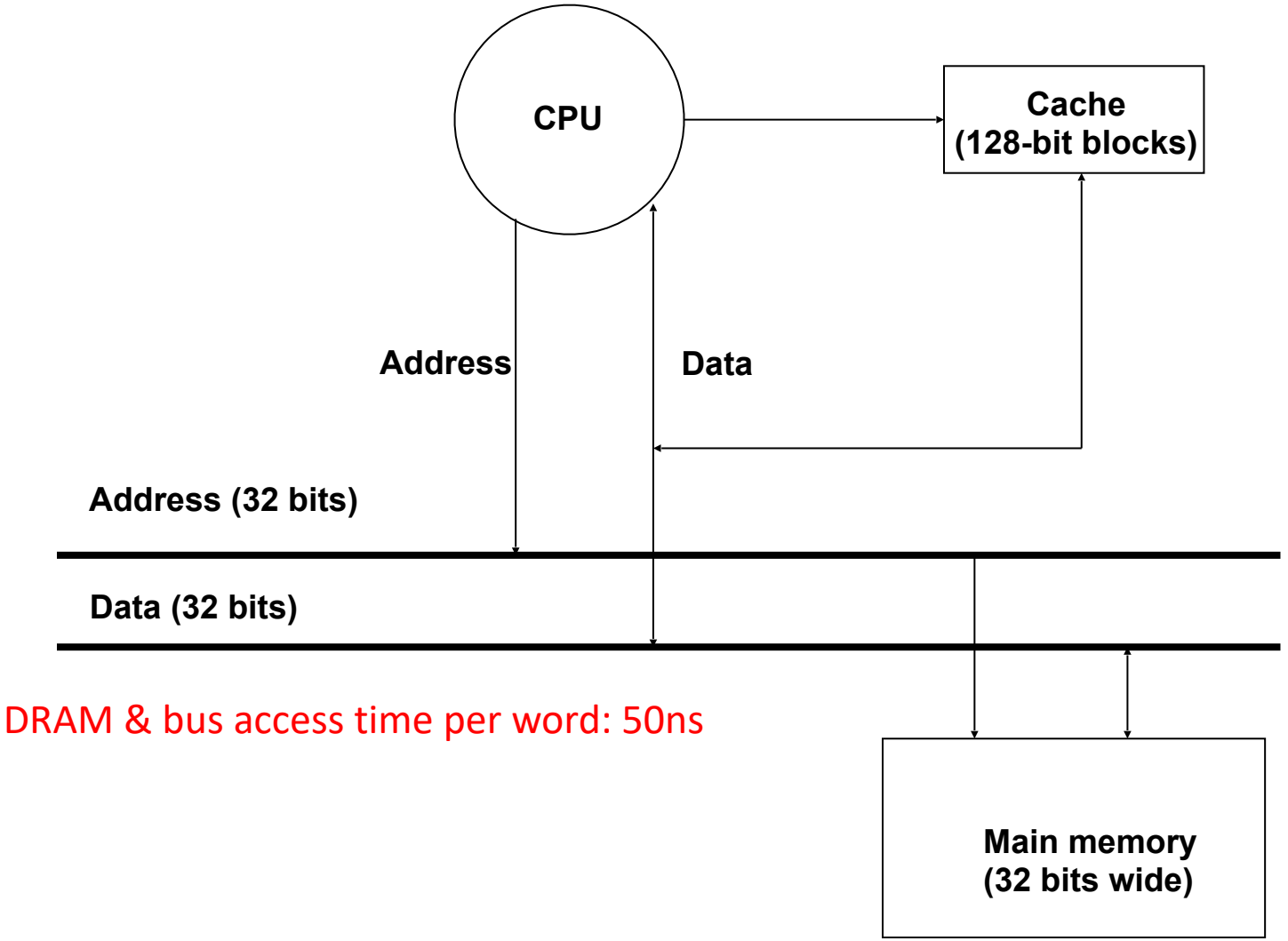
Page coloring impacts

- Require the low bits of the VPN and PFN to be identical (4 low order bits in this example)
  - This allows the use of these bits as part of either the virtual or physical address, just like we use the offset (as they won't change when virtual address is translated to physical)
- What does that impact?
  - A virtual page can only occupy a subset of page frames in which the low bits of the VPN match the low bits of the PFN (i.e., limit the *fully associative* nature of Virtual Memory Management).
  - We refer to this as page coloring which means the page replacement algorithm must keep track of the "color" of the VPNs and PFNs (namely those low bits) to ensure virtual pages are only loaded into like-colored page frames.
  - It could make it harder to fit a working set into physical memory, but it's often workable because processes tend to use contiguous pages so the VPNs of the pages are spread evenly among the colors
  - In this example using the low 4 bits of the VPN/PFN, we get 16 different "colors" of page/page frame. So a page with a VPN ending in 0010_2 can only be placed in a page frame with a number that ends in 0010_2

Simple memory system

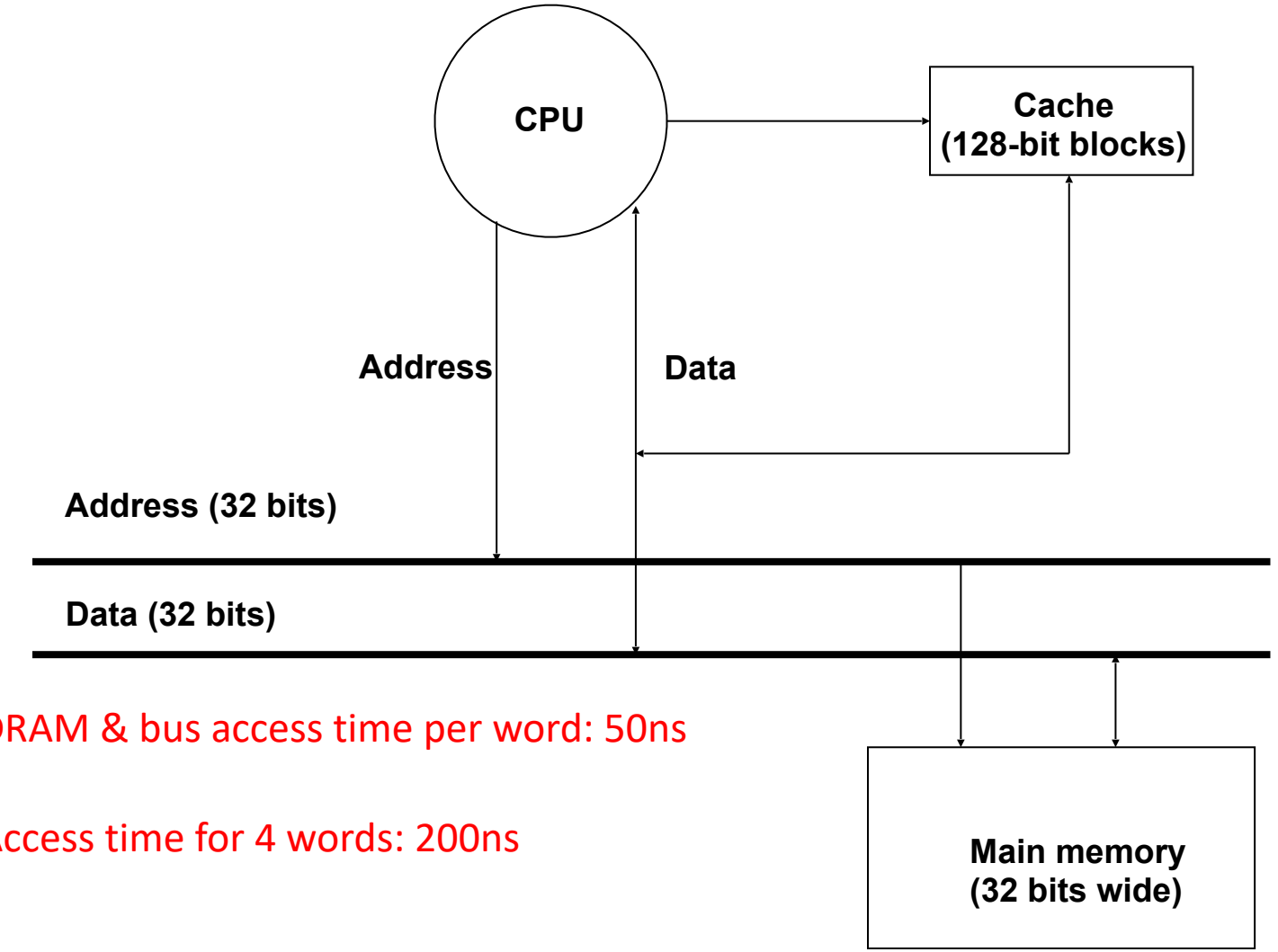


Simple memory system



DRAM & bus access time per word: 50ns

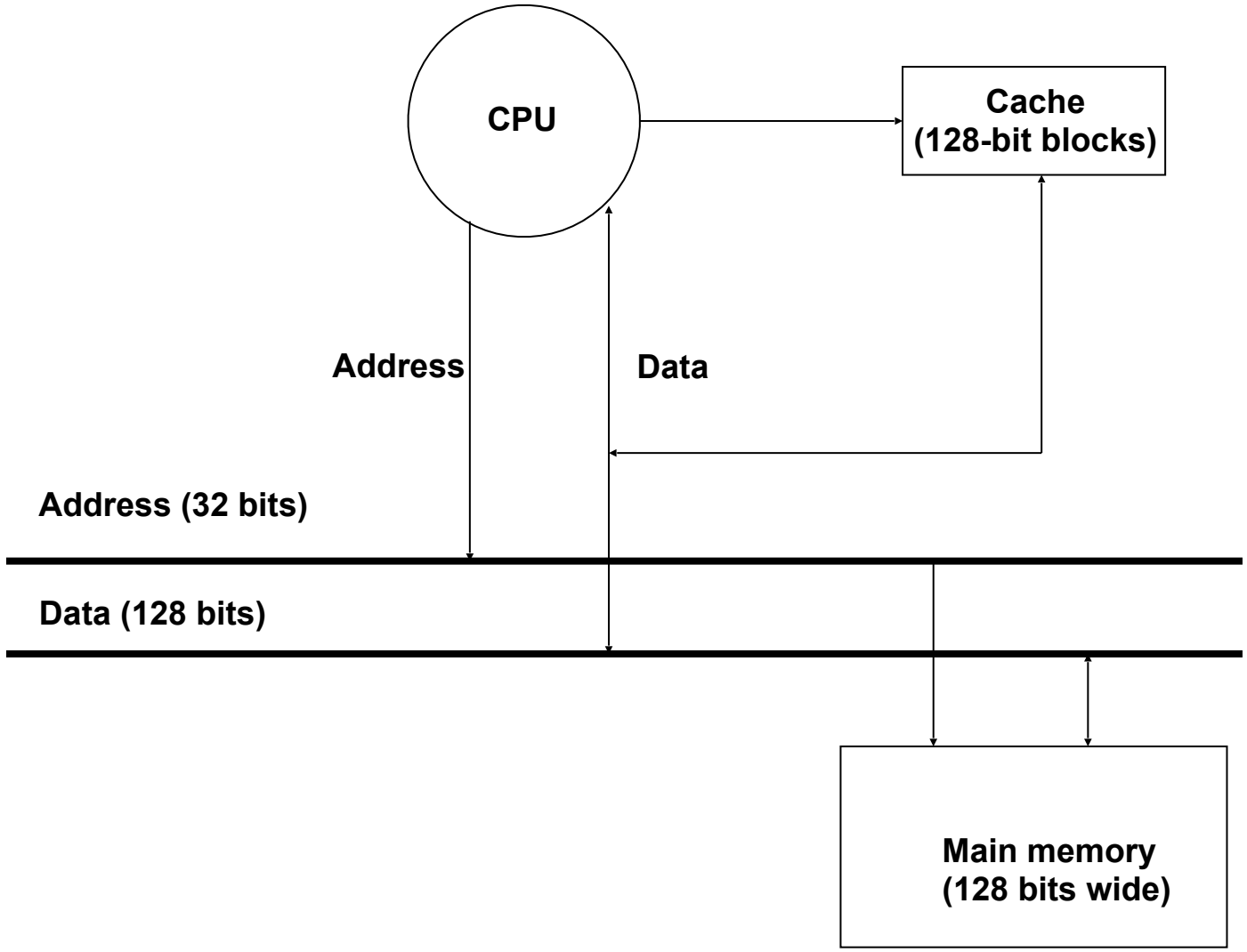
Simple memory system



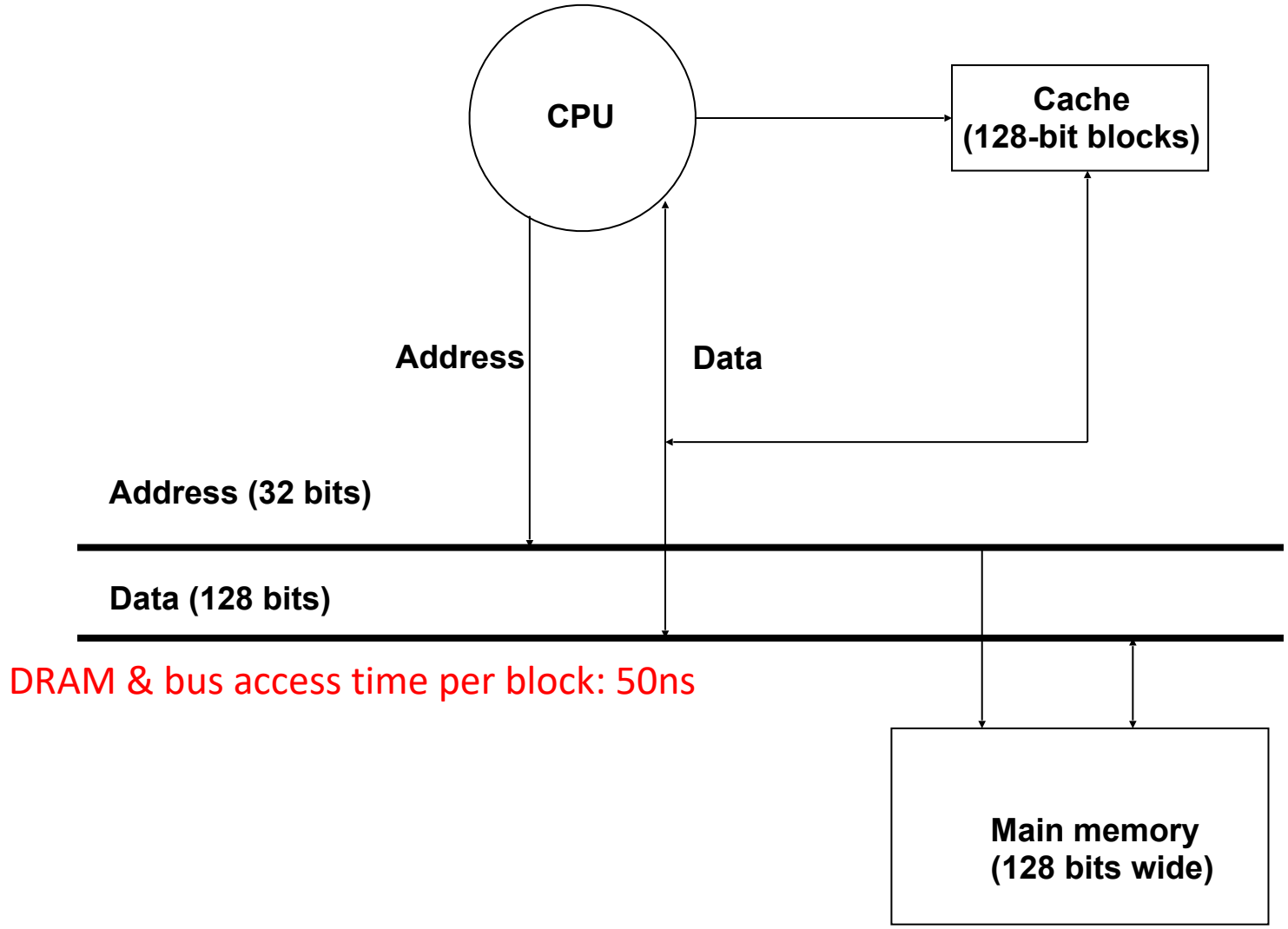
DRAM & bus access time per word: 50ns

Access time for 4 words: 200ns

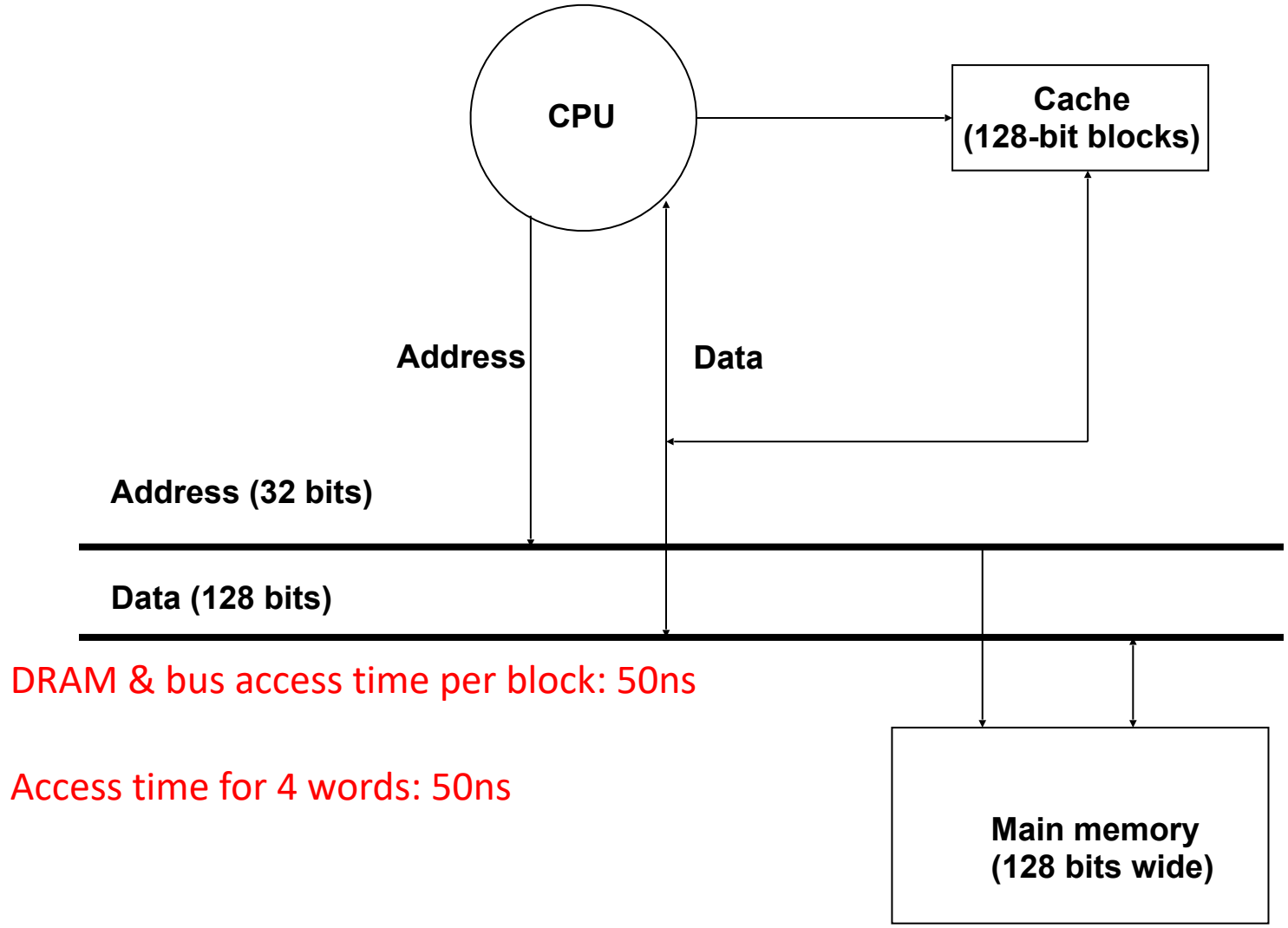
Memory system matching cache block size



Memory system matching cache block size



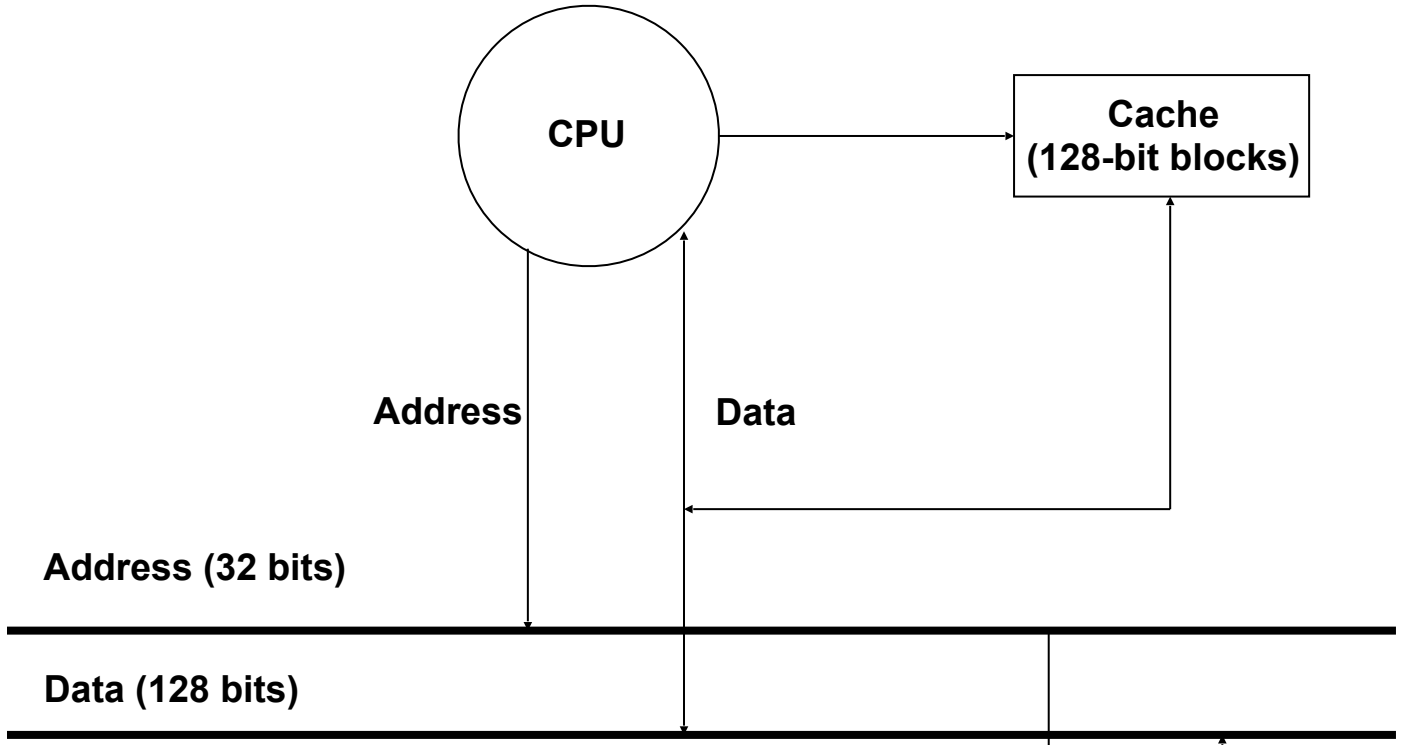
Memory system matching cache block size



DRAM & bus access time per block: 50ns

Access time for 4 words: 50ns

Memory system matching cache block size

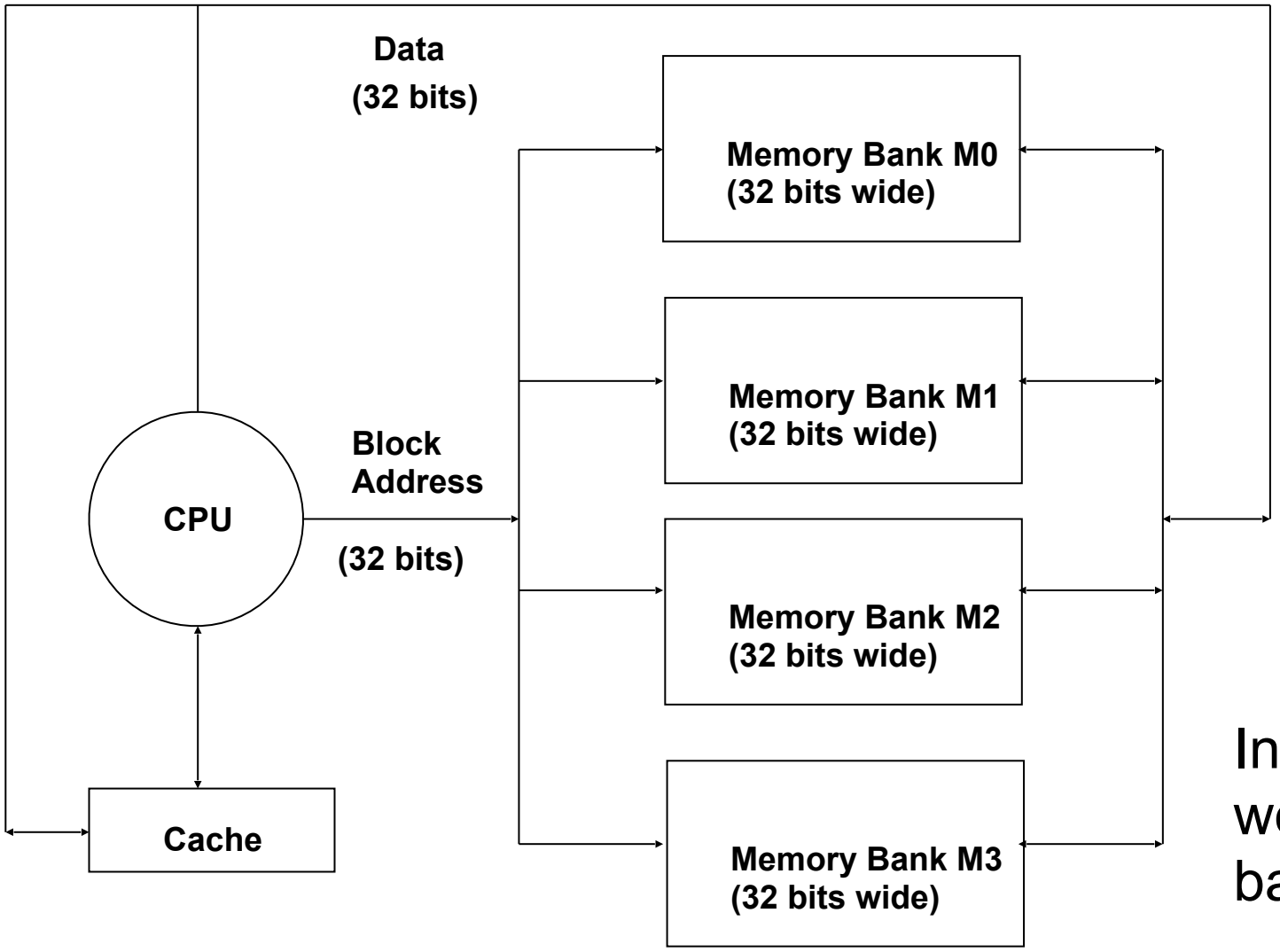


DRAM & bus access time per block: 50ns

Access time for 4 words: 50ns

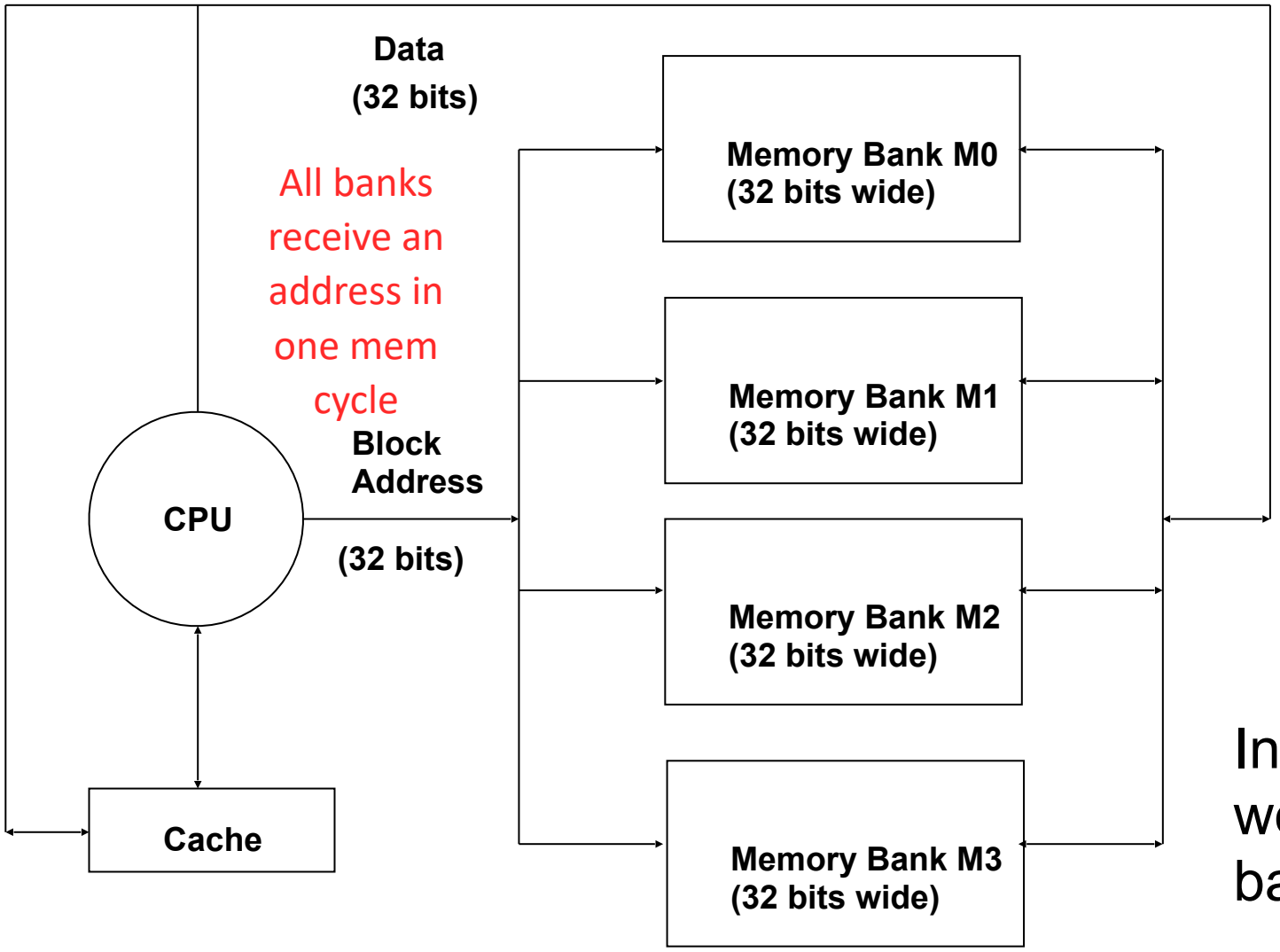
Drawback: 128-bit busses

Interleaved memory



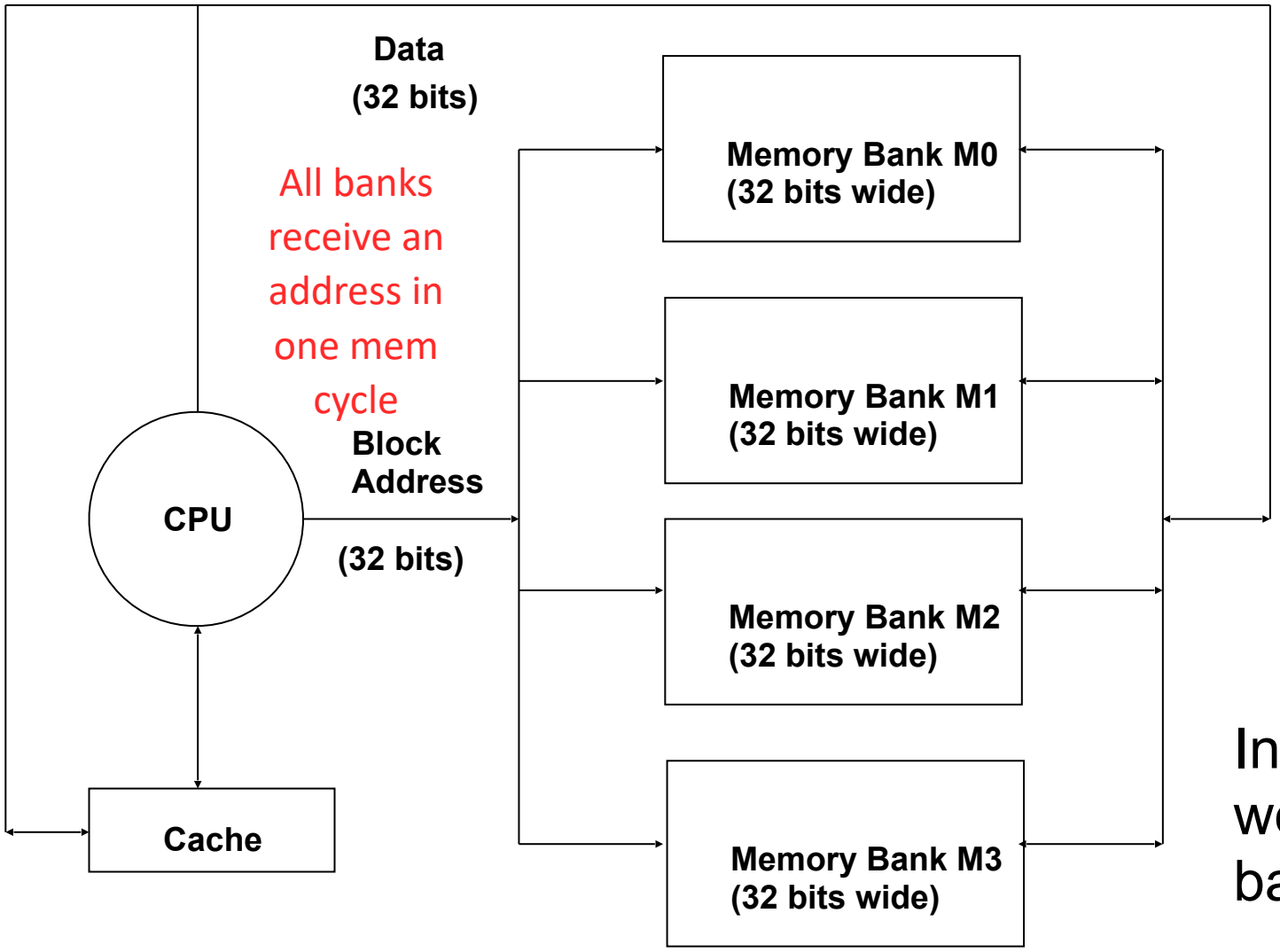
Interleave consecutive words across memory banks

Interleaved memory



Interleave consecutive words across memory banks

Interleaved memory

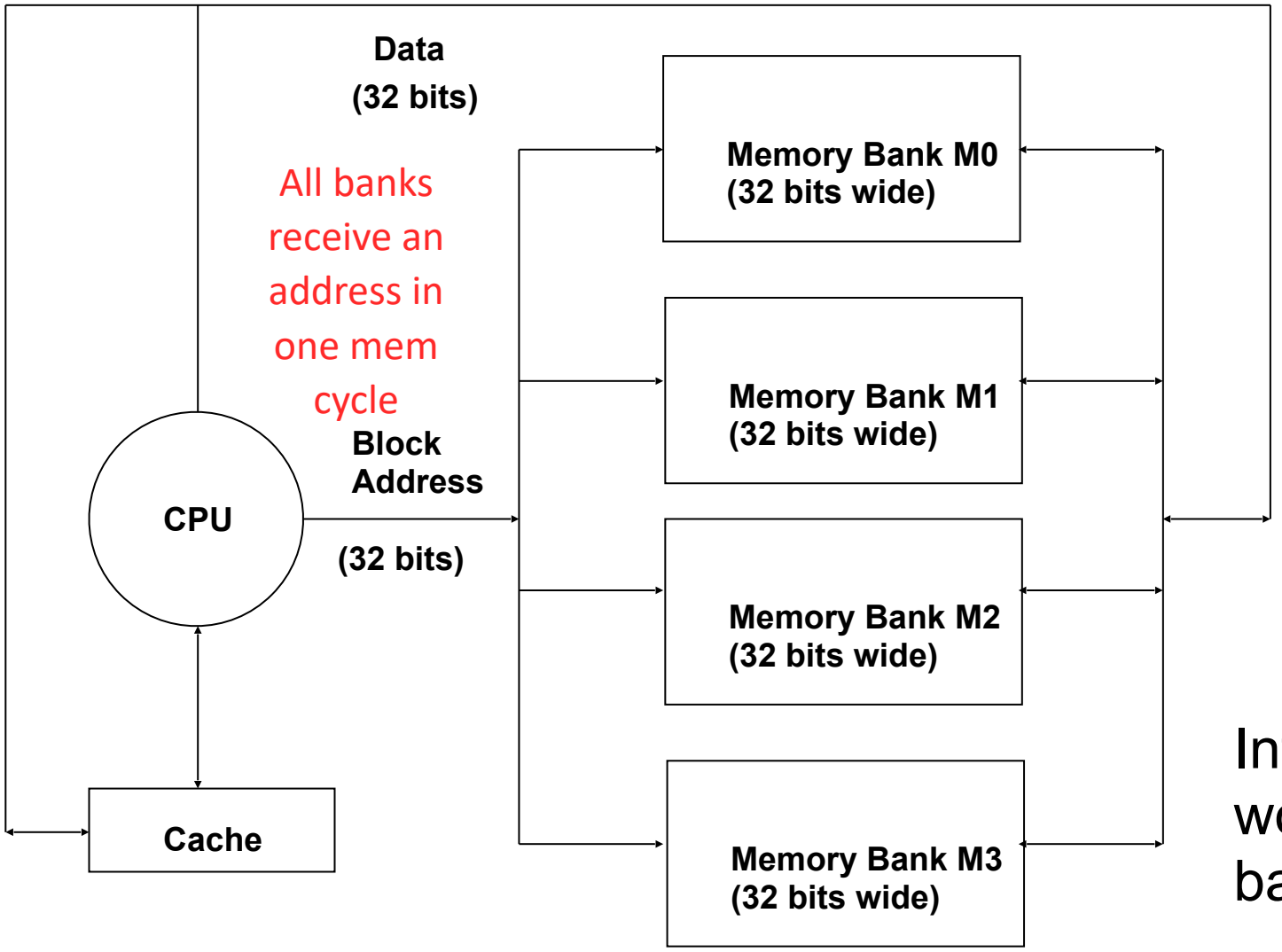


Each bank transfers in successive memory cycles

Interleave consecutive words across memory banks

Interleaved memory

Transfer time << memory access time



Each bank transfers in successive memory cycles

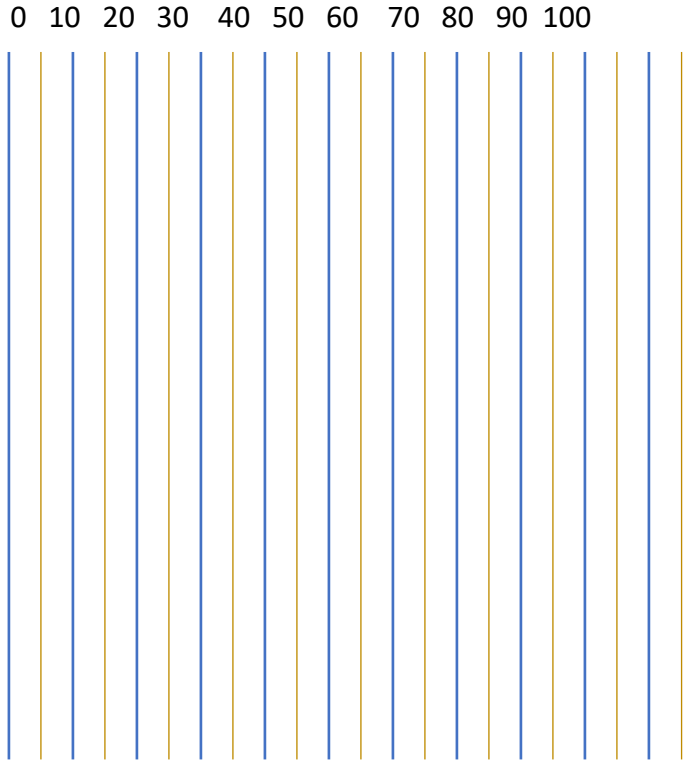
Interleave consecutive words across memory banks

Example

- 4-way interleaved memory
- DRAM access time: 80 cycles.
- Memory bus cycle time: 5 cycles
- Compute the block transfer time for a block size of 4 words. Assume all 4 words are first retrieved from the DRAM before the data transfer to the CPU.

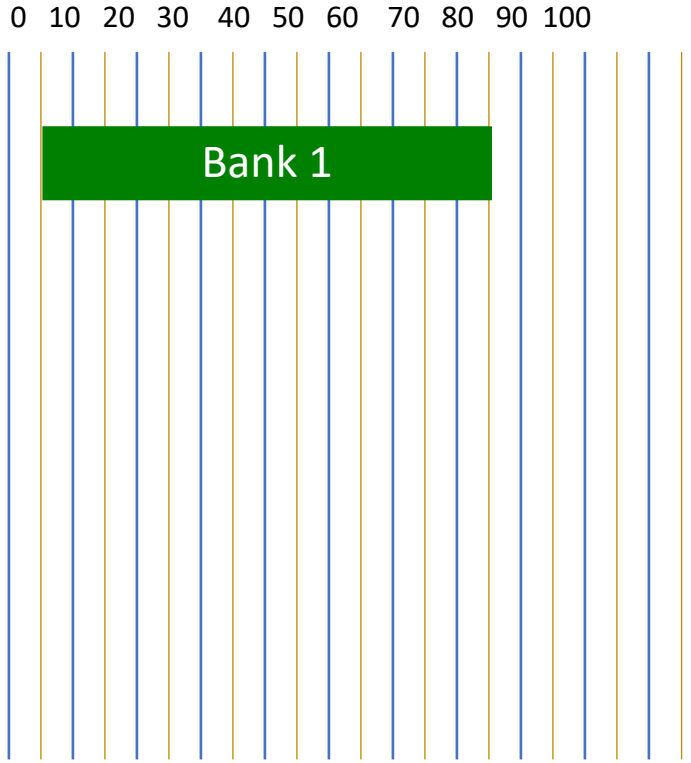
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



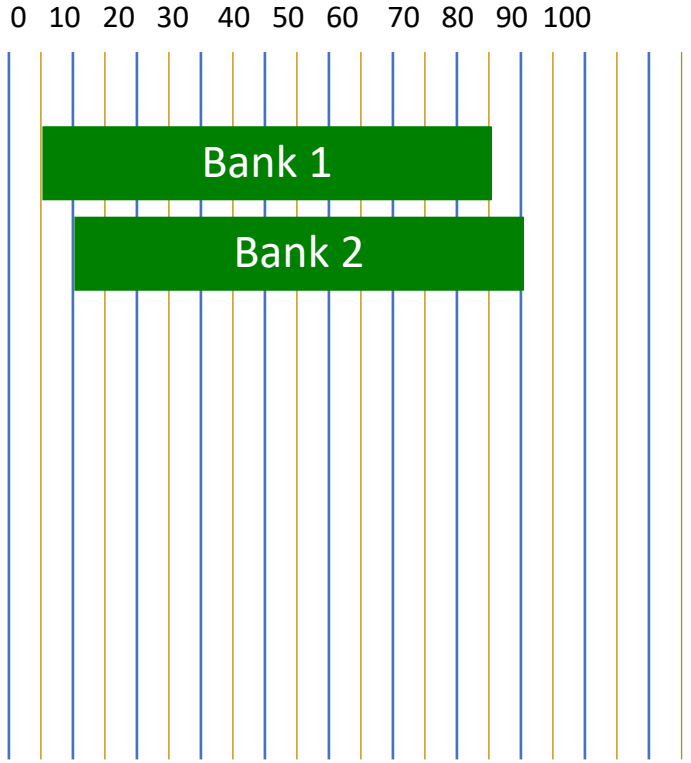
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



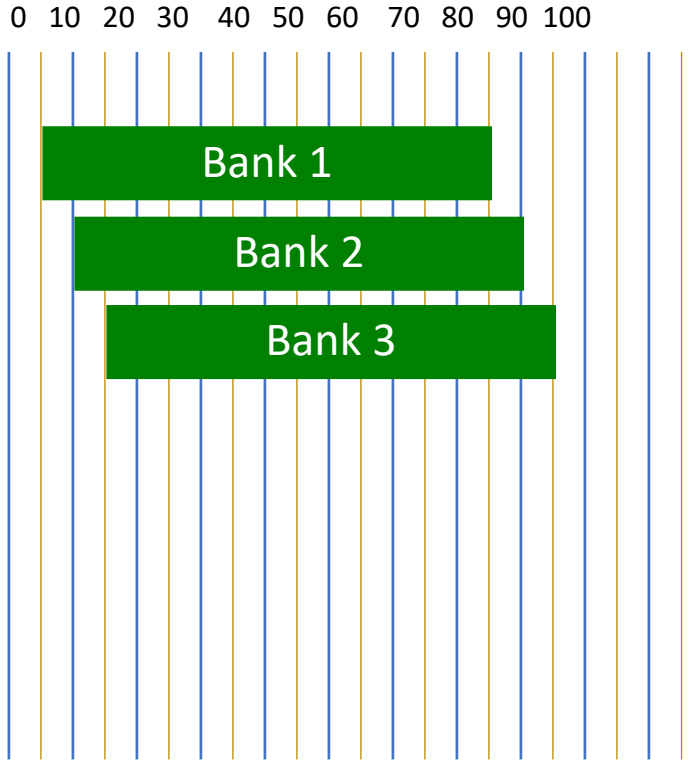
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



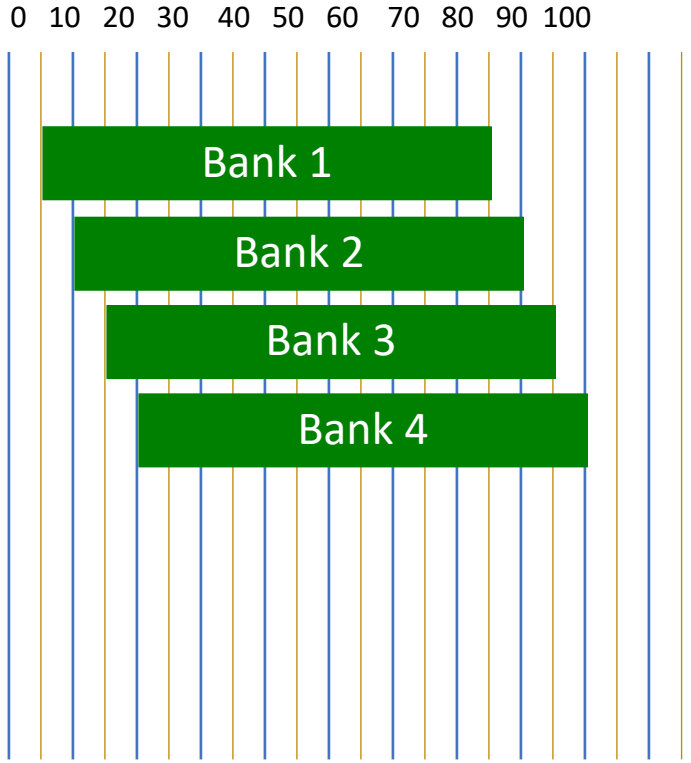
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



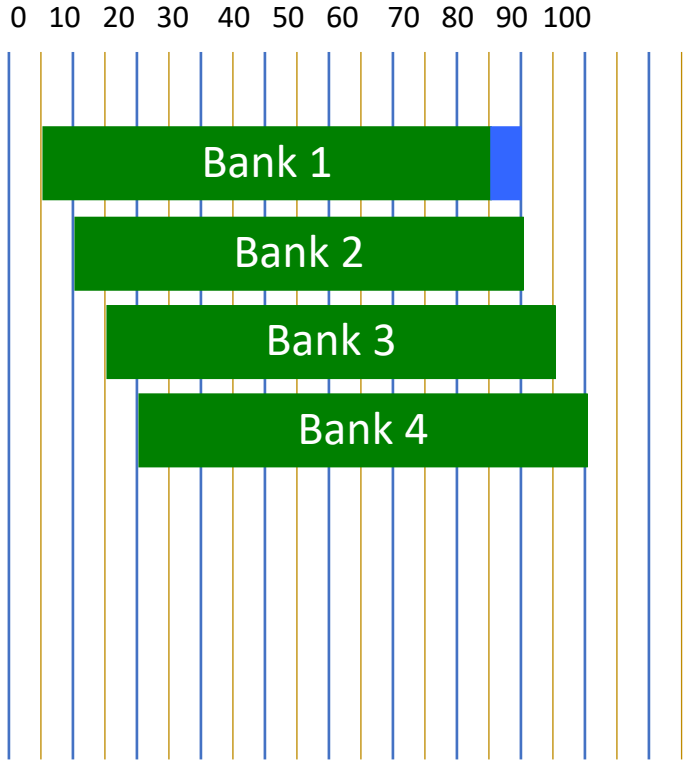
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



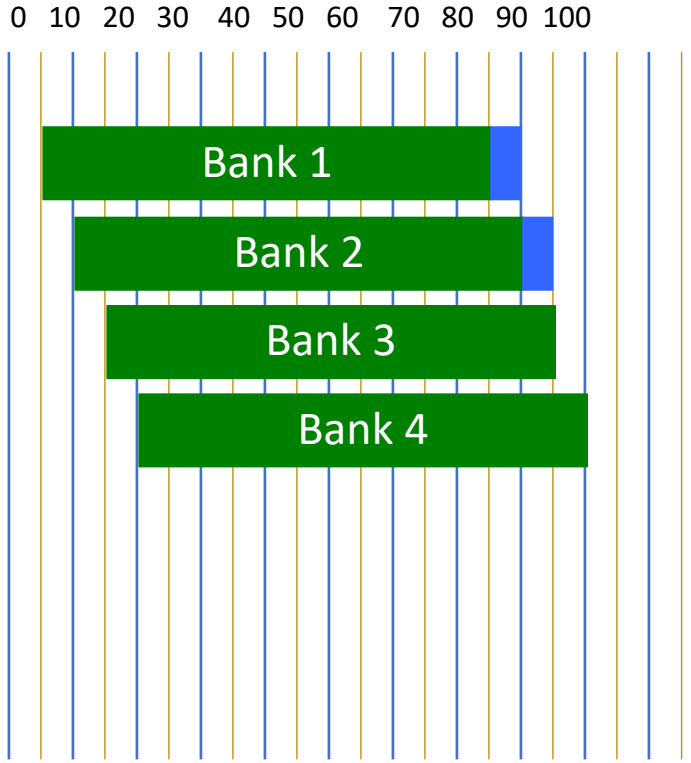
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



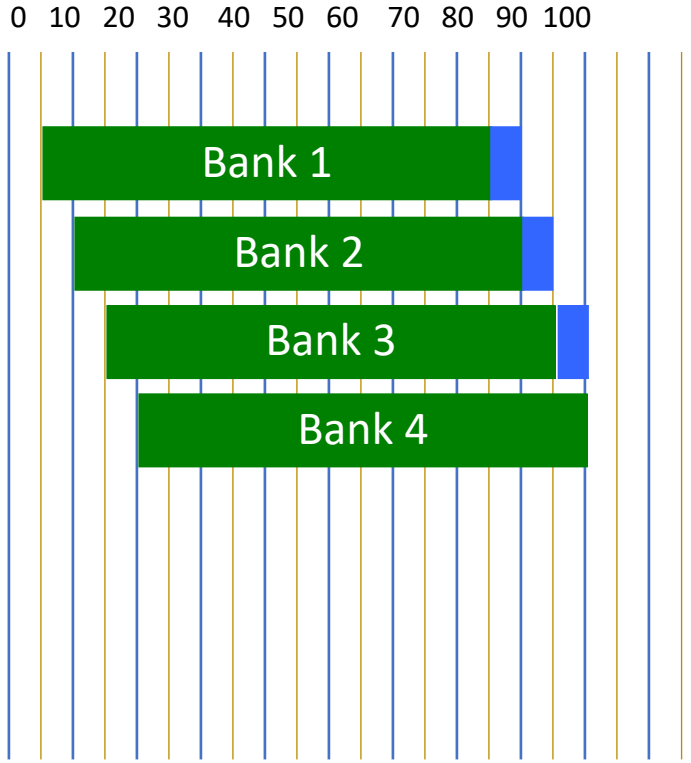
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



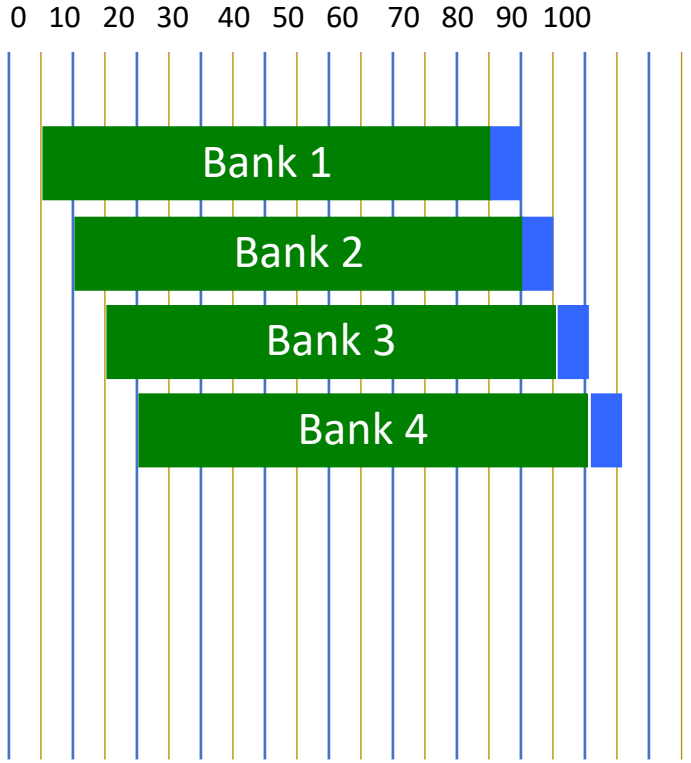
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



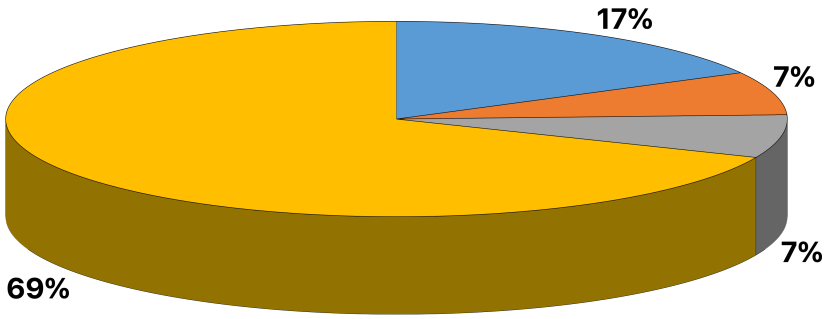
Example solution sketch

- Start memory fetches at 5, 10, 15, 20 cycles
- Bank 4 memory fetch finishes at 105 cycles
- Data transfers to cache complete at 90, 95, 100, 105 cycles thanks to bank-level parallelism
  - Principle similar to pipelining!



Interleaved memory is useful because

A. I just want the participation credit
B. It increases the cache block size
C. It reduces the cache block size
D. It allows concurrent fetches of data blocks from memory without requiring wider busses
E. It decreases the size of the cache index



Cache hierarchy: Relative sizes & latencies, circa 2017

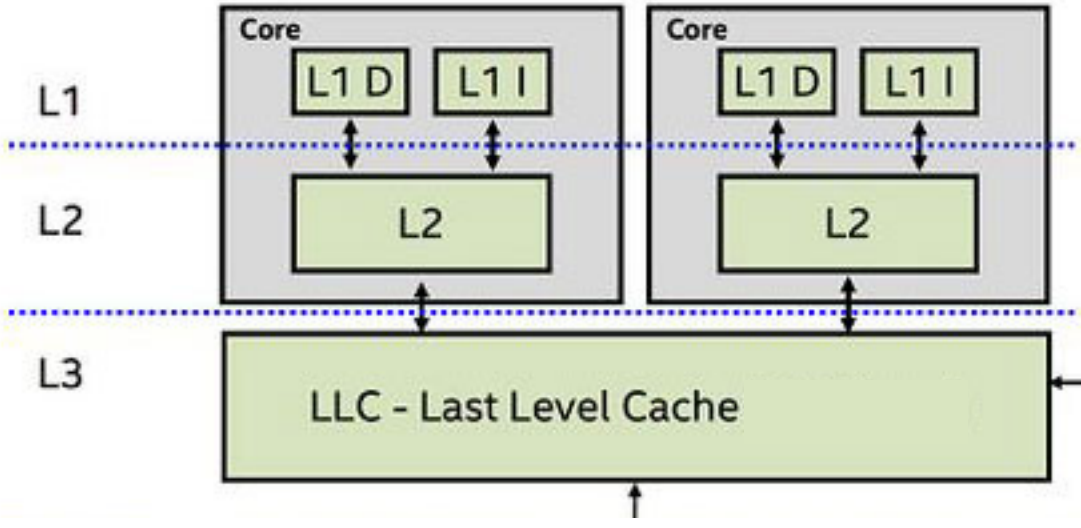
Skylake-SP server processors (no major changes since)

Sizes:

- L1 Data cache = 32 KB private per core, 64 B/line, 8-WAY.
- L1 Instruction cache = 32 KB private per core, 64 B/line, 8-WAY.
- L2 cache = 1MB private per core, 64 B/line, 16-WAY
- L3 cache = shared 1.375MB per core, 64 B/line, 11-WAY

Latencies (@ 3.7GHz frequency):

- L1 Data Cache = 4 cycles
- L2 Cache = 12 cycles
- L3 Cache = 42 cycles
- DRAM (memory) Latency = 42 cycles + 51 ns



Summary of Chapter 9 terminology

Category	Vocabulary	Details
Principle of locality (Section 9.2)	Spatial	Access to contiguous memory locations
	Temporal	Reuse of memory locations already accessed
Cache organization	Direct-mapped	One-to-one mapping (Section 9.6)
	Fully associative	One-to-any mapping (Section 9.12.1)
	Set associative	One-to-many mapping (Section 9.12.2)
Cache reading/writing (Section 9.8)	Read hit/Write hit	Memory location being accessed by the CPU is present in the cache
	Read miss/Write miss	Memory location being accessed by the CPU is not present in the cache
Cache write policy (Section 9.8)	Write through	CPU writes to cache and memory
	Write back	CPU only writes to cache; memory updated on replacement
Cache parameters	Total cache size (S)	Total data size of cache in bytes
	Block Size (B)	Size of contiguous data in one data block
	Degree of associativity (p)	Number of homes a given memory block can reside in a cache
	Number of cache lines (L)	S/pB
	Cache access time	Time in CPU clock cycles to check hit/miss in cache
	Unit of CPU access	Size of data exchange between CPU and cache
	Unit of memory transfer	Size of data exchange between cache and memory
	Miss penalty	Time in CPU clock cycles to handle a cache miss
Memory address interpretation	Index (n)	$\log_2 L$ bits, used to look up a particular cache line
	Block offset (b)	$\log_2 B$ bits, used to select a specific byte within a block
	Tag (t)	a – (n+b) bits, where a is number of bits in memory address; used for matching with tag stored in the cache

Summary of Chapter 9 terminology

Category	Vocabulary	Details
Cache entry/cache block/ cache line	Valid bit	Signifies data block is valid
	Dirty bits	For write-back, signifies if the data block is more up to date than memory
	Tag	Used for tag matching with memory address for hit/miss
	Data	Actual data block
Performance metrics	Hit rate (h)	Percentage of CPU accesses served from the cache
	Miss rate (m)	1 – h
	Avg. Memory stall	$\text{Misses-per-instruction}_{Avg}$ * $\text{miss-penalty}_{Avg}$
	Effective memory access time ($AMAT_i$) at level i	$AMAT_i = T_i + m_i * AMAT_{i+1}$
	Effective CPI	$\text{CPI}_{Avg} + \text{Memory-stalls}_{Avg}$
Types of misses	Compulsory miss	Memory location accessed for the first time by CPU
	Conflict miss	Miss incurred due to limited associativity even though the cache is not full
	Capacity miss	Miss incurred when the cache is full
Replacement policy	FIFO	First in first out
	LRU	Least recently used
Memory technologies	SRAM	Static RAM with each bit realized using a flip flop
	DRAM	Dynamic RAM with each bit realized using a capacitive charge
Main memory	DRAM access time	DRAM read access time
	DRAM cycle time	DRAM read and refresh time
	Bus cycle time	Data transfer time between CPU and memory
	Simulated interleaving using DRAM	Using page mode bits of DRAM

Concluding Remarks

- Project 4 (released tomorrow) learning outcome is CPU scheduling algorithms

- Extra Credit project released after spring break

Have a great Spring Break!