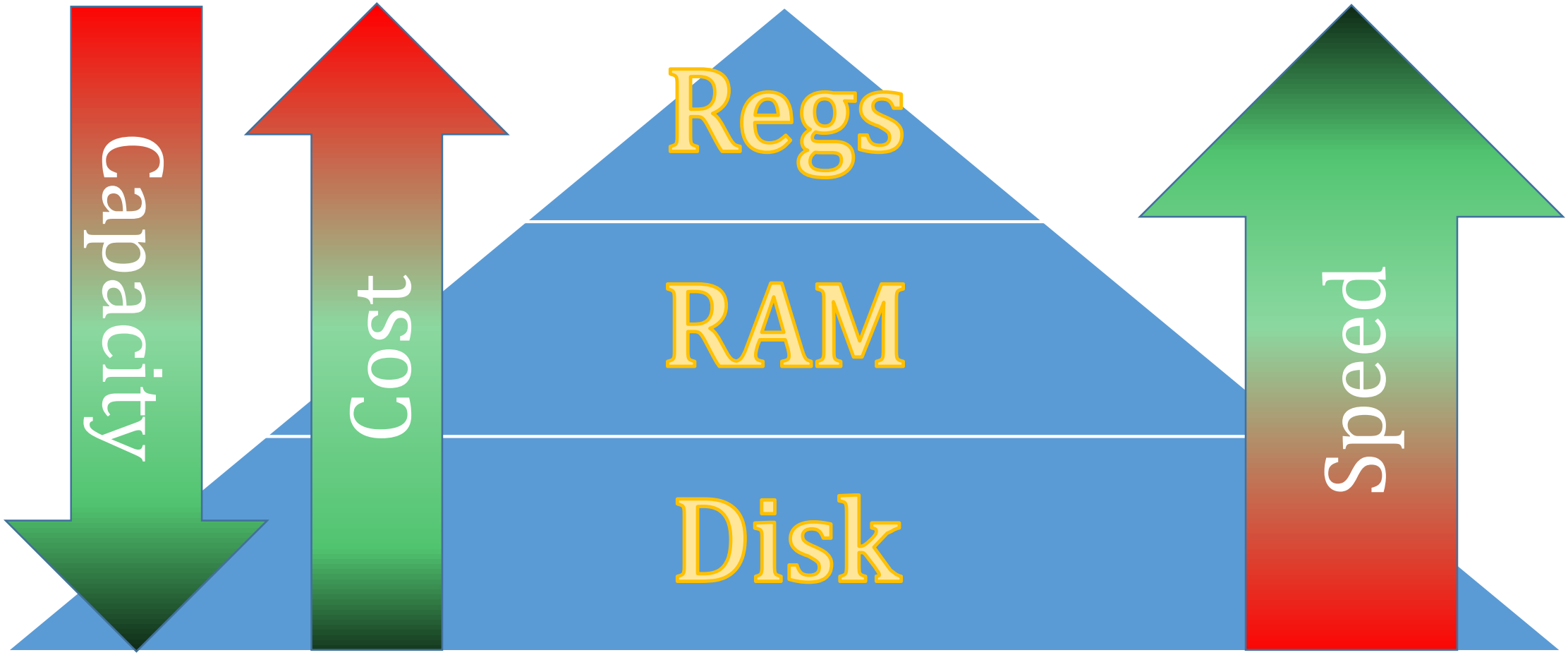
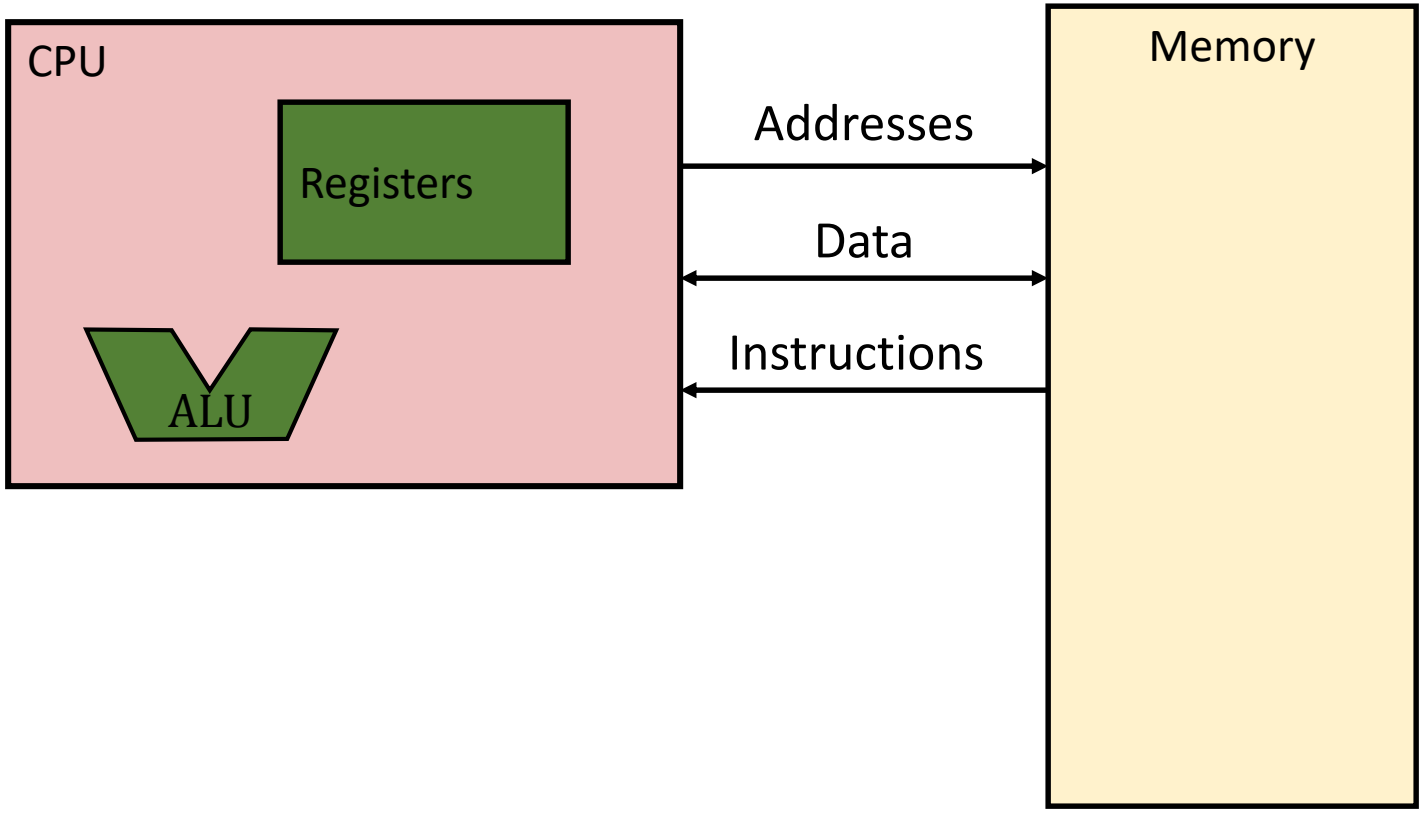
Cached Memory

Computer Systems Chapter 6.2-6.5

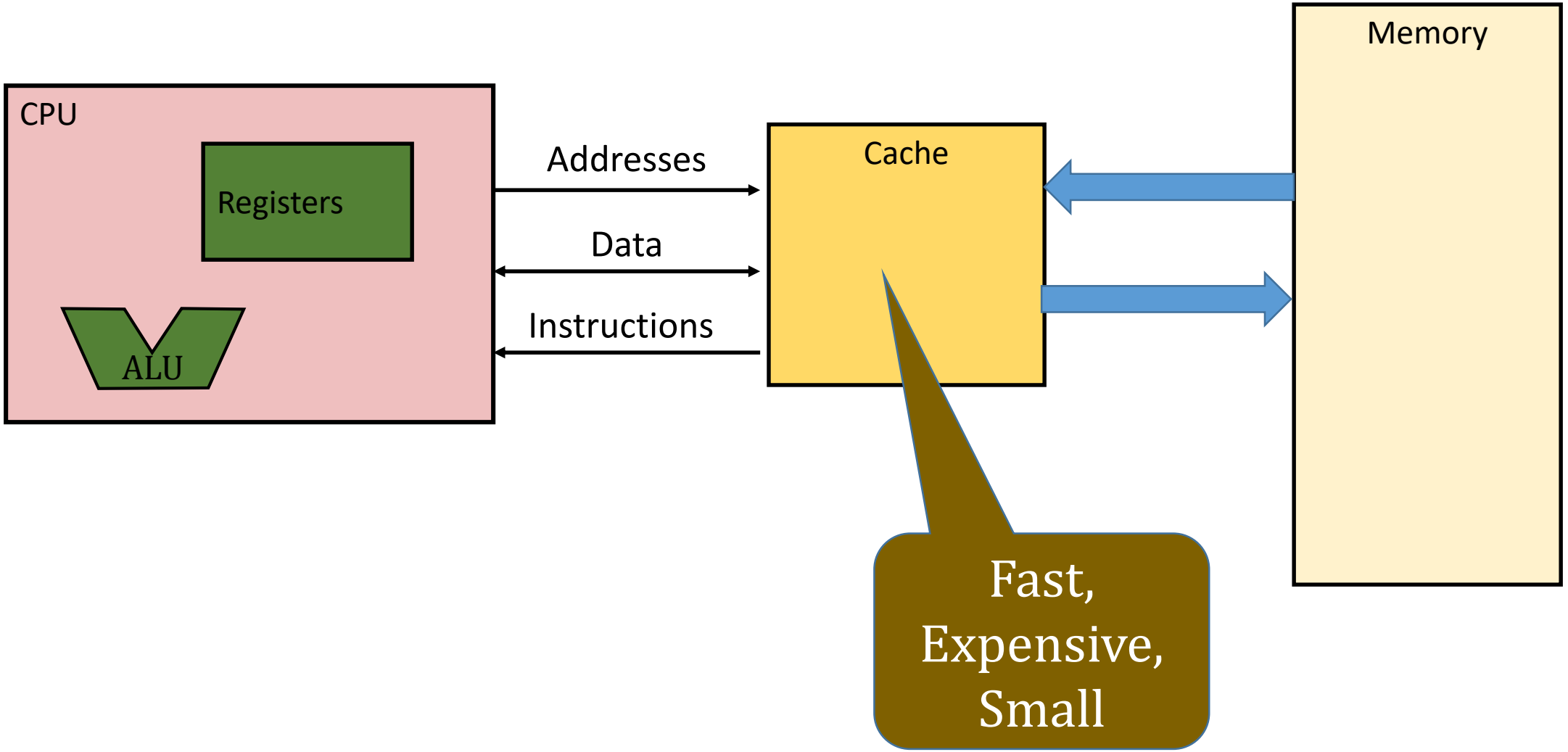
The Memory Hierarchy



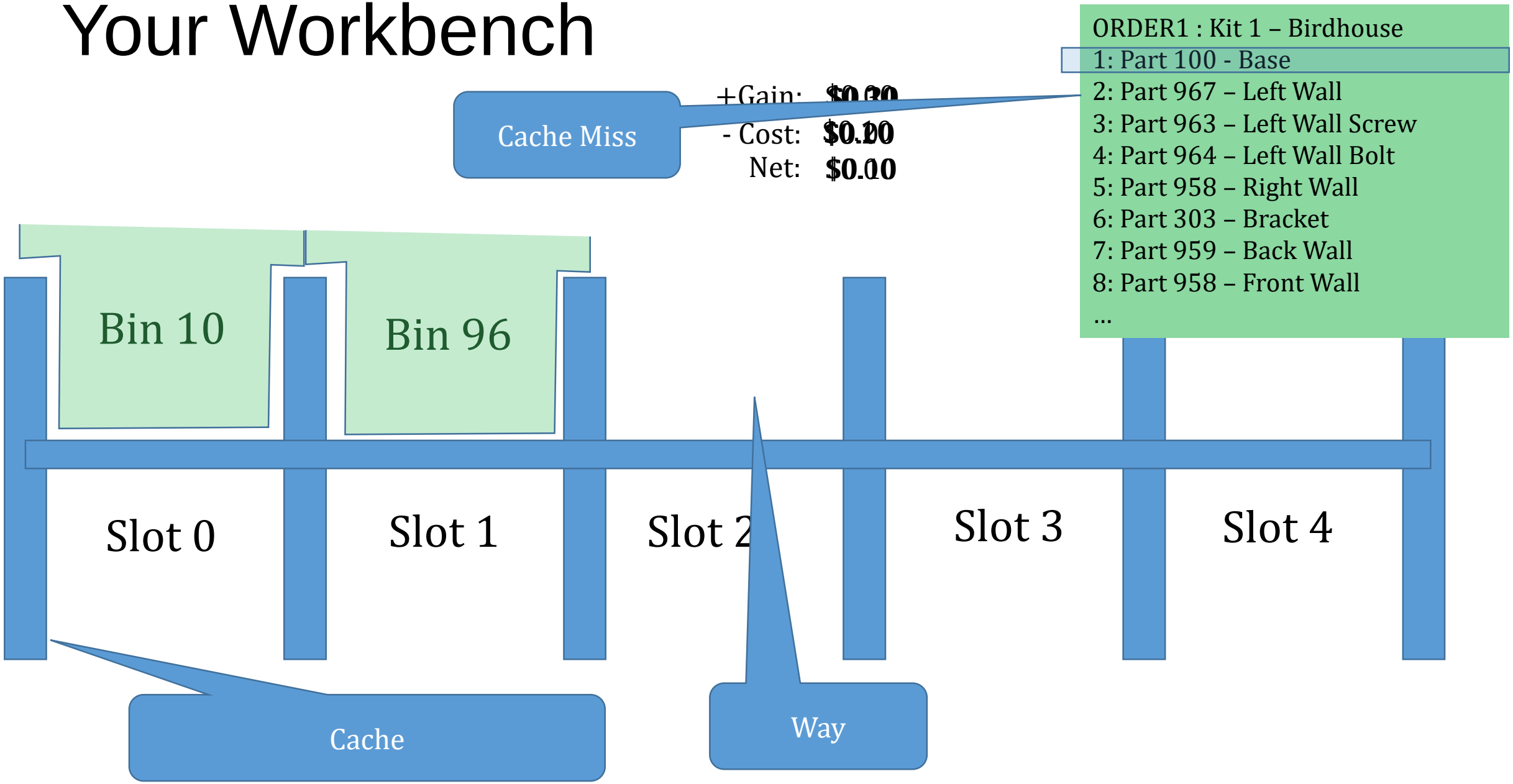
The Cache Concept



The Cache Concept



Your Workbench



Cache Pro's and Con's

Pro's

- If CPU requests an address that is in cache, it gets it MUCH faster!

Cache Hit

Con's

- If CPU requests an address that is NOT in cache, it gets it slower

Cache Miss

What happens on Cache Miss?

1. Recognize data is not in cache
2. Make room in Cache
   copy modified data block from cache to memory
3. Copy requested block of memory into cache
4. Return data from cache

Challenge

- Don't return a ~~bin~~ way that you are going to need soon!
  - if you do, you have to pay (in time) for sending it back, and then getting it again
- Don't know if you will need that ~~bin~~ block of memory again soon!
  - No "look-ahead" to see what ~~parts~~ addresses in ~~the order~~ memory you are going to need
- Only know one thing about ~~orders~~ programs:

"~~Part numbers~~ addresses tend to be sequential, but when the sequence is not followed, there is a high probability that the next ~~part number~~ address showed up recently ~~on the order~~."

Example Memory Trace

- Fetch Instruction at 0x000055555554a92
- Fetch Integer at 0x00007fffffffffab48
- Fetch Instruction at 0x000055555554a94
- Store Integer at 0x00007fffffffffab48
- Fetch Instruction at 0x000055555554a96
- Fetch Integer at 0x00007fffffffffab8c
- Fetch Instruction at 0x000055555554905
- …

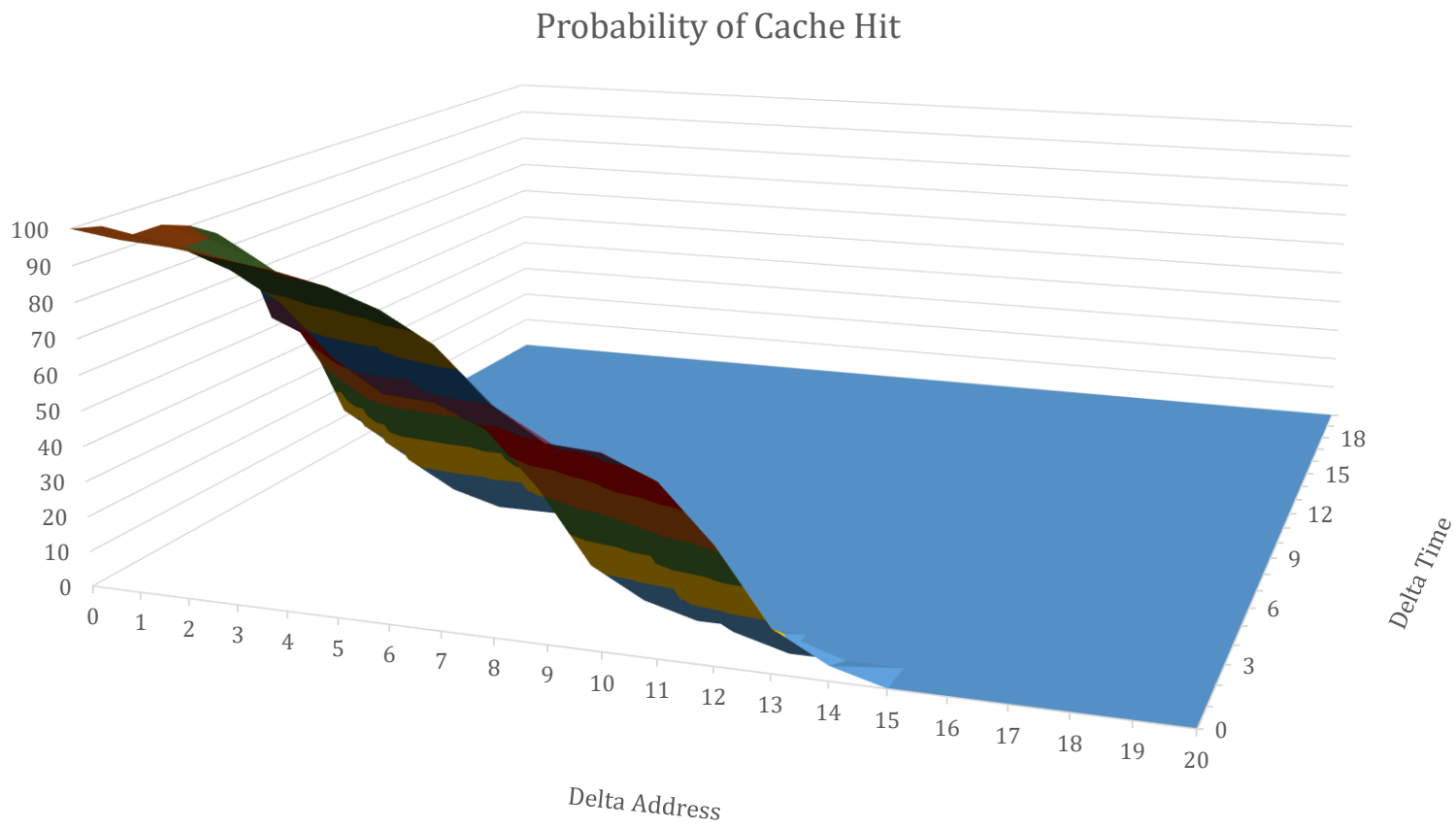
Trace
I 000055555554a92
R 00007fffffffffab48
I 000055555554a94
W 00007fffffffffab48
I 000055555554a96
R 00007fffffffffab8c
I 000055555554905

Locality

- Local variables are next to each other in the stack frame
- Tend to read a matrix sequentially in row major order
- Instructions are read sequentially
  - If there is a loop, a small cluster of instructions is re-read multiple times

- Chances are, for most memory accesses, the block is already in the cache!
- It is not uncommon to get 98%+ cache hit rates!

Locality has two dimensions



Cache is in Hardware

- Small block of fast, expensive random-access memory
  - Small because it's expensive

When CPU requests memory:

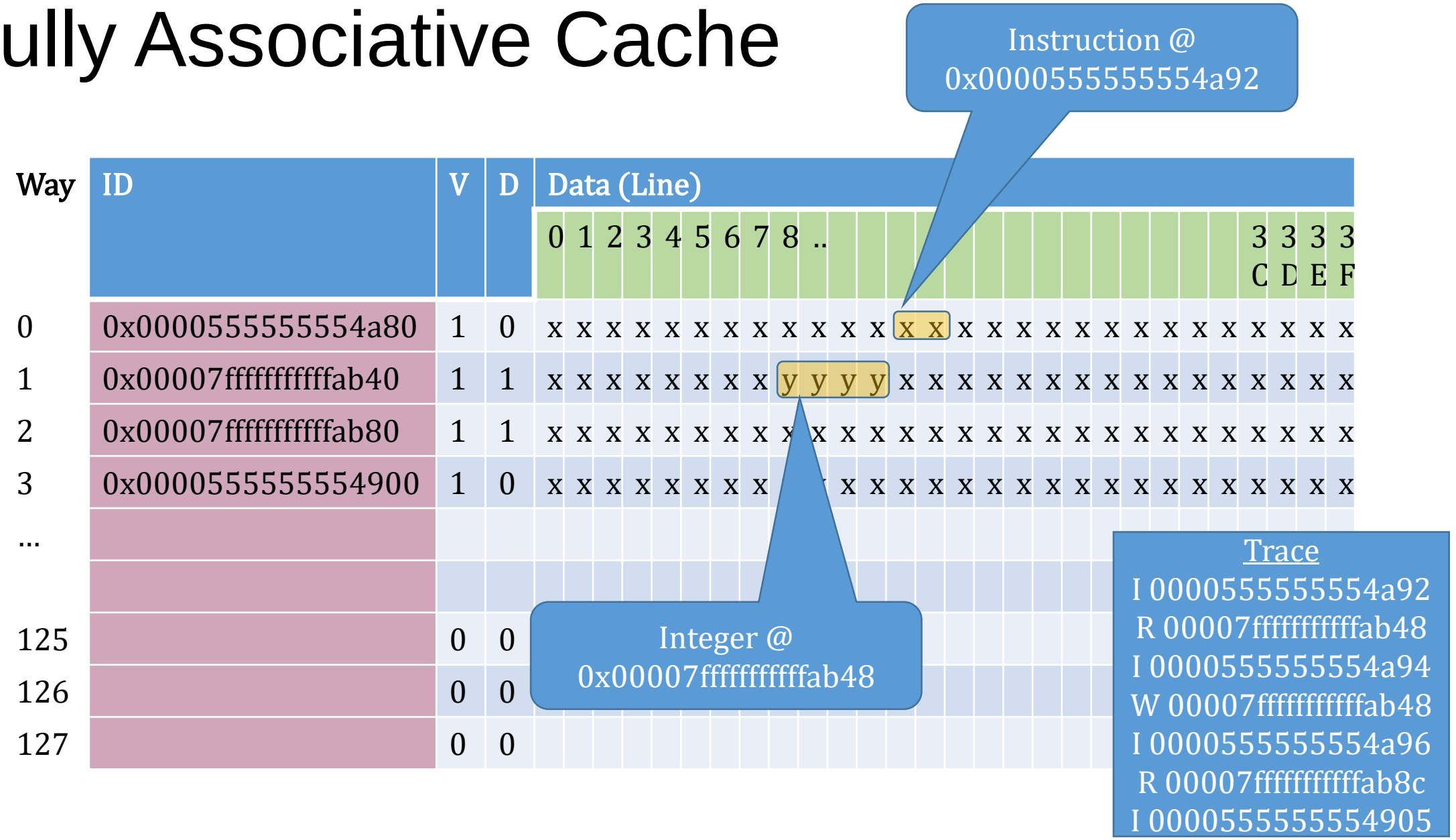
- Need to check to see if that memory is in cache
- If not:
  - need to eject a "victim" cache block, send it back to memory
  - need to fetch block from memory into cache
  - need to keep track of which block this is
- Return data values from cache

Fully Associative Cache

- Define cache *block size* = 2^b the amount of data transferred between cache and ROM memory – most often $64=2^6$ bytes

- Divide the cache up into 2^w *ways*
  - Each way contains one block (e.g. 64 bytes)
  - Each way contains an ID *tag* – where in memory does this block come from
  - Each way contains flags, *valid* flag and *dirty* flag
  - For instance, an 8K fully associative cache contains $128=2^7$ ways

Fully Associative Cache



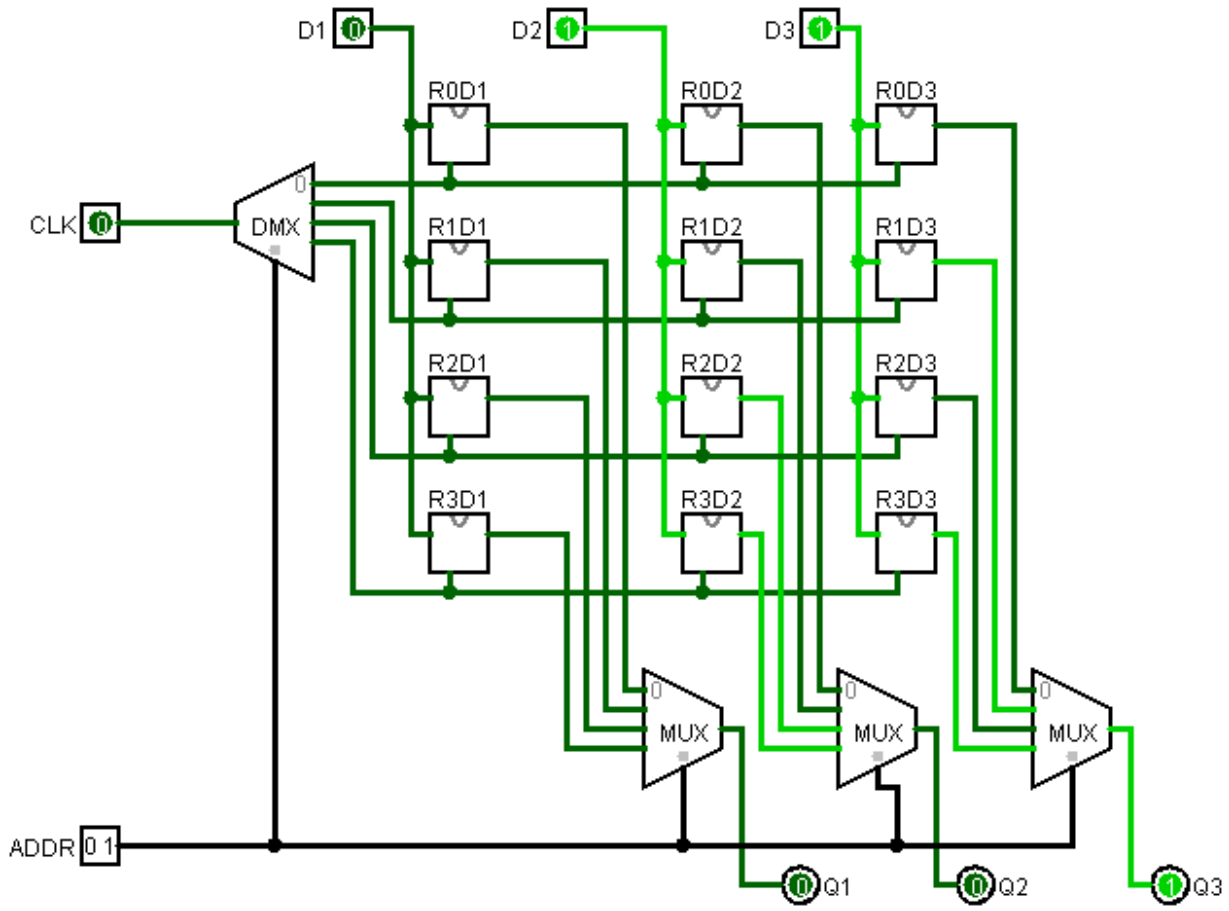
Address Sub-Fields – Fully Associative

- Divide each memory address into two sub-fields
  - Tag – High order bits - where in memory this block comes from
  - Block Offset – b Low order bits that define the offset within a block

Note: Hex digit boundary

2^{63}	2^{62}	…	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{63}	b_{62}	…	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
Tag					Block Offset					
0x000055555554a8[/12]			1	0	0	1	0	0	1	0
00007fffffffffab4[/08]			0	1	0	0	1	0	0	0
00007fffffffffab8[/0c]			1	0	0	0	1	1	0	0
55555555490[/05]			0	0	0	0	0	1	0	1

Random Access Memory (RAM)



ADDR	DATA		
00	0	0	0
01	0	0	1
10	0	1	0
11	1	1	1

Random Access vs. Content Addressable

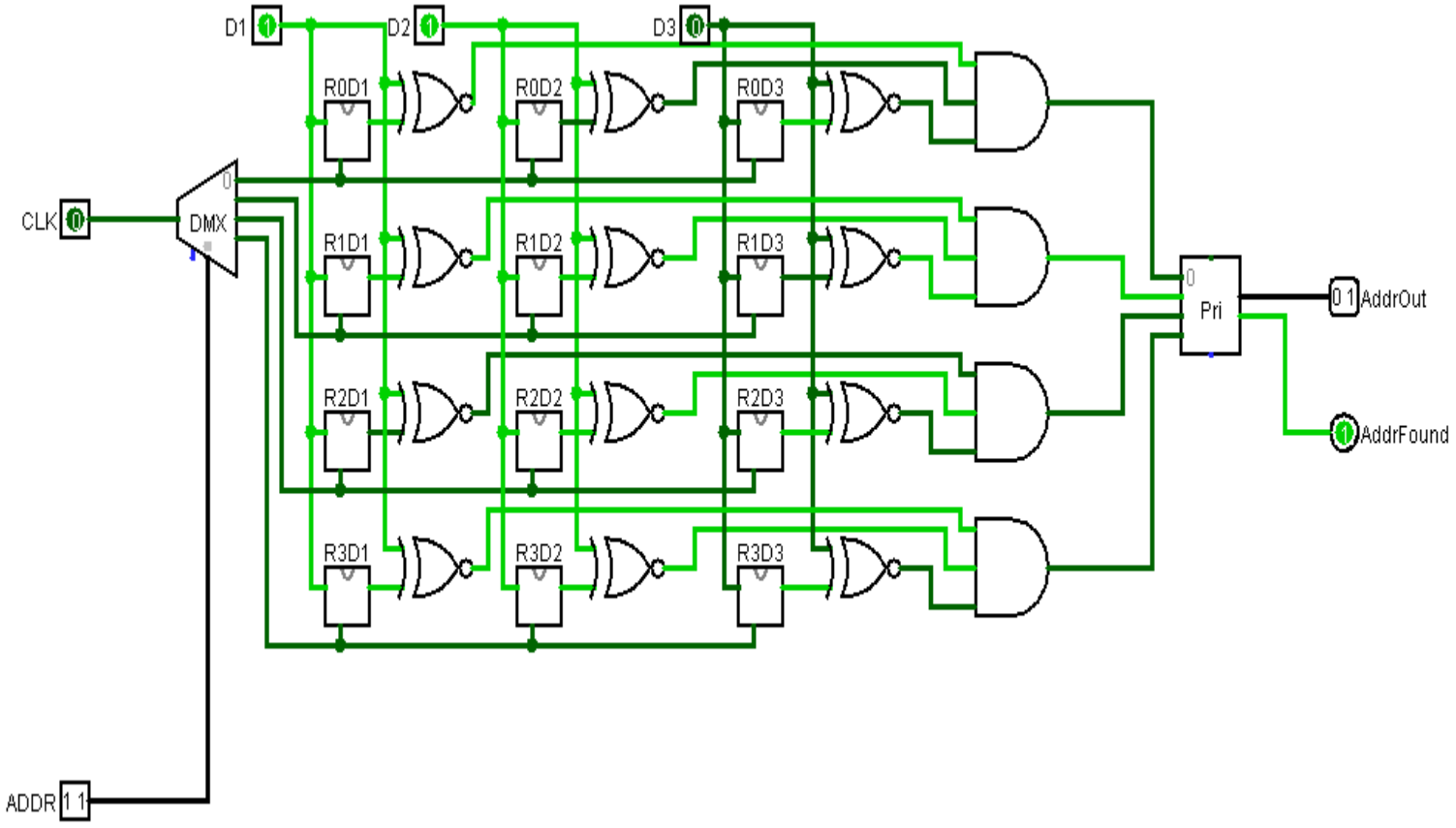
RAM

- Write(address,data)
  - Writes data to the address row
- data=Read(address)
  - Reads data at the address row

CAM

- Write(address,data)
  - Writes data to the address row
- address=Read(data)
  - Returns the address of the row that contains the data
  - Or returns “data not found”

Content Addressable Memory (CAM)

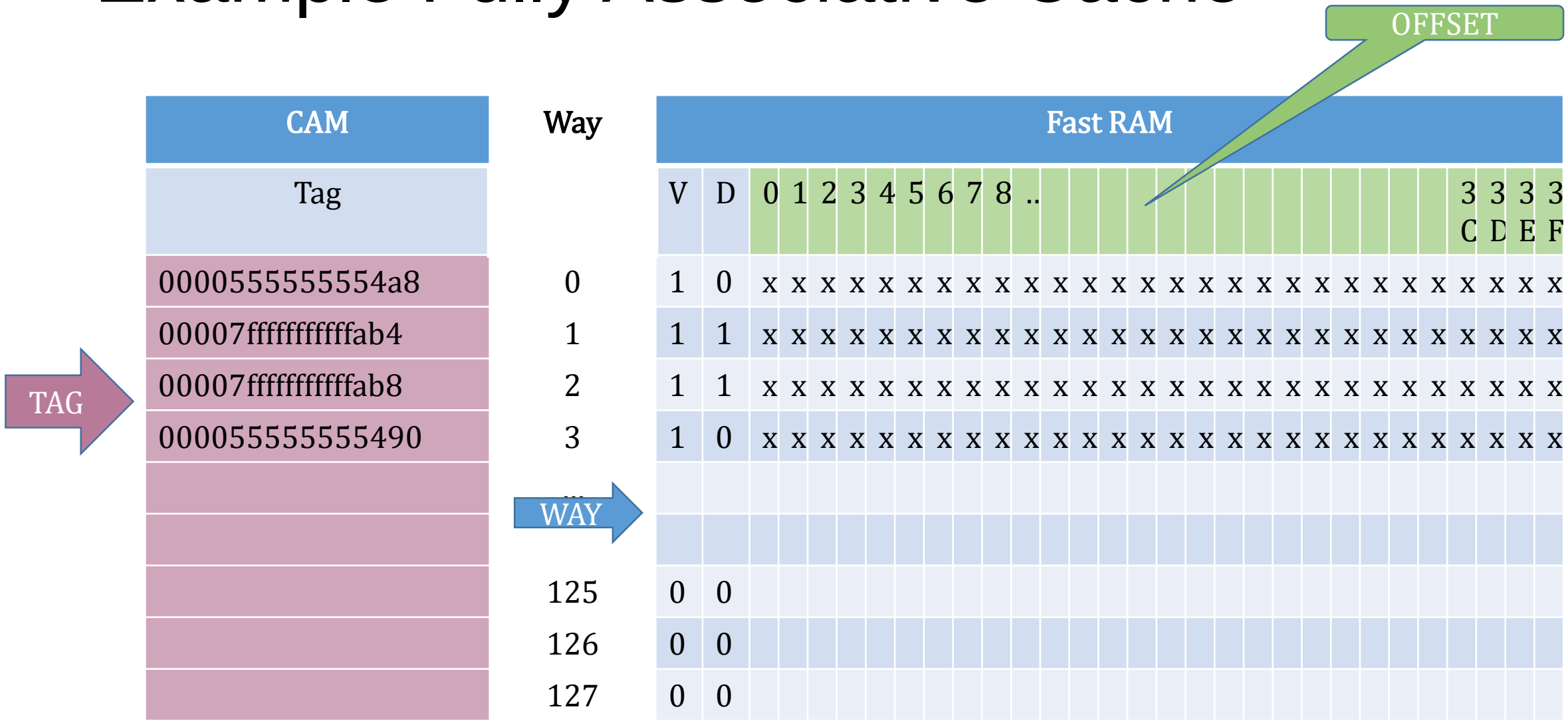


ADDR	DATA		
00	1	0	1
01	1	1	0
10	0	0	1
11	1	1	1

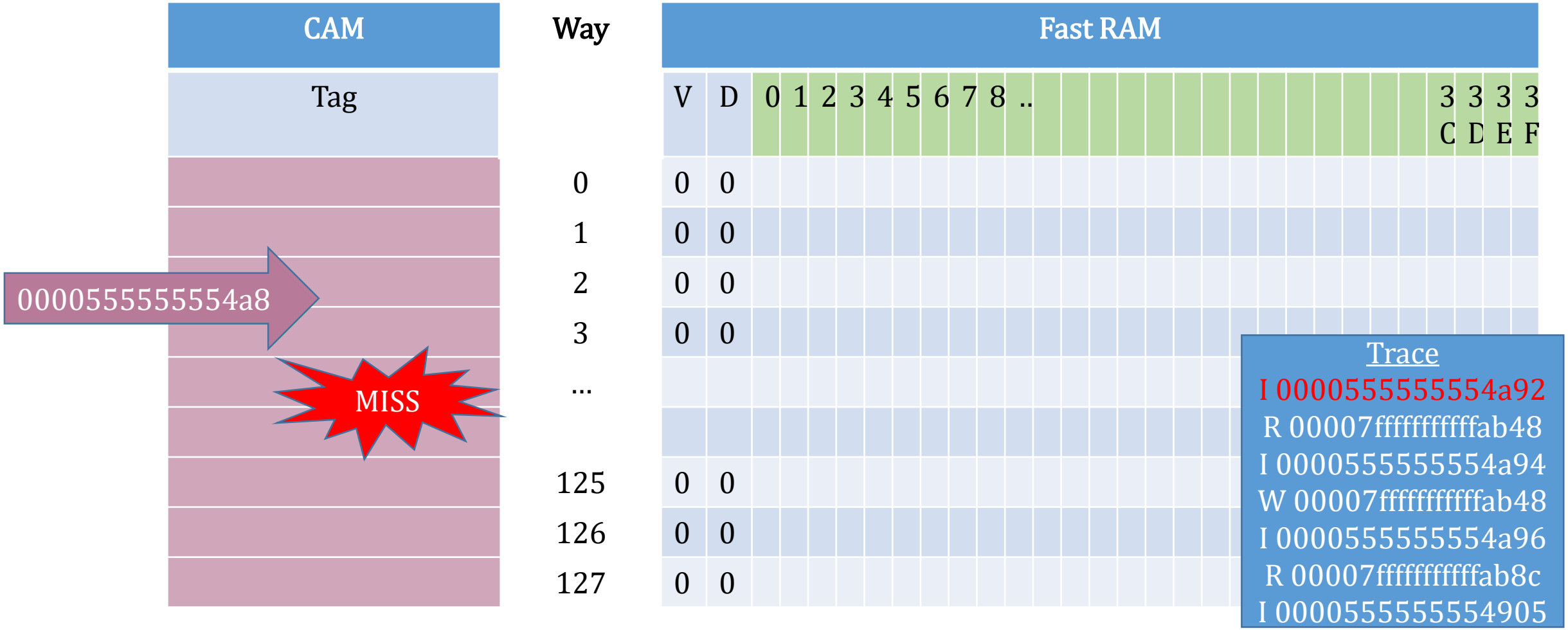
CAM in Fully Associative Cache

- Keep tags in CAM – data width = tag size
- Address in CAM is the way index in the cache
  - When main memory written to cache, also write its tag to CAM
- When accessing memory, provide requested tag to CAM
  - CAM may find that tag is not available… cache miss
  - CAM may find that tag is available and return way index of the way that contains that data… probable cache hit!

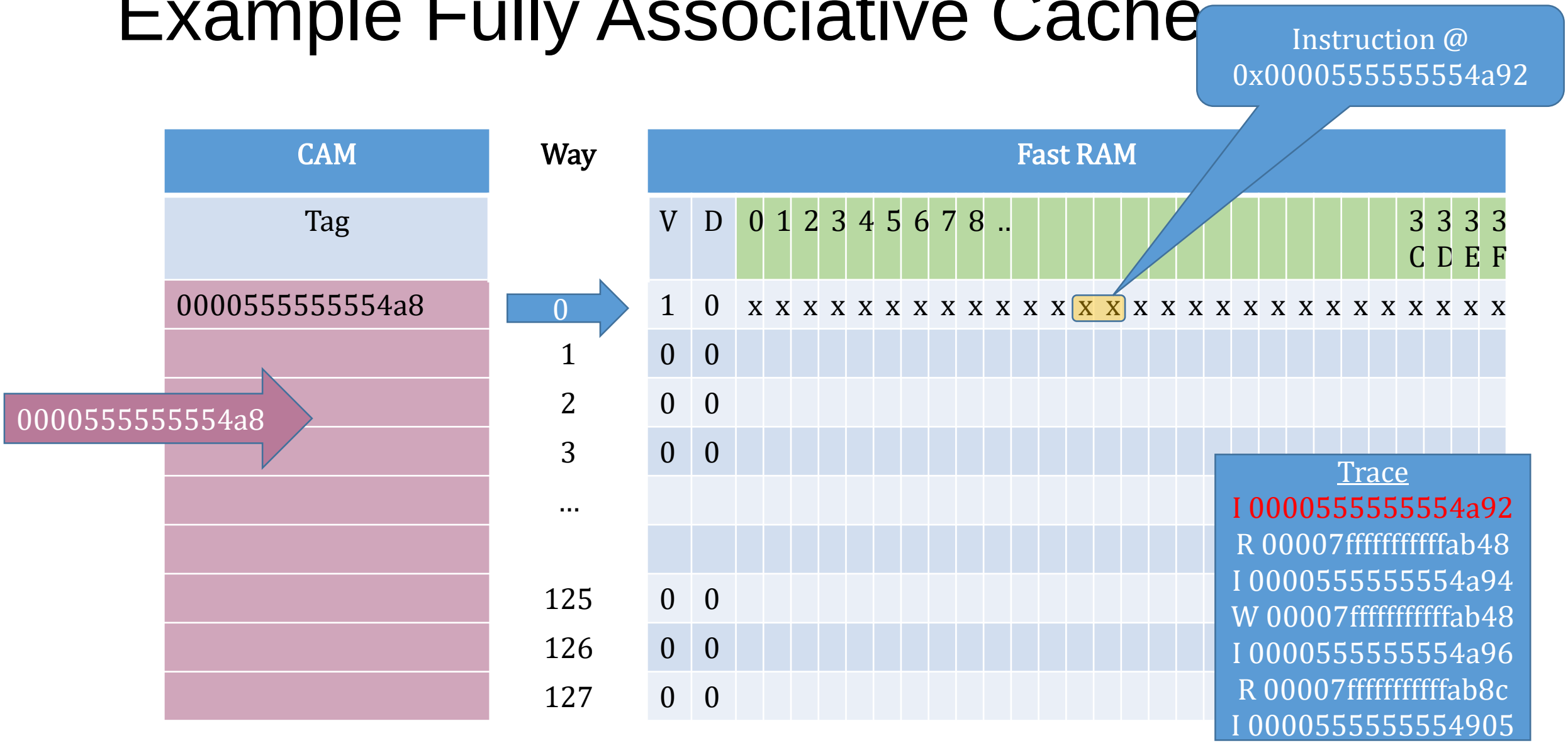
Example Fully Associative Cache



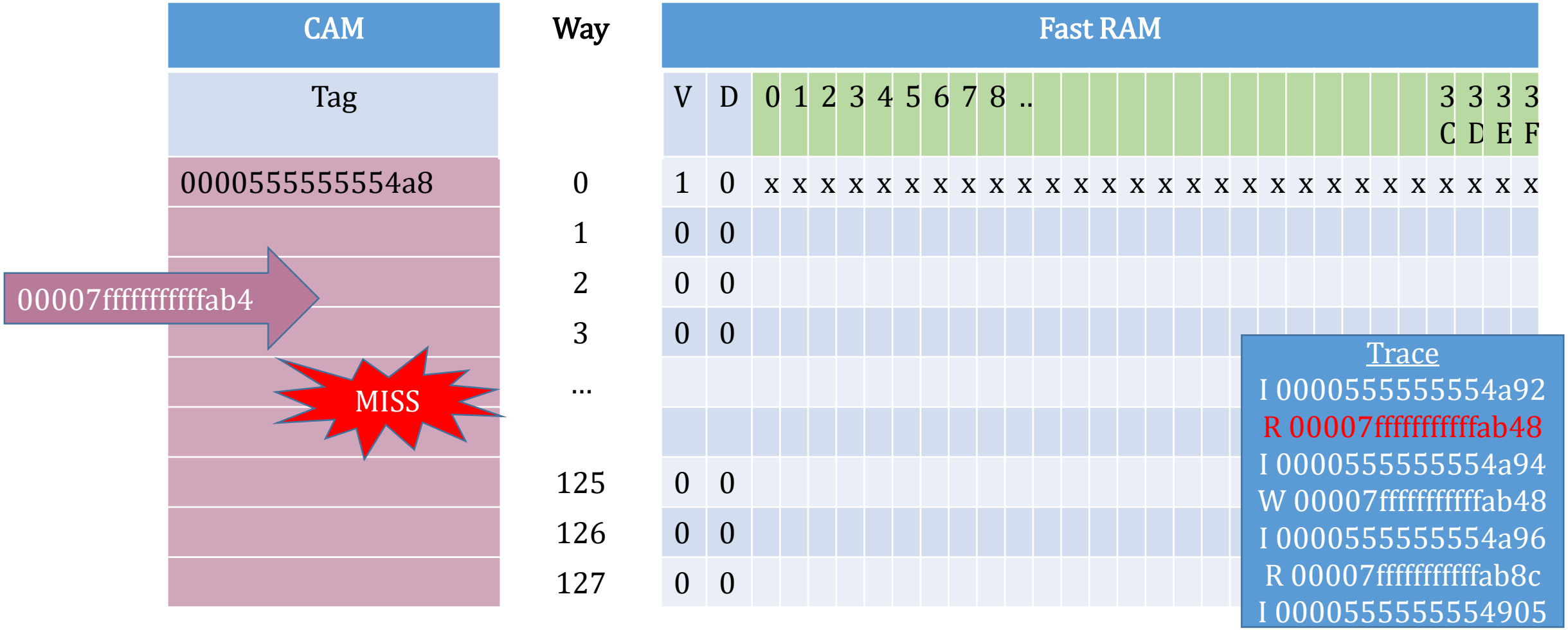
Example Fully Associative Cache



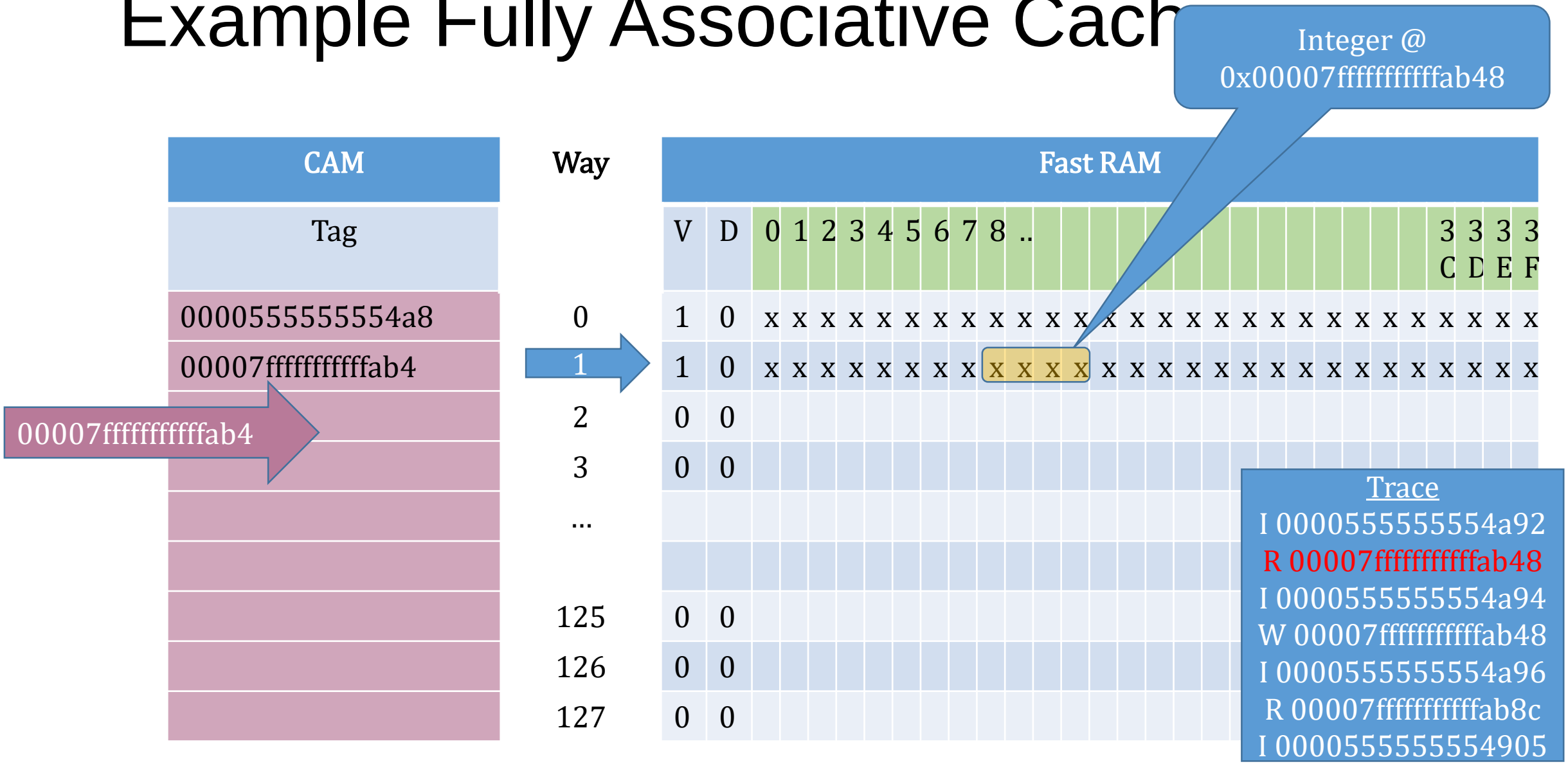
Example Fully Associative Cache



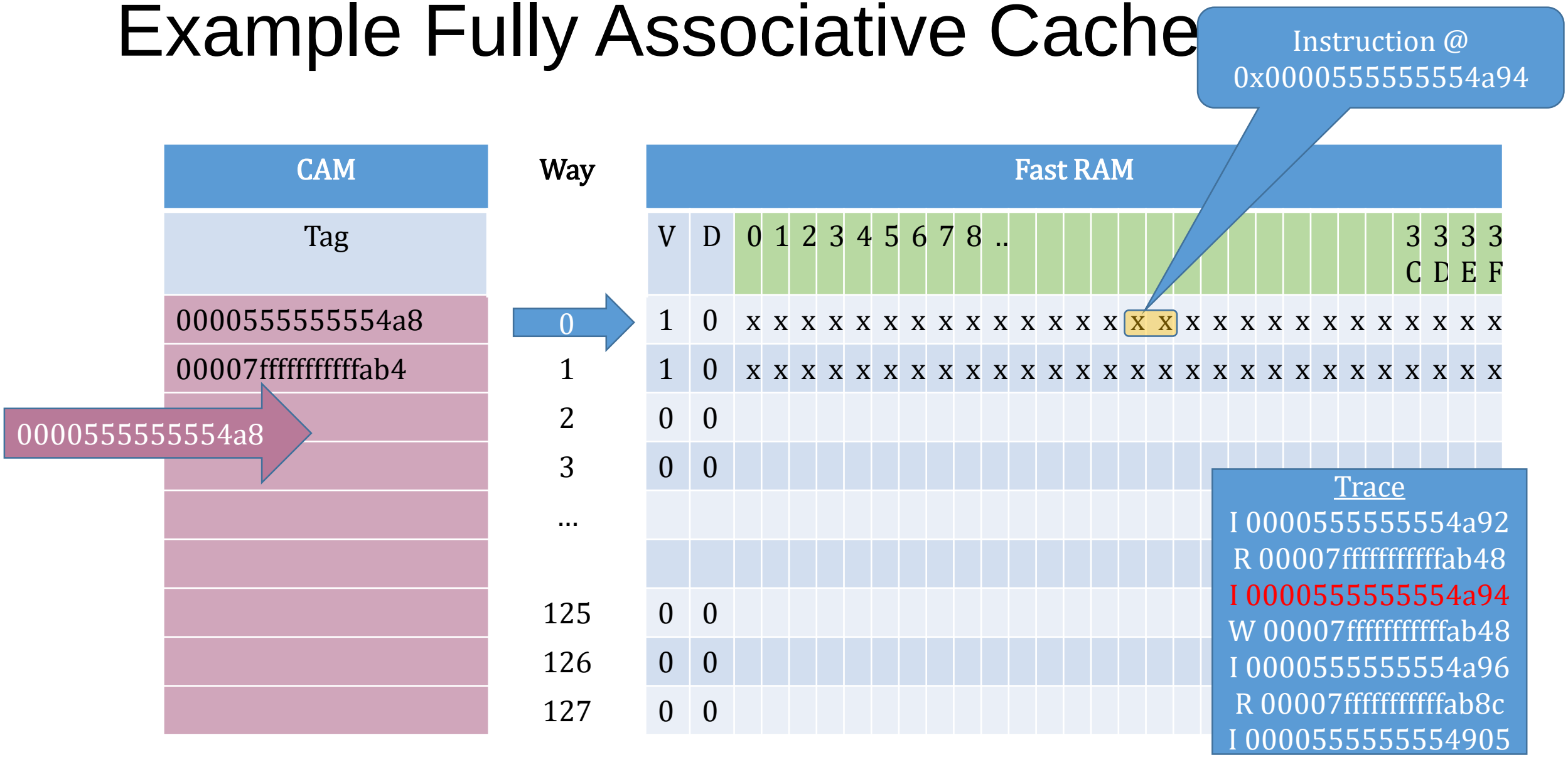
Example Fully Associative Cache



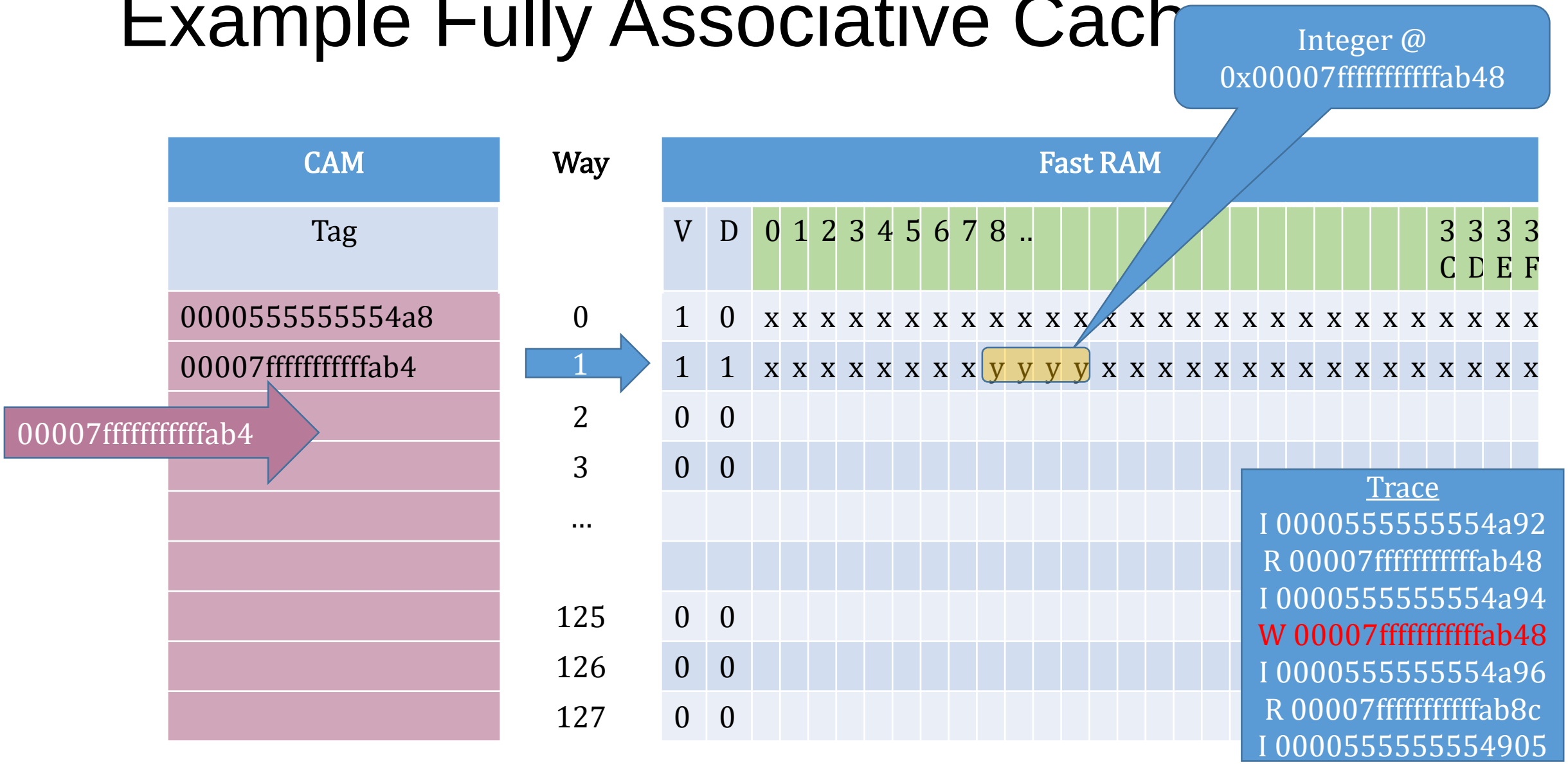
Example Fully Associative Cach



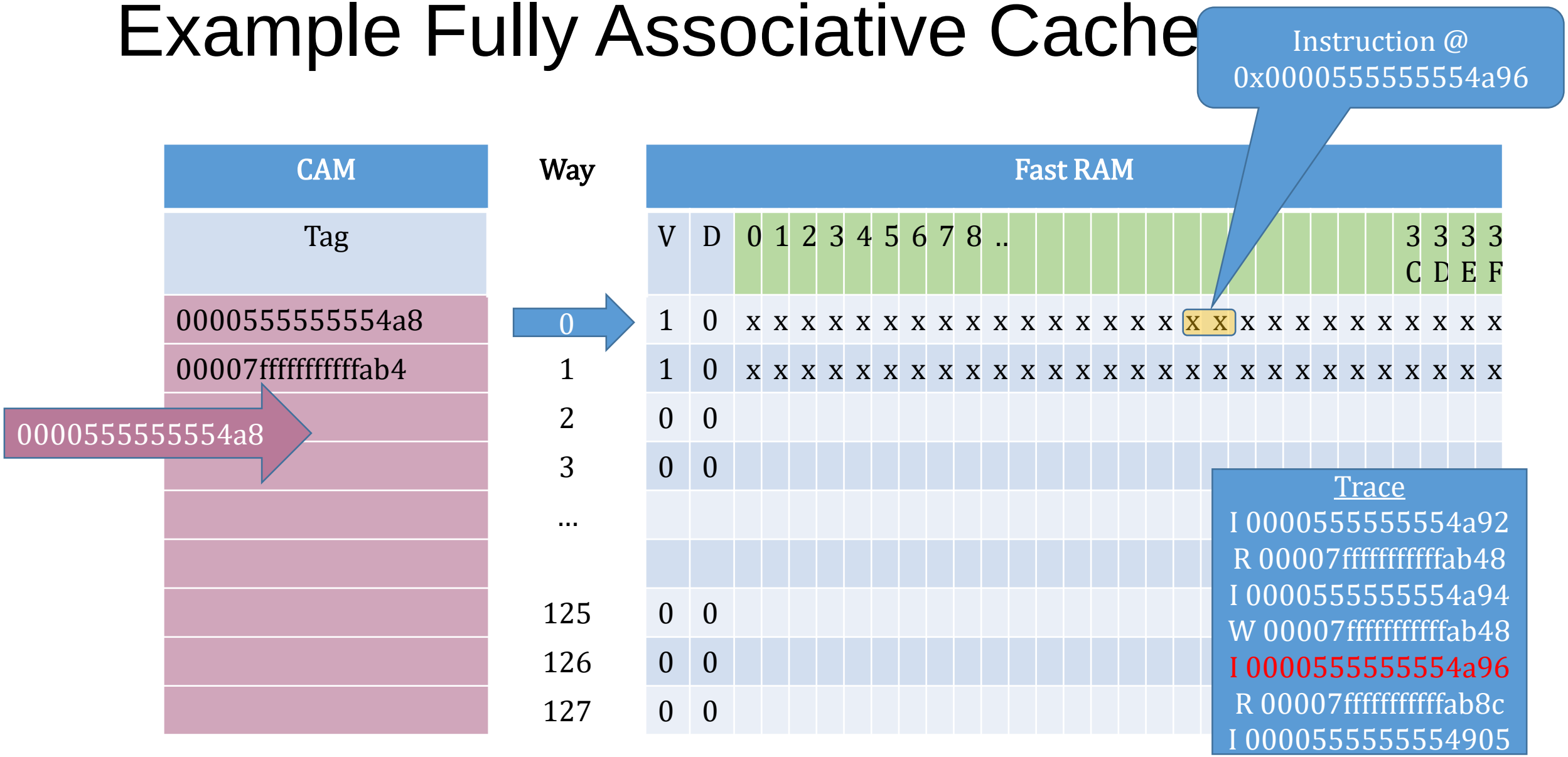
Example Fully Associative Cache



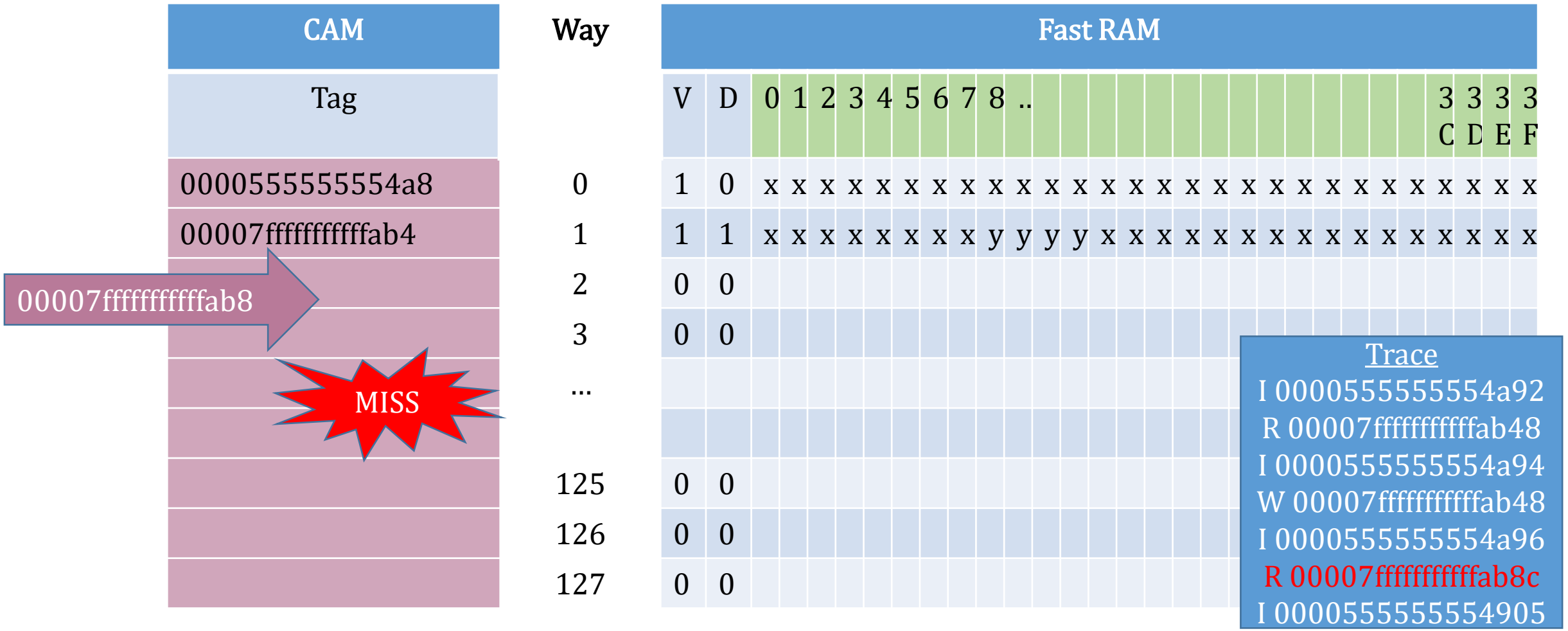
Example Fully Associative Cach



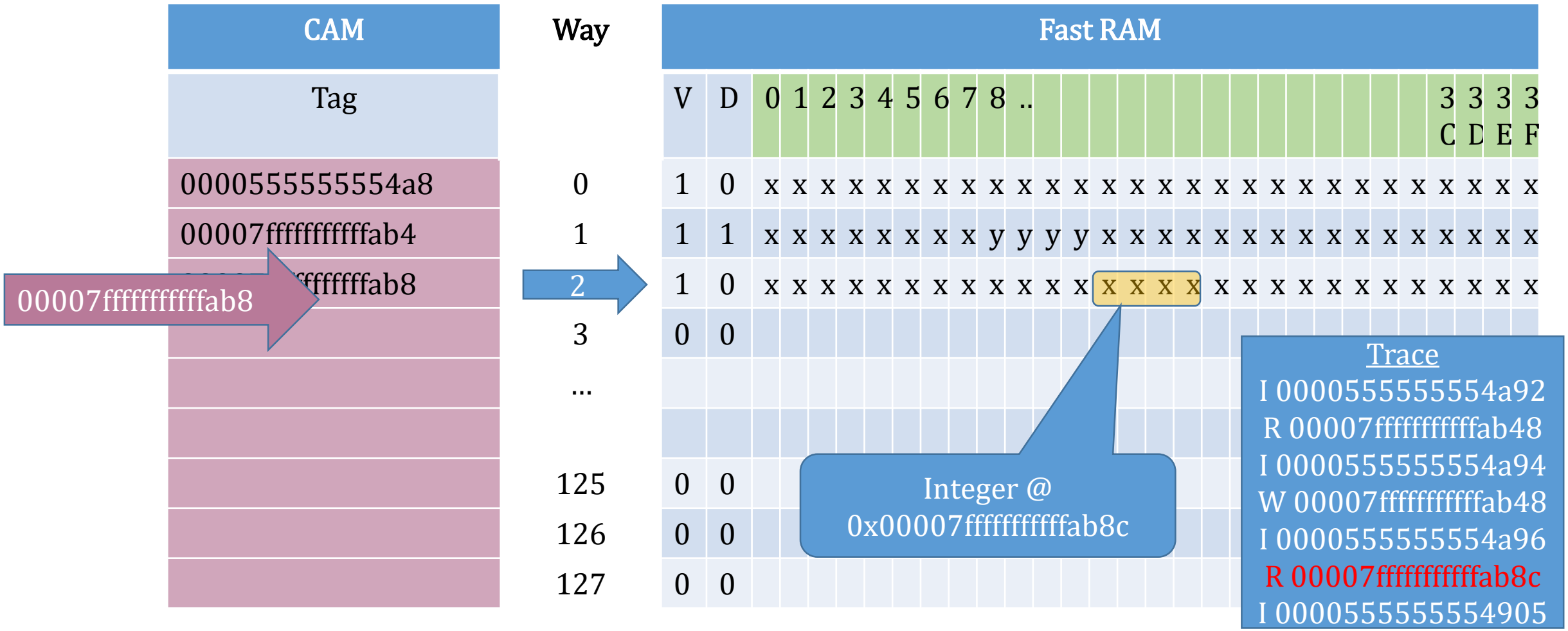
Example Fully Associative Cache



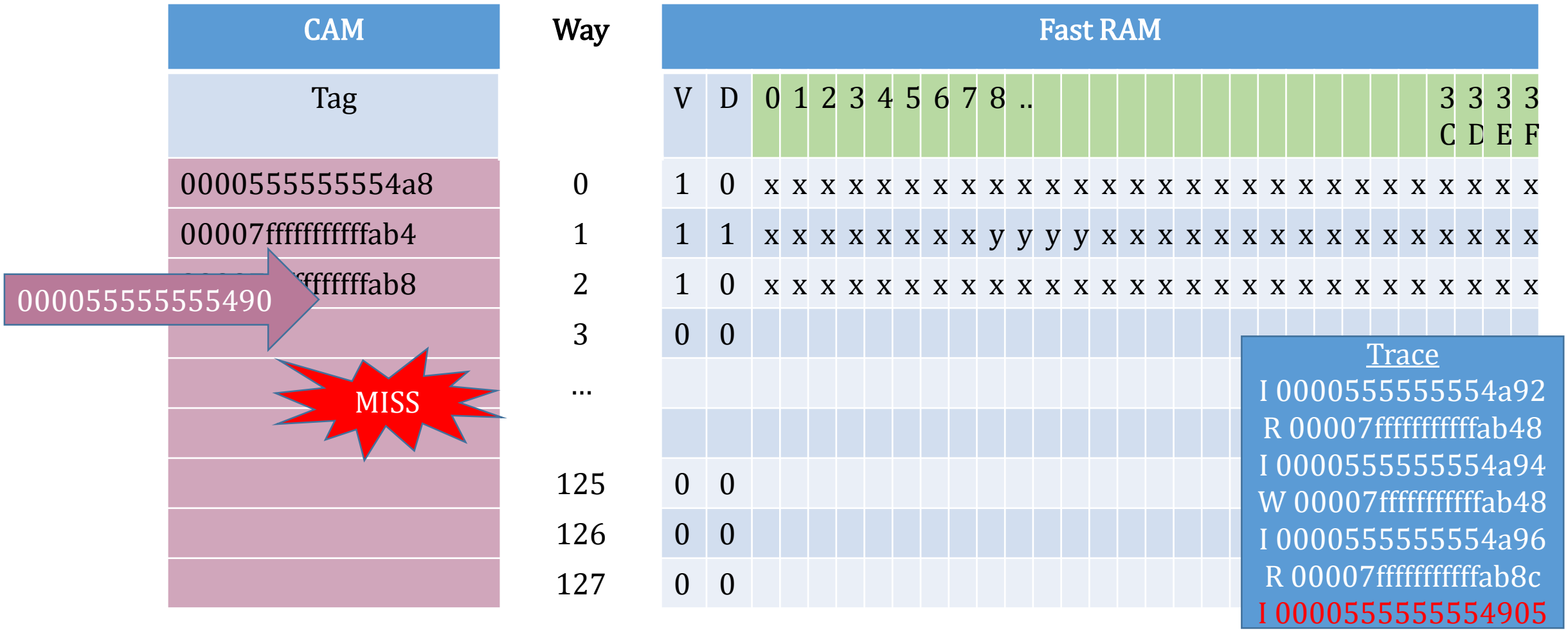
Example Fully Associative Cache



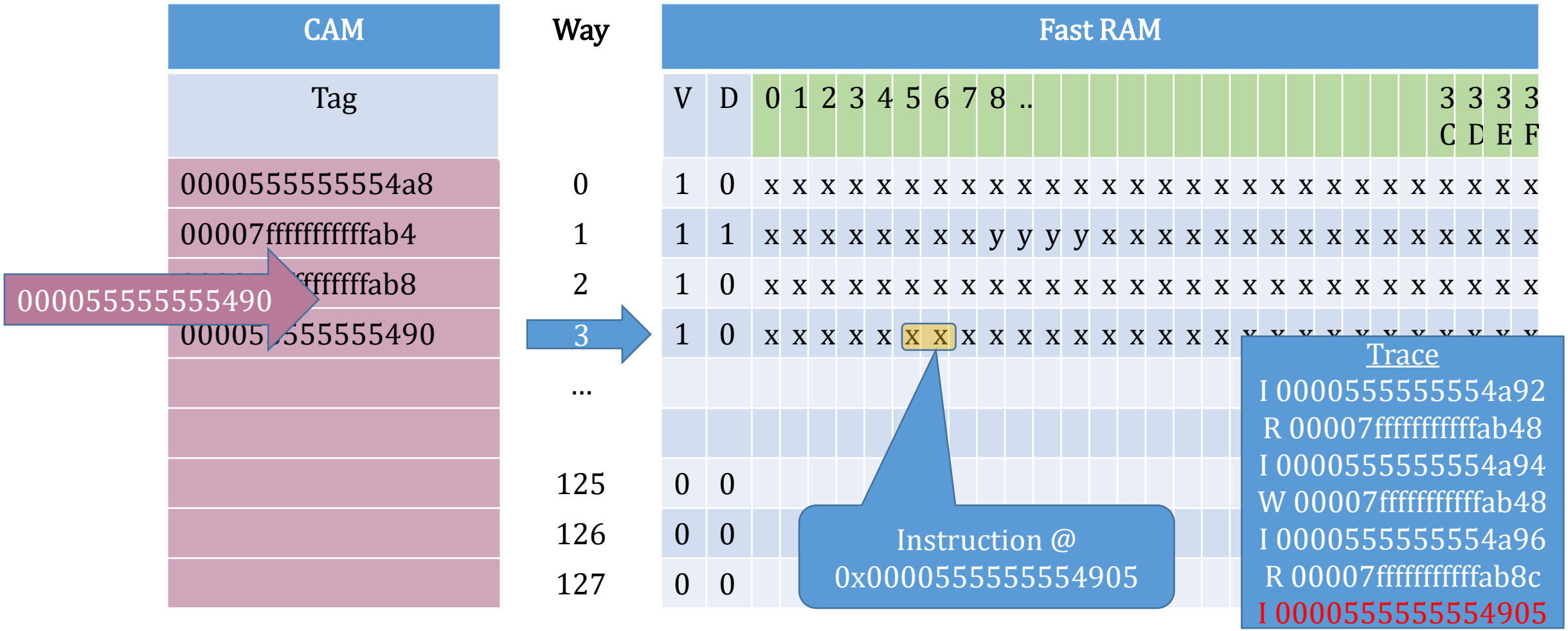
Example Fully Associative Cache



Example Fully Associative Cache



Example Fully Associative Cache



Cache Miss

- Select a “victim” way index … way to replace
  - If there is an unused way (valid flag off) , use it
  - Pick a way that is unlikely to be used in the near future (Project 1)
    - Random, Least Recently Used, Least Frequently Used
- If “dirty” flag is on, copy block from this way to memory
- Write tag to CAM at the chosen way index
- Copy block starting at tag (with block offset zeroed) from memory to the chosen way in Cache RAM
- Turn valid flag for this set on, and dirty bit off

Fully Associative Cache Pros and Cons

Advantages

- Cache always contains most important memory
  - Therefore, Cache Hit Rate is high
  - Therefore, speed is fast

Disadvantages

- Lots of very expensive CAM memory
- Choosing victim can be slow or incorrect
- Need extra data to choose victim

Direct Map Cache

- Define cache *block size* = 2^b the amount of data transferred between cache and RAM memory – most often $64=2^6$ bytes

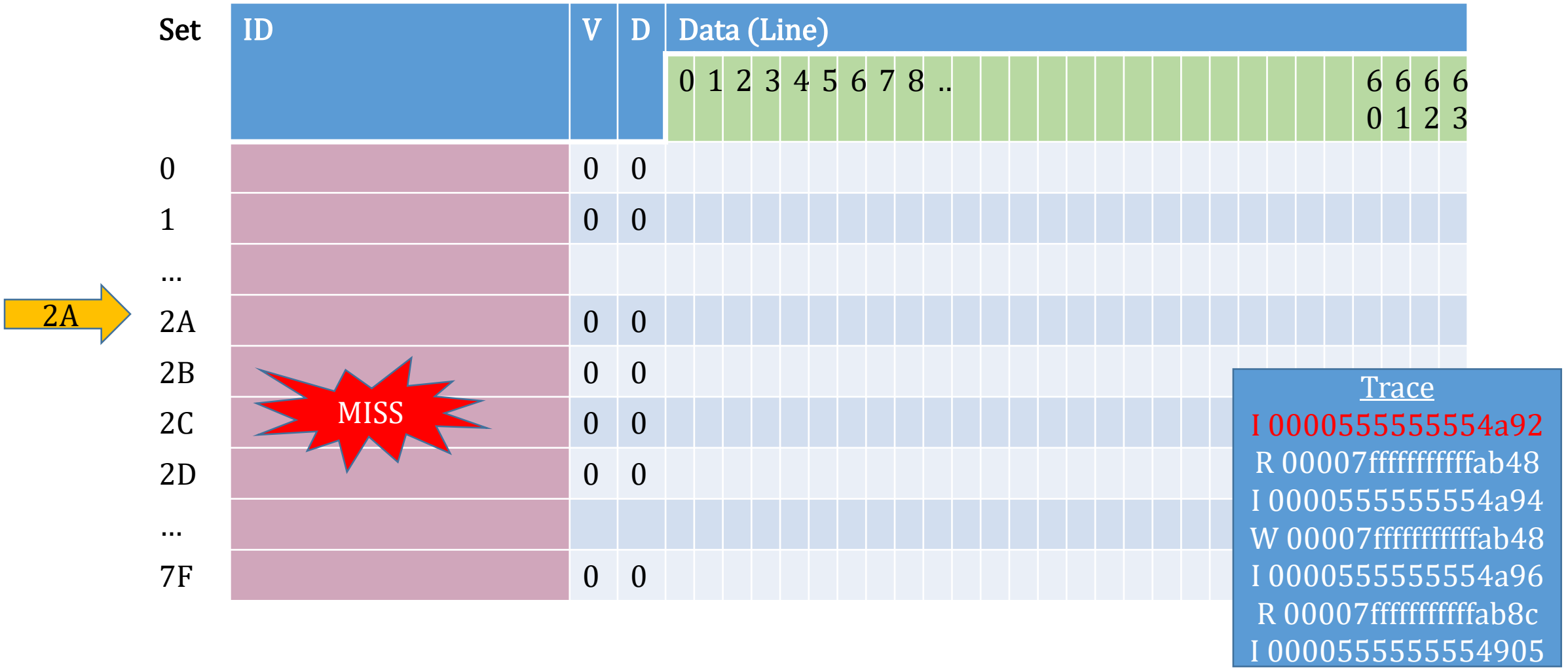
- Divide the cache up into 2^s *sets*
  - Each set contains one way (e.g. 64 byte block of memory)
  - Each set identified by ID *tag* – where this way comes from
  - Each set contains flags, *valid* flag and *dirty* flag
  - For instance, an 8K direct map cache contains $128=2^7$ sets

Address Sub-Fields

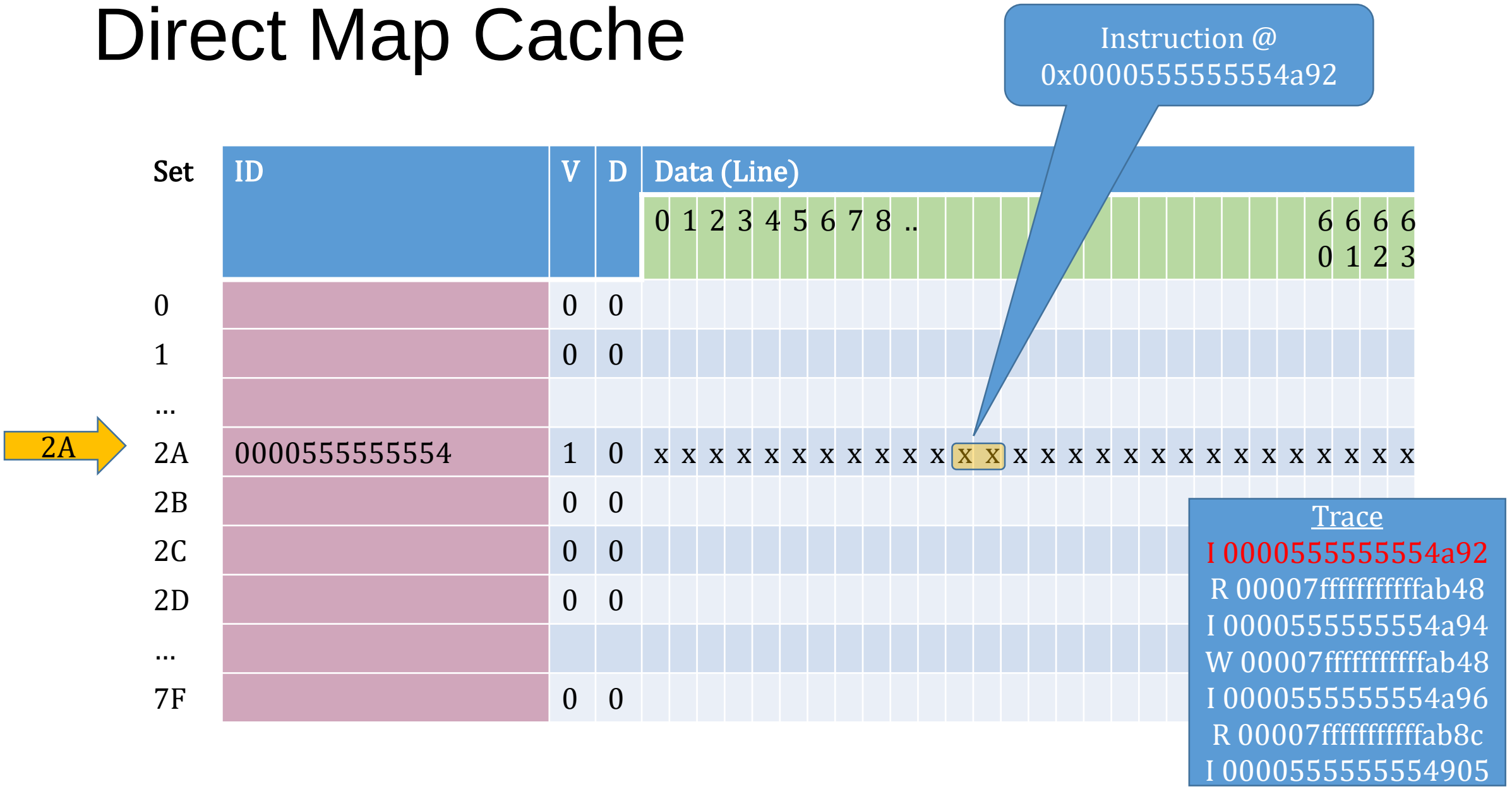
- Divide each memory address into three sub-fields
  - Tag – High order bits - where in memory this block comes from
  - Set Index – s intermediate bits -which set this block is associated with
  - Block Offset – b Low order bits that define the offset within a block

2^{63}	2^{62}	…	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{63}	b_{62}	…	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
Tag					Set Index							Block Offset					
000055555554[a92]					0	1	0	1	0	1	0	0	1	0	0	1	0
					2			A				1		2			
00007fffffffffa[b48]					0	1	0	1	1	0	1	0	0	1	0	0	0
					2			C				0		8			
00007fffffffffa[b8c]					0	1	0	1	1	1	0	0	0	1	1	0	0
					2			D				0		C			

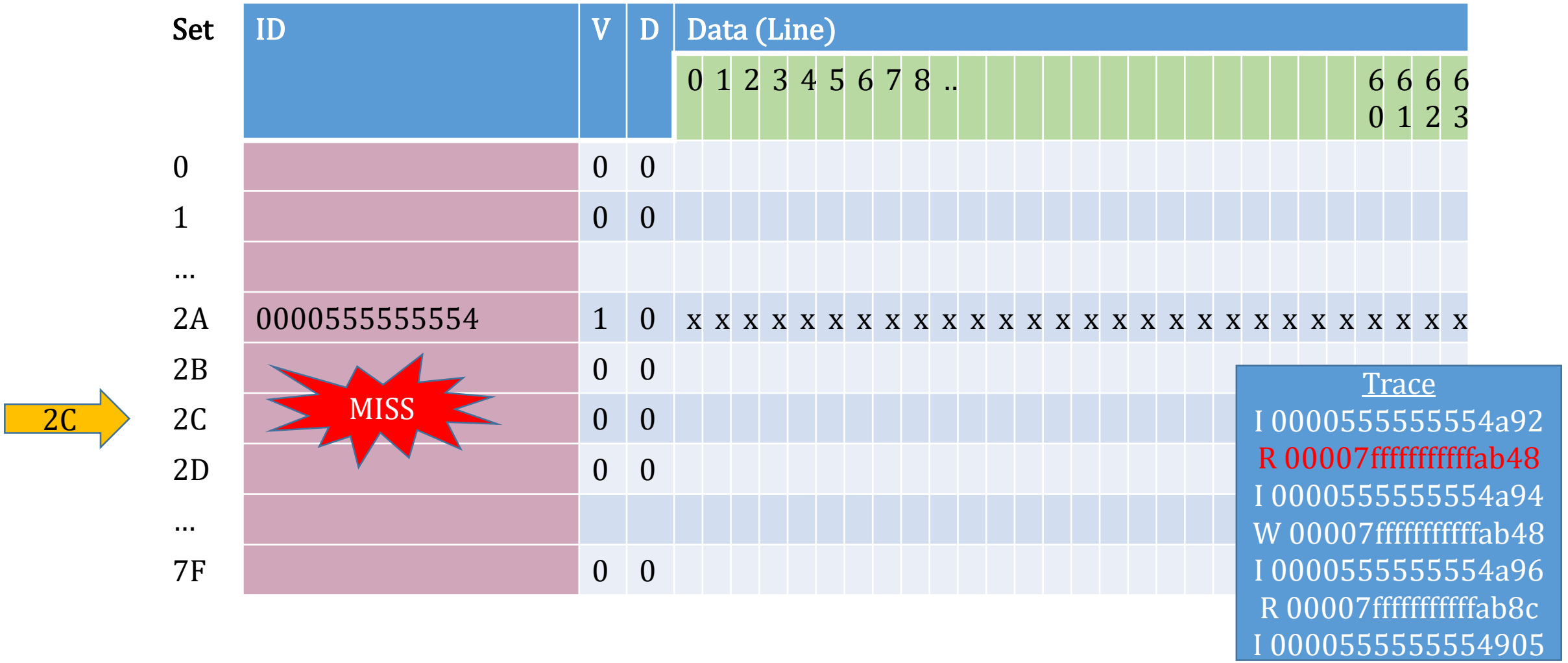
Direct Map Cache



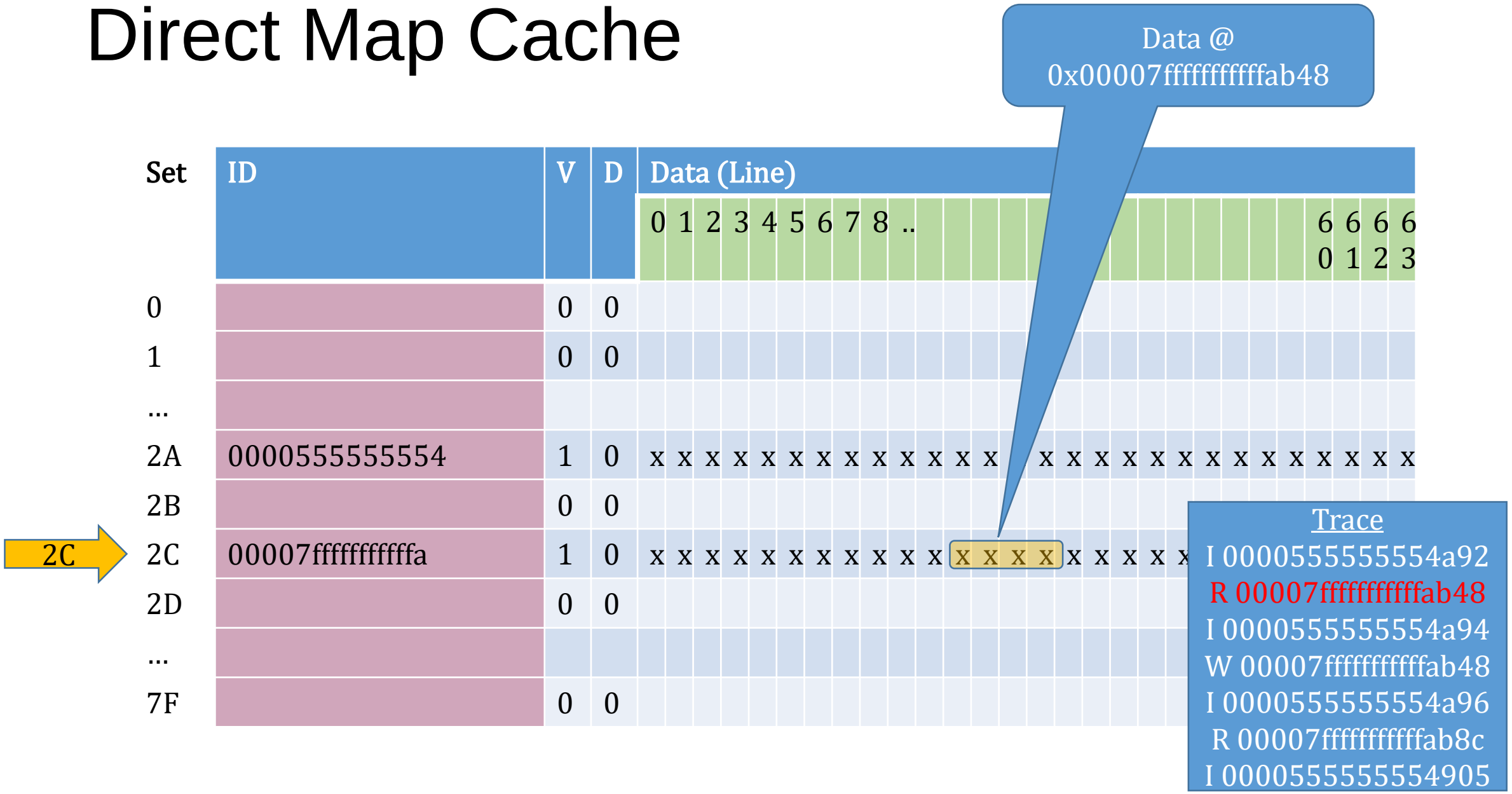
Direct Map Cache



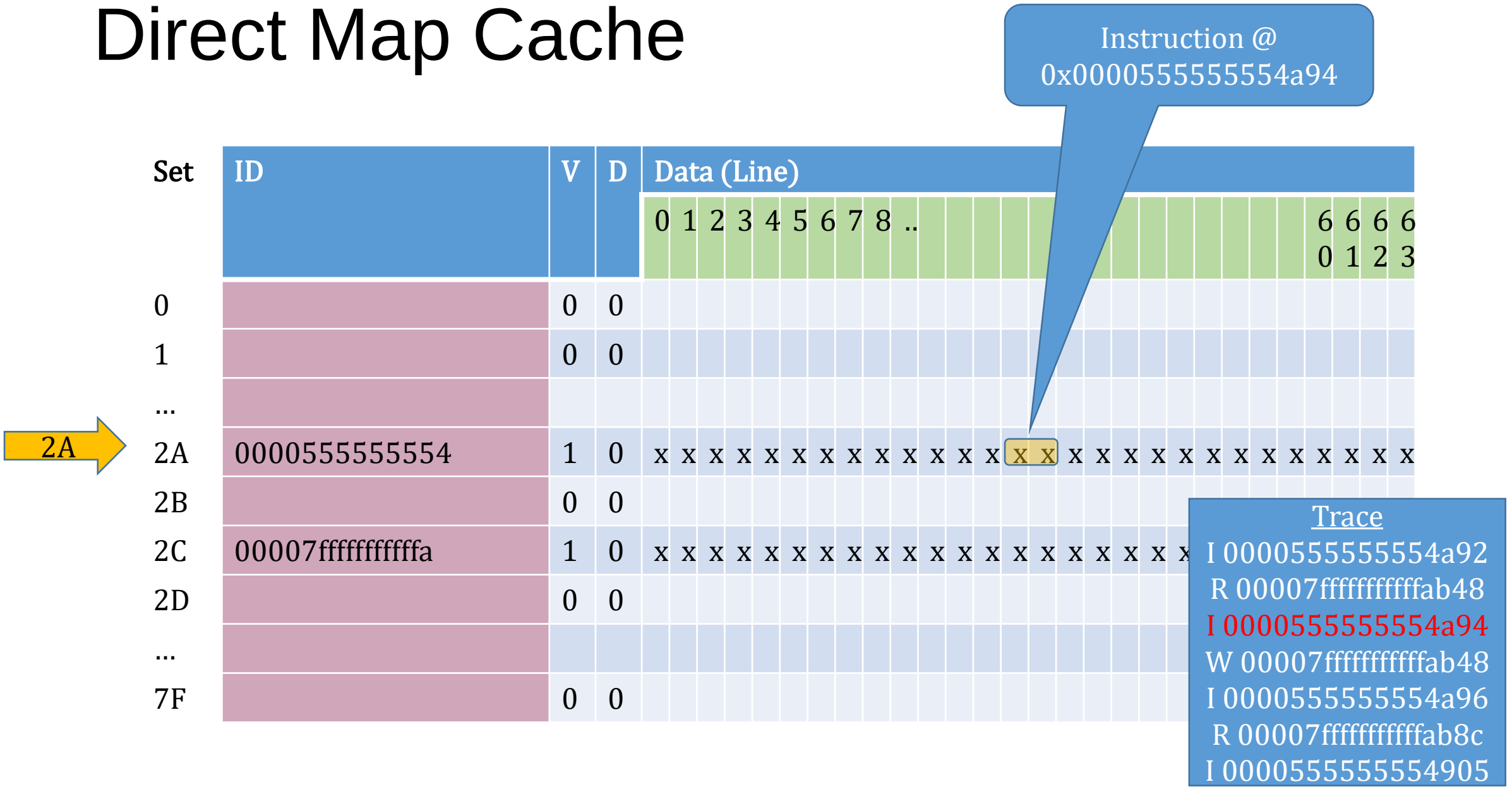
Direct Map Cache



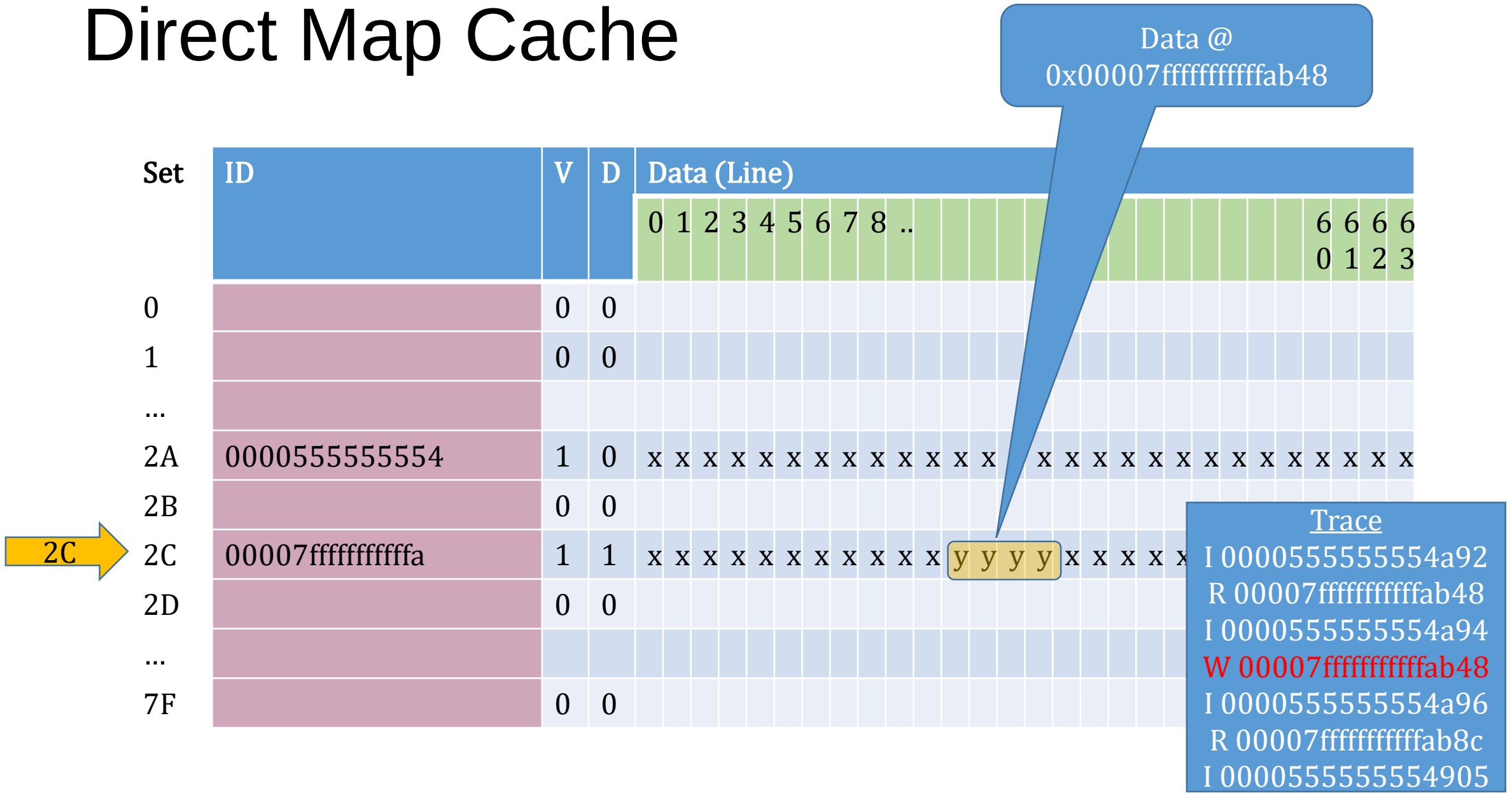
Direct Map Cache



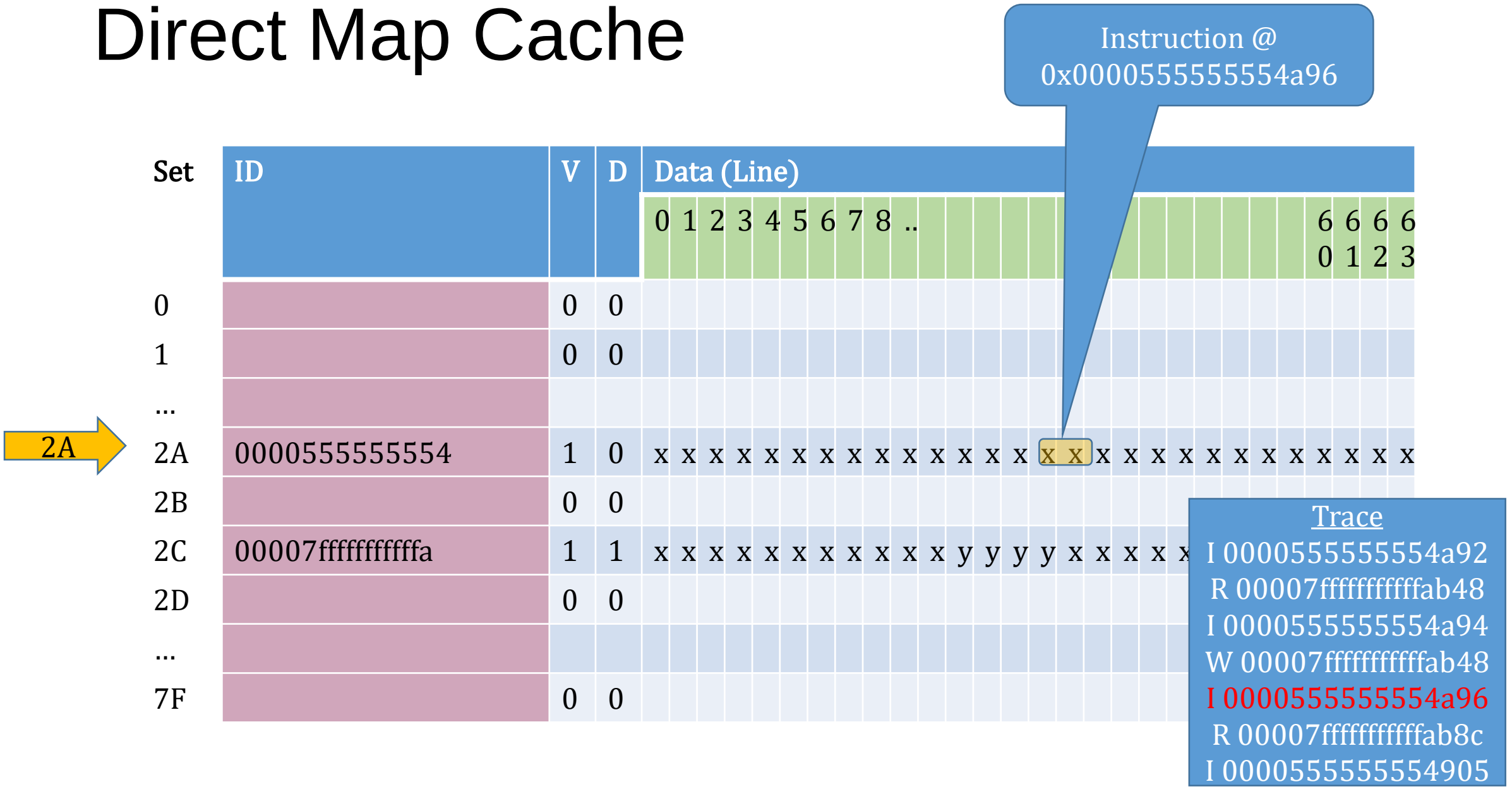
Direct Map Cache



Direct Map Cache



Direct Map Cache



Direct Map Cache

Set	ID	V	D	Data (Line) 0 1 2 3 4 5 6 7 8 .. 60 61 62 63
0		0	0	
1		0	0	
…				
2A	000055555554	1	0	x x x x x x x x x x x x x x x x x x x x x x x x x x x x
2B		0	0	
2C	00007fffffffffa	1	1	x x x x x x x x x x x y y y y x x x x x
2D		0	0	
…				
7F		0	0	

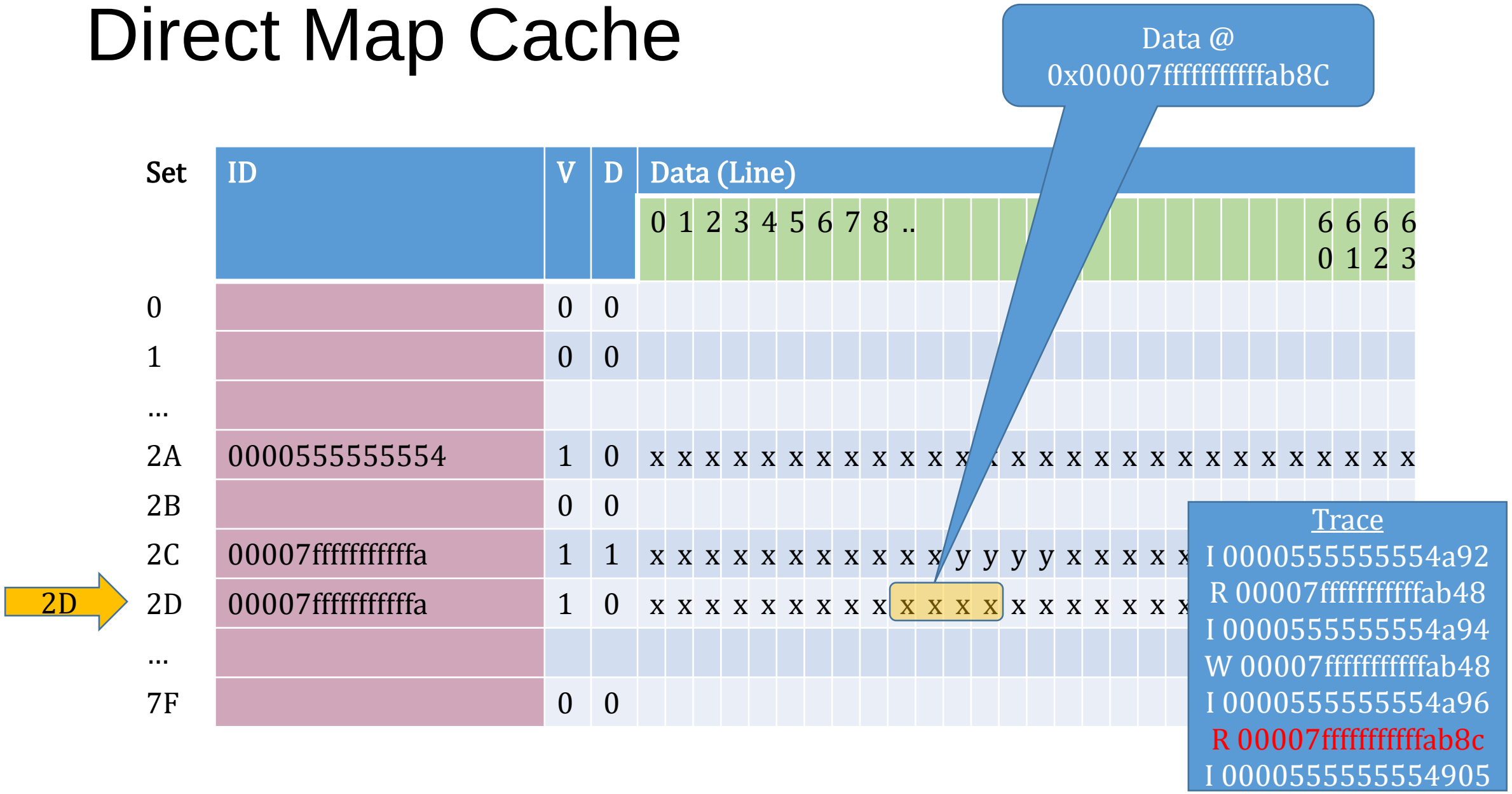
2D

MISS

Trace

I 000055555554a92
R 00007fffffffffab48
I 000055555554a94
W 00007fffffffffab48
I 000055555554a96
R 00007fffffffffab8c
I 000055555554905

Direct Map Cache



On memory access request

1. Use set index to choose correct set
2. If “valid” flag is off, cache miss…
3. Compare tag – if not equal evict block… cache miss…
   - If “dirty” flag is on, write line back to main memory based on tag and set
   - Read main memory line from new tag/set into this set
4. Use block offset to access data in cache
   - If memory write, turn on “dirty” flag for this set.

Direct Map View of Real Memory

Address			64 byte block of memory																					
TAG	SET	OFFSET	00	01	02	…																	…	3E	3F
FFFF	7F	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
FFFF	7E	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
…																									
FFFF	00	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
…	…	…																							
0001	01	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0001	00	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0000	7F	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
…	…	…																							
0000	01	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0000	00	00-3F	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Direct Map Hit Rates

- Sequential access – Cache miss after $8{,}192 = 0x2000 = 2^{13}$ bytes
  - 99.988% hit rate … pretty good!

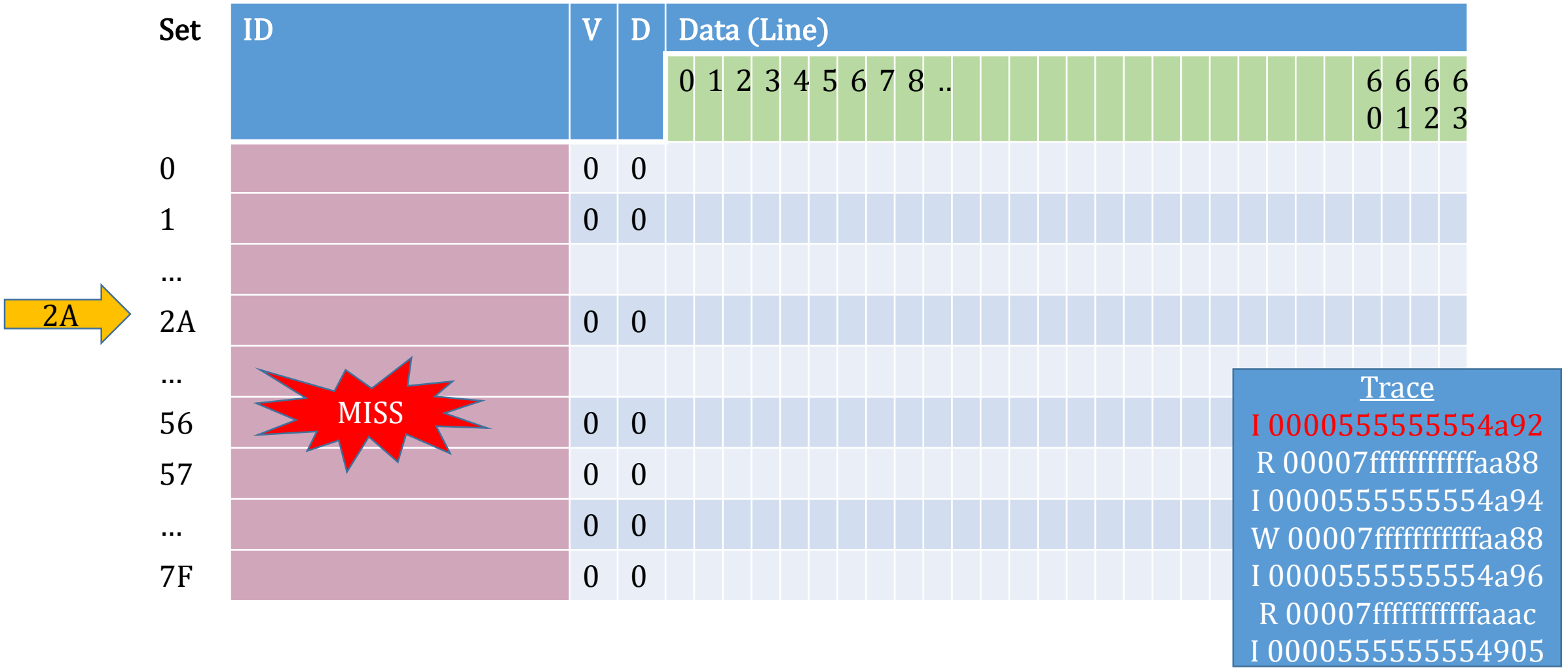
- Problems occur when two access trends overlap
  - For instance, if binary code is stored at 0x000000000ca4000
  - data is stored at 0x7FFFFFFFFF64000
  - BOTH MAP TO SET 0!
  - Assuming alternate code/data memory access, cache miss EVERY time!
  - 0.000 hit rate … pretty bad!

Address Sub-Fields

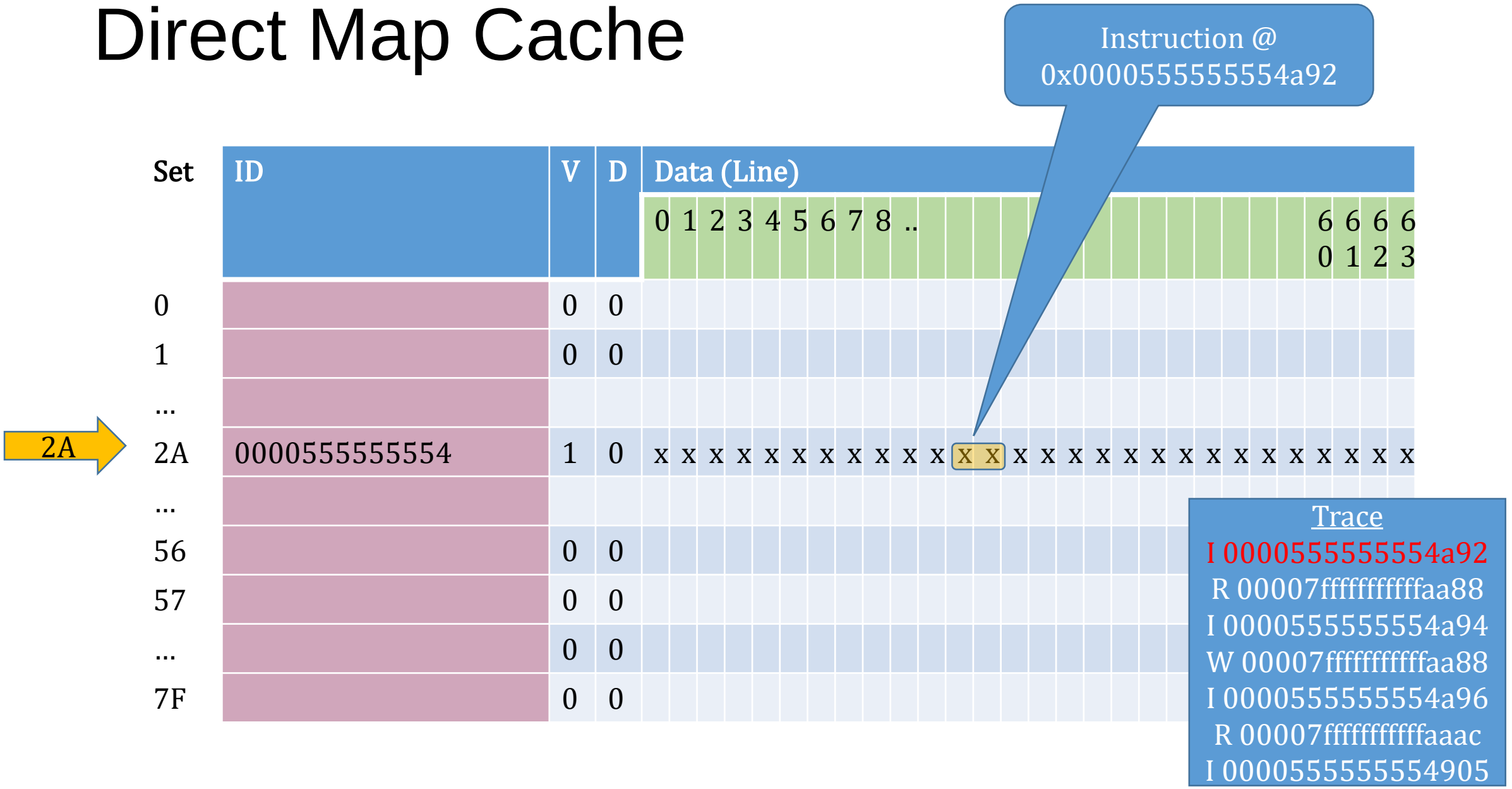
- Divide each memory address into three sub-fields
  - Tag – High order bits - where in memory this block comes from
  - Set Index – s intermediate bits -which set this block is associated with
  - Block Offset – b Low order bits that define the offset within a block

2^{63}	2^{62}	…	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{63}	b_{62}	…	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
Tag					Set Index							Block Offset					
000055555554[a92]					0	1	0	1	0	1	0	0	1	0	0	1	0
					2			A				1		2			
00007fffffffffa[a88]					0	1	0	1	0	1	0	0	0	1	0	0	0
					2			A				0		8			
00007fffffffffa[acc]					0	1	0	1	0	1	1	0	0	1	1	0	0
					2			B				0		C			

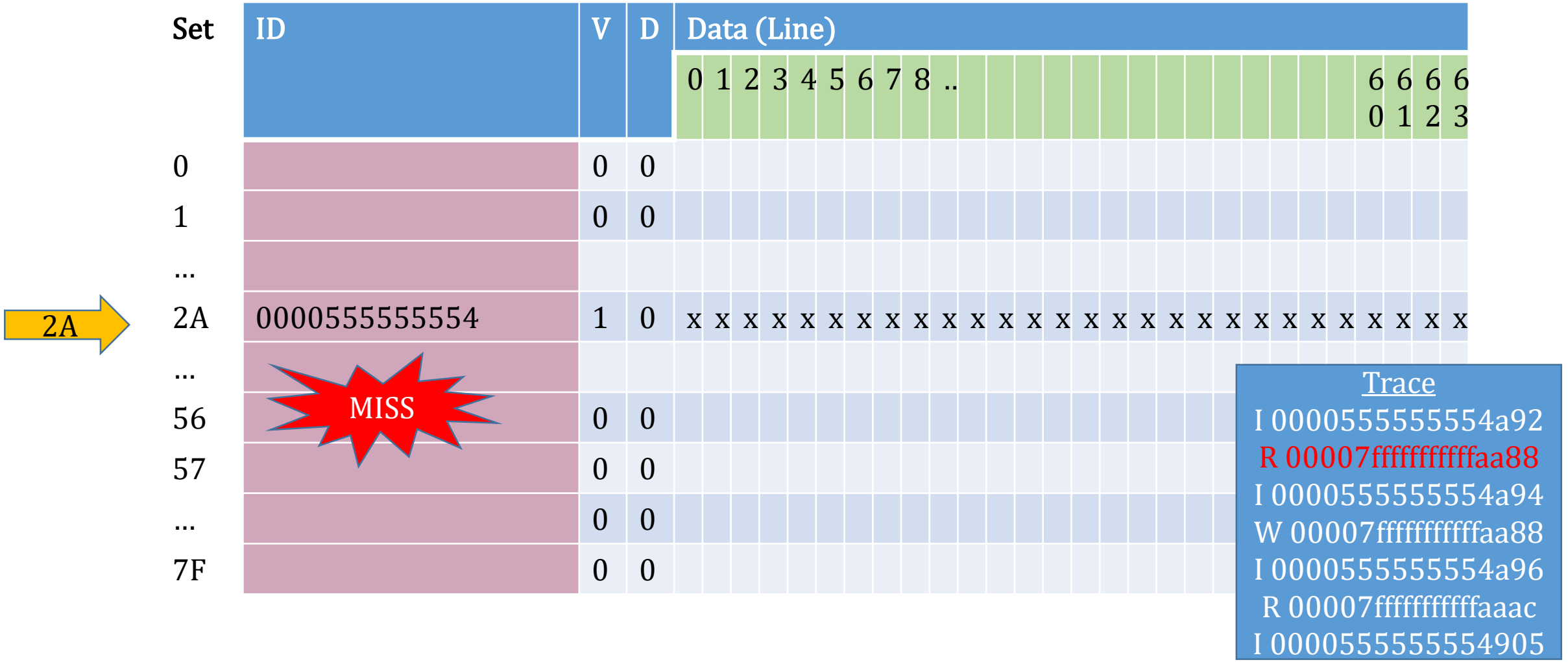
Direct Map Cache Overlap



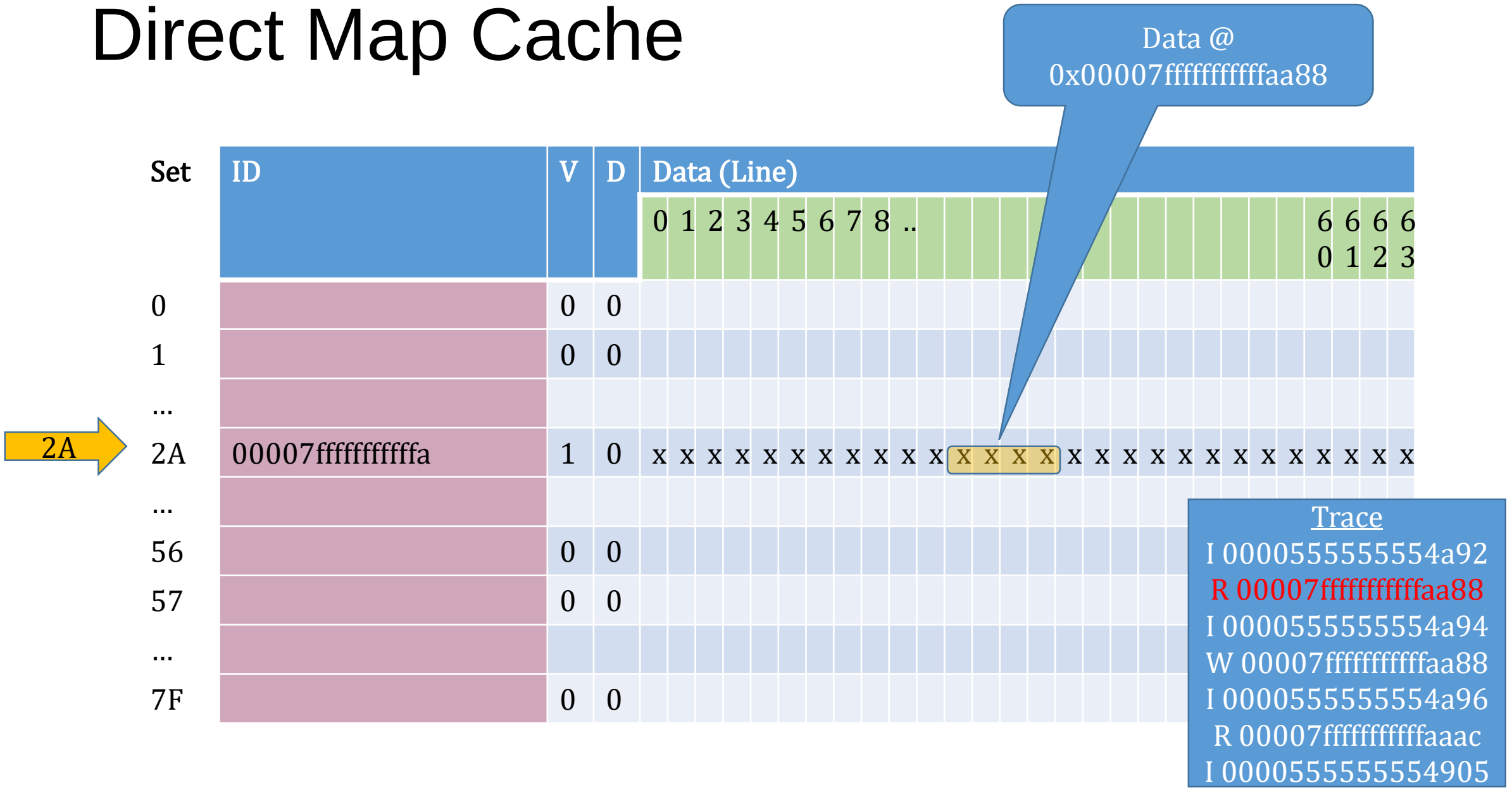
Direct Map Cache



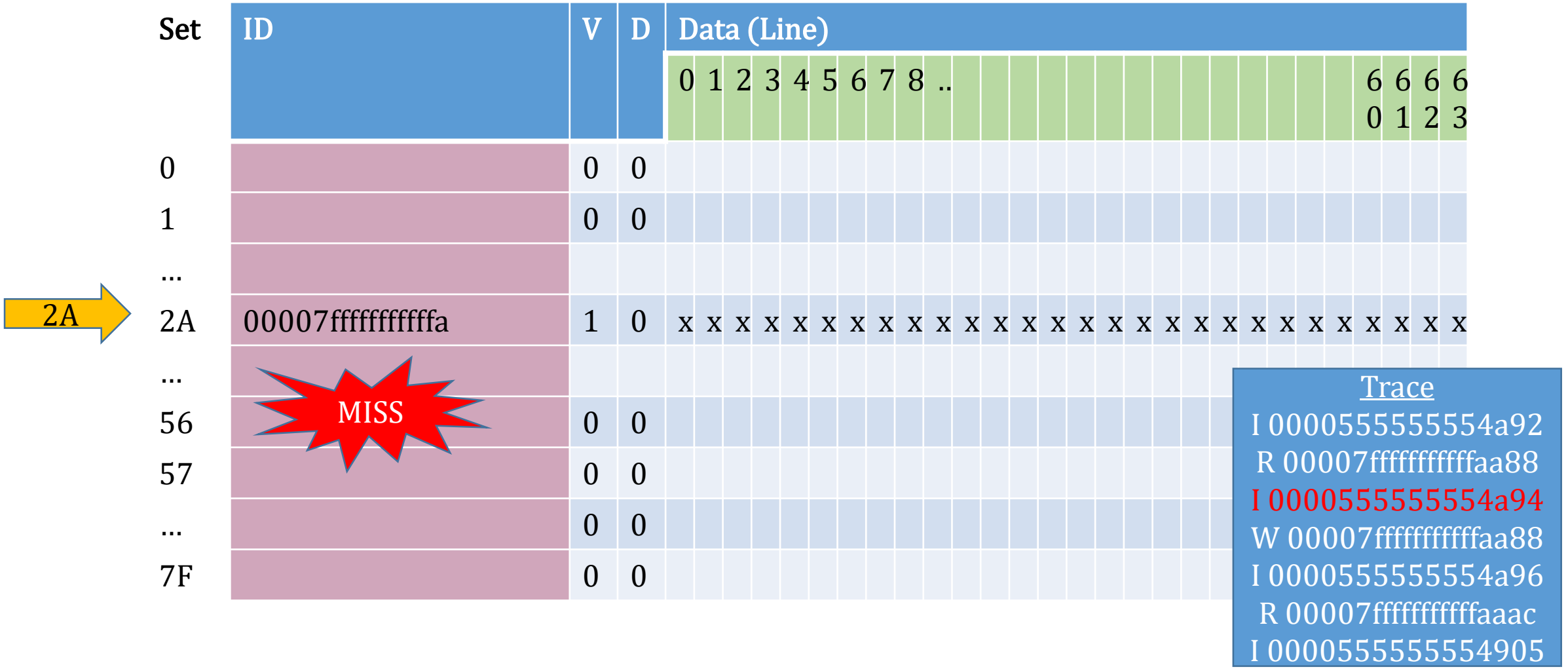
Direct Map Cache



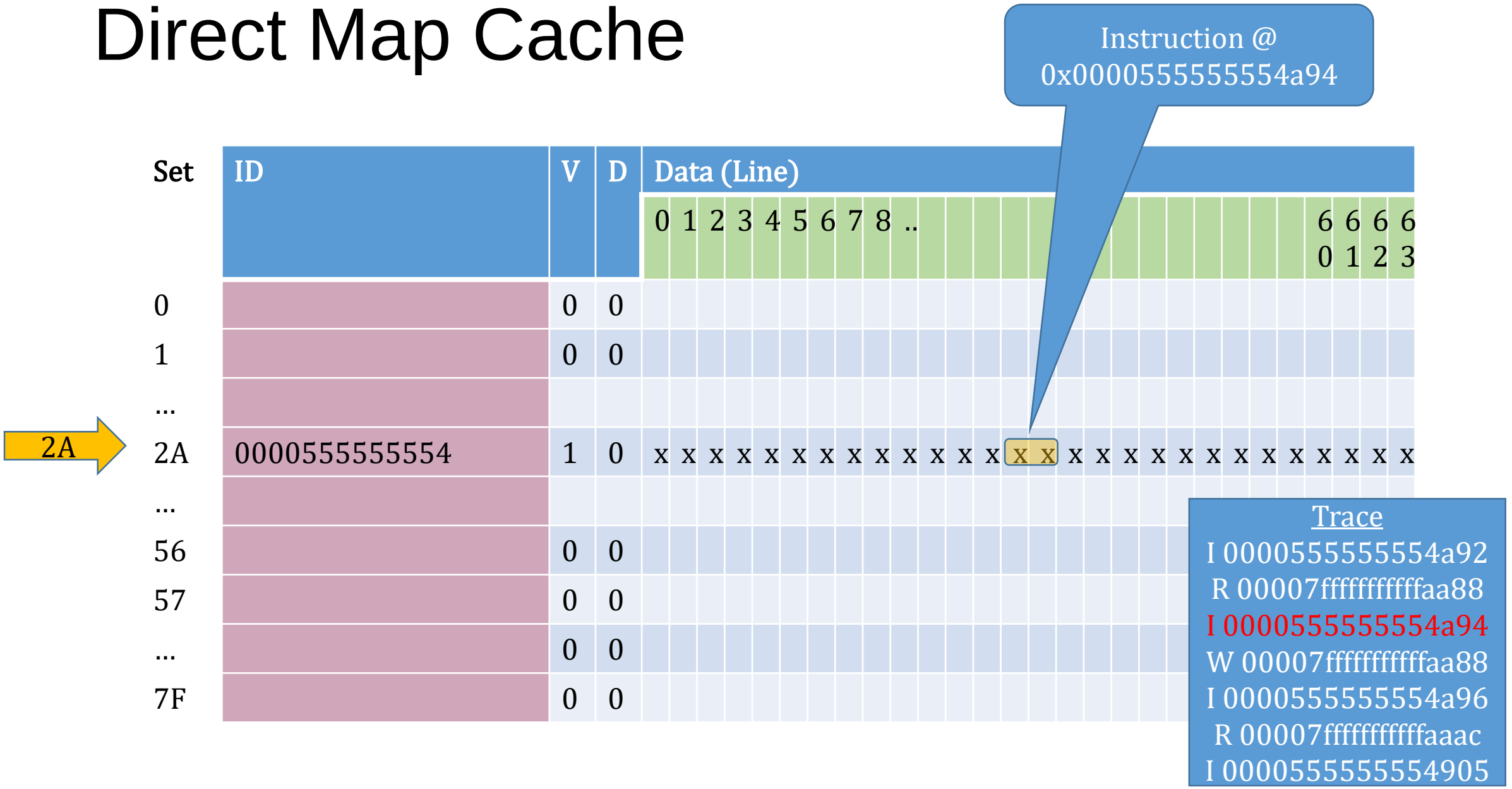
Direct Map Cache



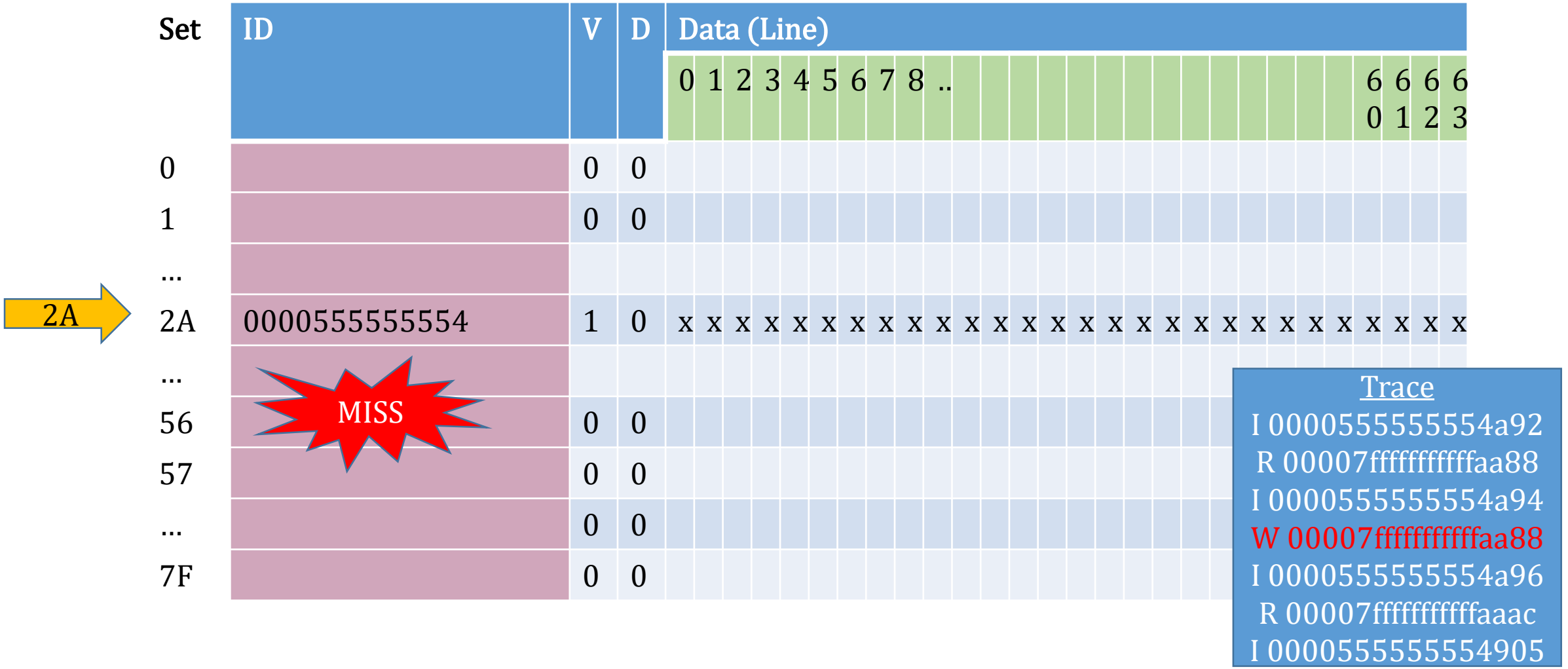
Direct Map Cache



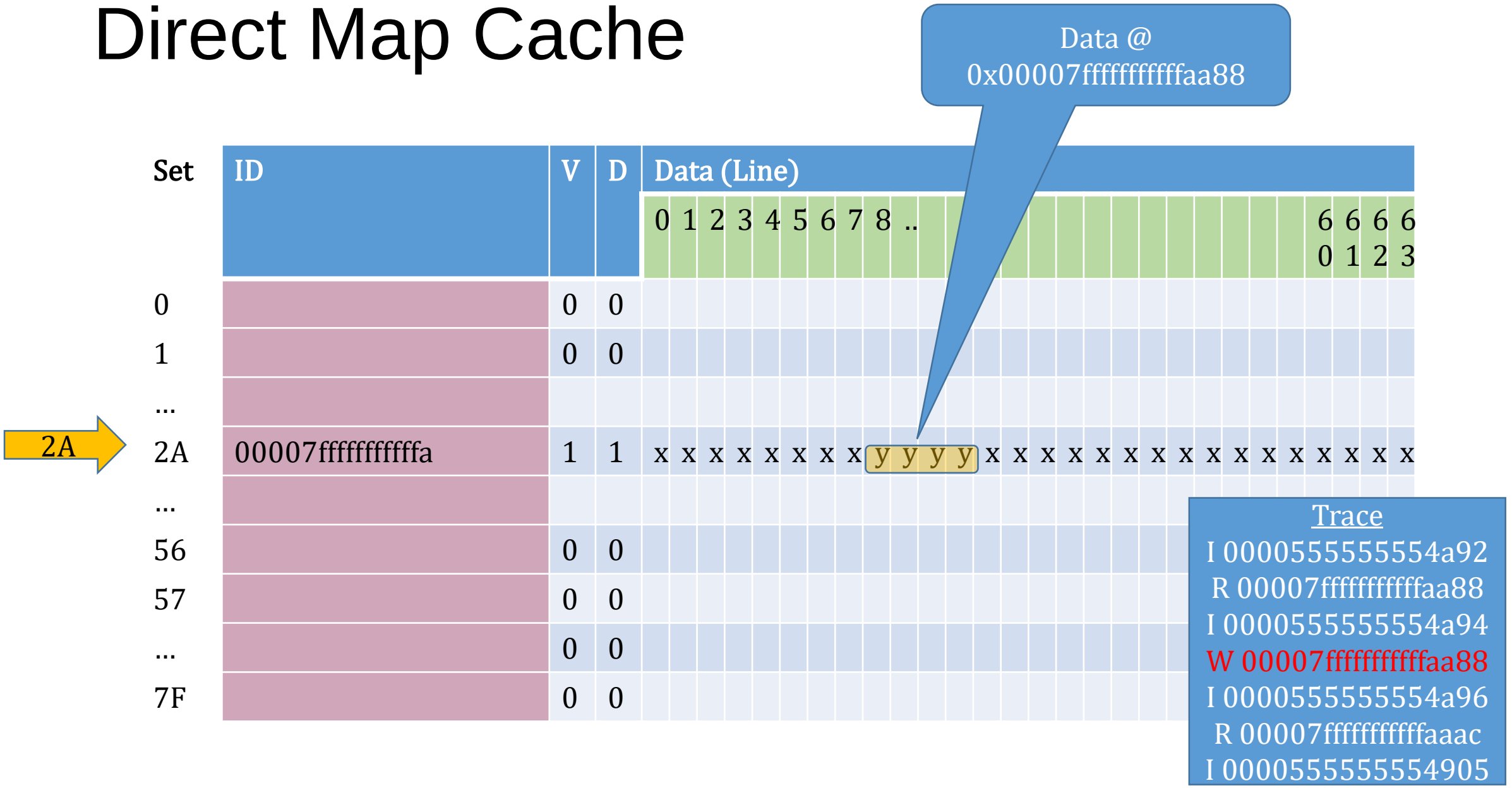
Direct Map Cache



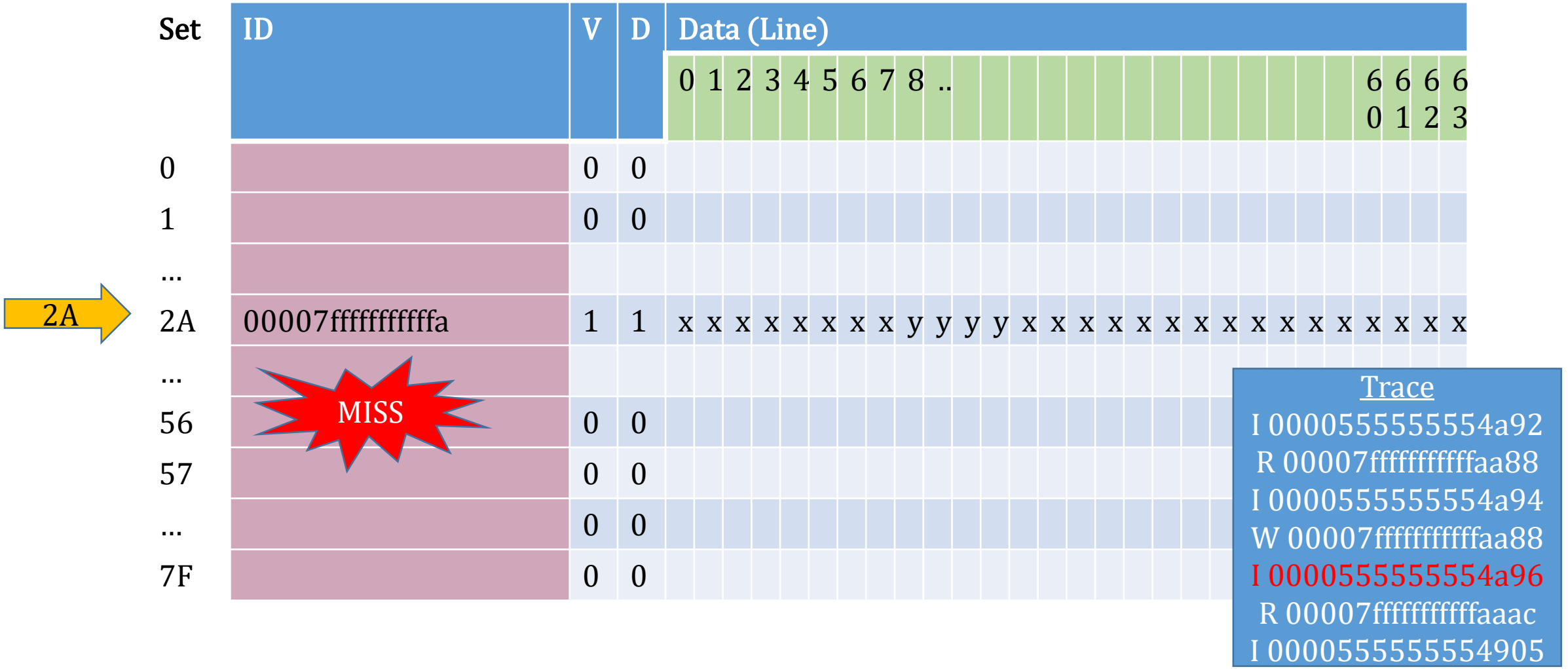
Direct Map Cache



Direct Map Cache



Direct Map Cache



Direct Map Cache Pros and Cons

Advantages

- No expensive CAM memory
- Choosing victim no cost / fast
- No extra data for victim choice
- Speeds up cache miss

Disadvantages

- Hit rate not always high
- Therefore, can be slow

Set Associative Cache (Hybrid)

- Define cache *block size* = 2^b the amount of data transferred between cache and ROM memory – most often $64=2^6$ bytes
- Divide the cache up into 2^s *sets*
  - Each set contains several ways (e.g. 64 byte block of memory)
  - Each set identified by ID *tag* – where this way comes from
  - Each set contains flags, *valid* flag and *dirty* flag
  - For instance, an 8K 4-way set associative cache contains $32=2^5$ sets
- Divide each set up into 2^w *ways*
  - Each way contains one block (e.g. 64 bytes)
  - Each way contains an ID *tag* – where in memory does this block come from
  - Each way contains flags, *valid* flag and *dirty* flag
  - For instance, an 8K 4-way set associative cache contains $4=2^2$ ways

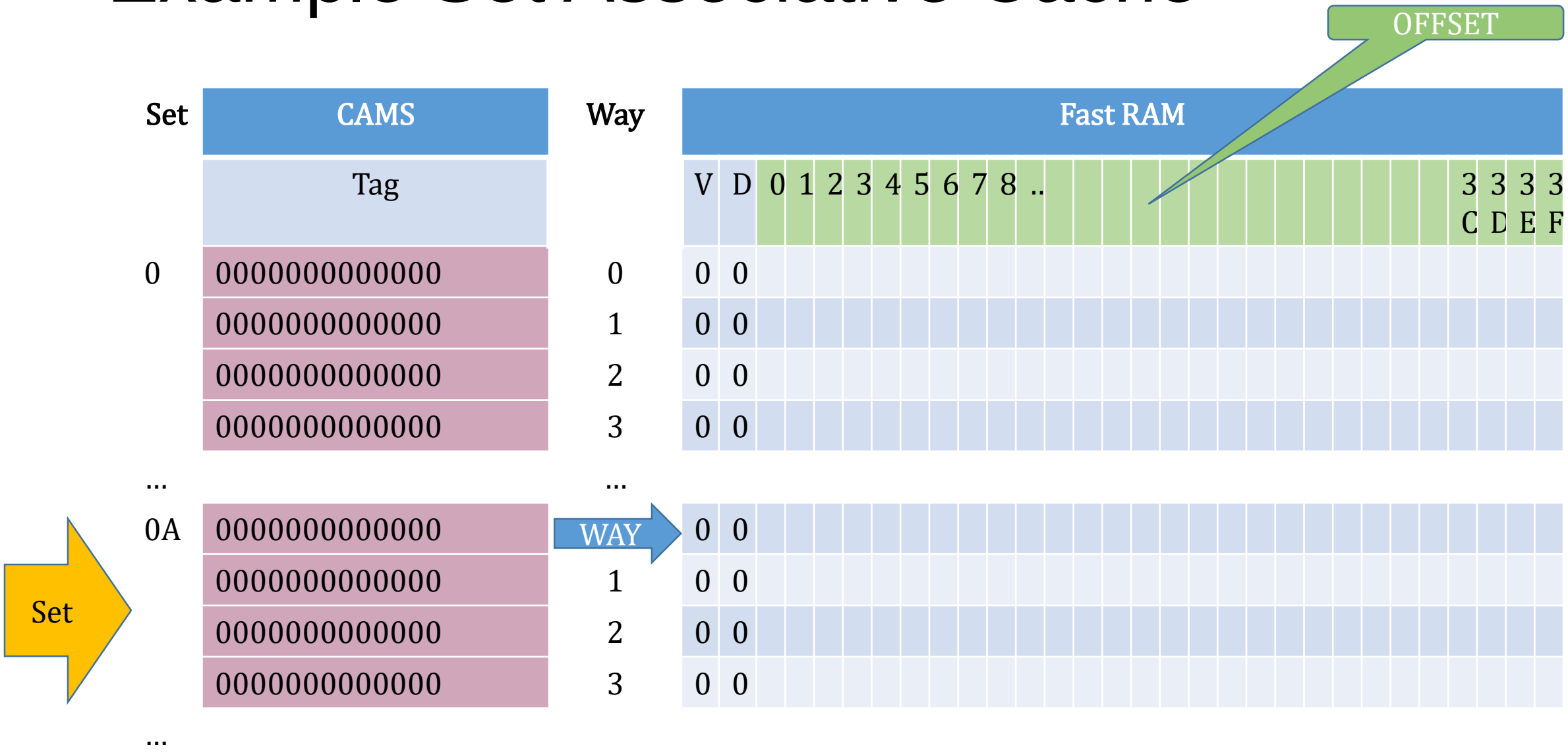
Address Sub-Fields

Trace
I 00005555555554a92
R 00007fffffffffffab48
I 00005555555554a94
W 00007fffffffffffab48
I 00005555555554a96
R 00007fffffffffffab8c
I 00005555555554905

- Divide each memory address into three sub-fields
  - Tag – High order bits - where in memory this block comes from
  - Set Index – s intermediate bits -which set this block is associated with
  - Block Offset – b Low order bits that define the offset within a block

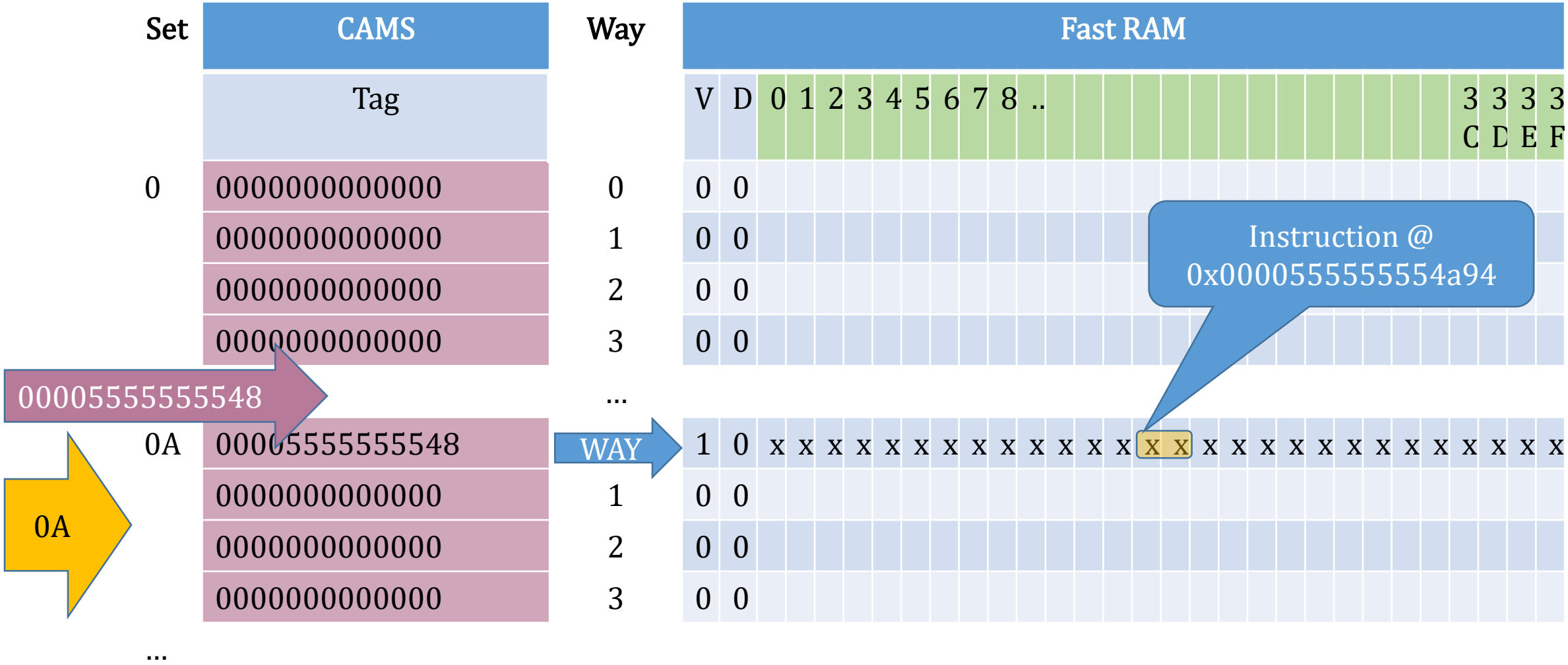
2^{63}	2^{62}	…	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
b_{63}	b_{62}	…	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_{9}	b_{8}	b_{7}	b_{6}	b_{5}	b_{4}	b_{3}	b_{2}	b_{1}	b_{0}
Tag							Set Index					Block Offset					
00005555555548[/a92]						1	0	1	0	1	0	0	1	0	0	1	0
						1	0	A				1		2			
00007fffffffffffa8[/b48]						1	0	1	1	0	1	0	0	1	0	0	0
						1	0	D				0		8			
00007fffffffffffa8[/b8c]						1	0	1	1	1	0	0	0	1	1	0	0
						1	0	E				0		C			

Example Set Associative Cache



Example Set Associative Cache

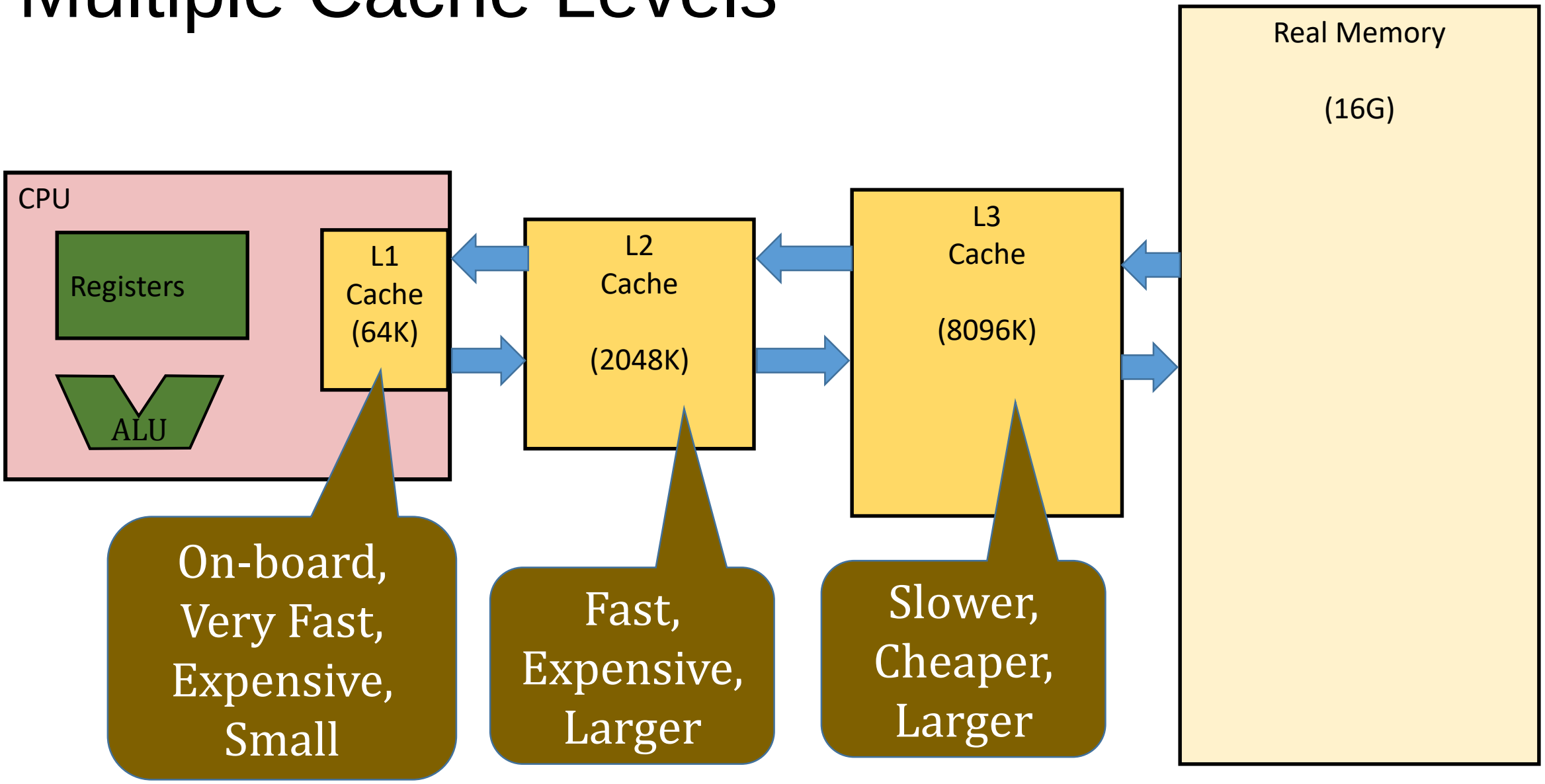
Example Set Associative Cache



On memory access request

1. Use set index to choose correct set
2. In that set, use tag w/ CAM to find correct way
3. If “valid” flag is off or tag not found, cache miss…
   - Choose way to evict from this set
   - If “dirty” flag is on, write line back to main memory based on tag and set
   - Read main memory line from new tag/set into this set/way
4. Use block offset to access data in cache
   - If memory write, turn on “dirty” flag for this set.

Multiple Cache Levels



Virtual Memory