Week8 Cache Intro

Wednesday, July 27, 2022 11:14 PM

0. Cache introduction

0.1. Intuitions: Take advantage of locality

- **Temporal Locality**
  - If a memory location is **referenced** then it **will tend to be referenced again soo**
  - => Then we **keep** the most **recently accessed** data items **closer to the processo**
- **Spatial Locality**
  - If a memory location is referenced, the **locations with nearby addresses** will te
  - => Move blocks consisting of **contiguous words** closer to the processor

Therefore, we tend to do the following

- **Temporal**: The data we **access is saved** in the cache for **potential future use**
- **Spatial**: We bring in a **chunk of data at a tim**e because there is a good chance that **w**

Consider the following **load word** instruction

- Instruction: ***lw t0, 0(t1)***
  - Say ***t1*** contains **0x12F0**, **Memory[0x12F0]** = **99**

0.2. Comparing memory access with / without cache

Without cache

1. Processor issues address 0x12F0 to memory
2. Memory reads word at address 0x12F0, *(0x12F0) = 99
3. Memory sends 99 to processor
4. Processor loads 99 into register t0

With cache

1. Cache checks to see if has copy of data at address 0x12F0
   a. If finds a match (Hit): cache reads 99, sends to processor
   b. If not (Miss): cache sends address 0x12F0 to memory
      i. Memory reads 99 at address 0x12F0
      ii. Memory sends 99 to cache
      iii. Cache replaces word which can store at address 0x12F0 with 99
      iv. Sends 99 to processor

n
or

nd to be **referenced soon**

e will want to access other data within the chunk

2. Processor loads 99 into register t0

2. Fully associative cache

0. Terminologies

- **Cache line/block:** A **single entry** in the cache
- **Line size / block size:** The **number of byte**s in **each cache line**
- **Tag: Identifies** the data stored at a given cache line
- **Valid bit:** Tells if data stored in a given cache line **is valid**
- **Capacity:** The **total number** of **data bytes** that can be stored in a cache

1. Address partition for fully associative cache

Only two fields in address: **Tag** for searching **word**, and **byte offset** for **byte**

- **# of byte offset bits = log(linesize)**
- **# tag bits = # address bits - # offset bits**

2. Eviction Policies

When cache is full, the least recently used entry will be evicted, and be overwritten

Least-Recently Used (LRU)

- Hardware keeps track of **access history**
- Replace the entry that has not been used for the longest time

3. Handling Stores (Write policy)

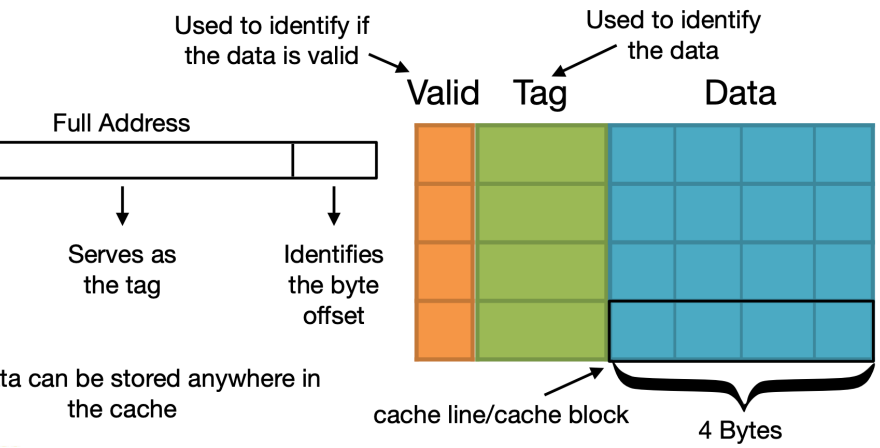
- Store instructions will write to memory and change values
- We must make sure cache and memory have consistent information

Write-through / Write-back policies

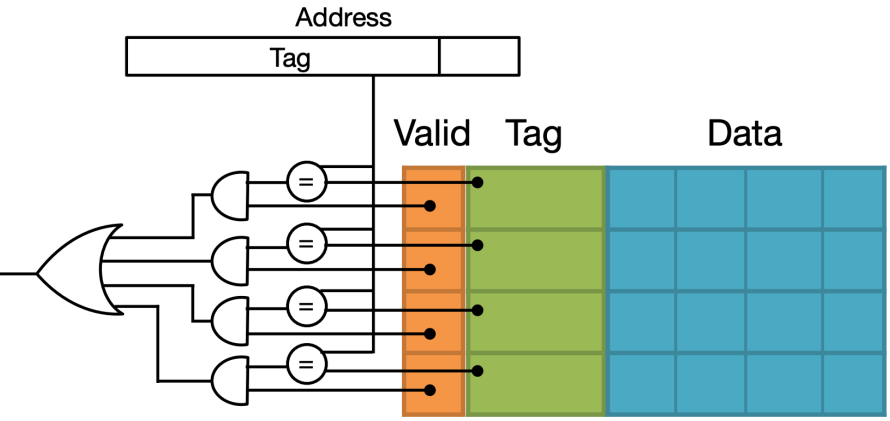
- **Write-through**
  - **Write to the cache and the memory at the same time**
    - Pros: Simple to implement
    - Cons: Write to memory will take longer, and increase the traffic to the m
- **Write-back**
  - **Write data in caches, and set the dirty bit to 1**
  - **When this line gets evicted from the cache, write it to memory**
    - Pros: Lower traffic to memory, because you may write to something mult
    - Cons: Additional control logics

Step by step demo: Assuming 12-bit width address

1. Load word at 0x5E0



with new information



emory

tiple times before you evict it from the cache

0. Initial Cache state

Address partitioning

0x5E0 = 12'b0101 1110 0000:

where tag = 10'b01 0111 1000 = 0x178, byte offset = 2'b00, loading the **entire cache**

Check hit/miss

Tag **Hits** 0x178, Load the word in **entry 0x178 (LRU is also updated)**

2. Load word at 0x524

Address partitioning

0x524 = 12'b0101 0010 0100:

Where tag = 10'b01 0100 1001 = 0x149, byte offset = 2'b00, loading the **entire cache**

Check hit/miss

Tag misses, we don't have that data. Place the loaded data into cache line with tag=

3. Load byte at 0x972 (Take **LRUs** into consideration)

Address partitioning

0x972 = 12'b1001 0111 0010:

Where tag = 10'b10 0101 1100 =0x25C, byte offset = 2'b10 = 0x2, load byte = 2'b10

Check hit/miss

- Tag misses, and cache is full. Evict the LRU cache line with LRU=3
- Then place tag = 0x25C at the cache line where LRU was 3

Update LRUs

Update LRU fields:

- LRU of newly placed content is set to 0
- Update all of their LRUs accordingly

*Then we are going to store byte at 0x524, take **write policies** into consideration*

4. Store byte at 0x524

Address partitioning

0x524 = 12'b0101 0010 0100:

Where tag = 10'b01 0100 1001 = 0x149, byte offset = 2'b00, loading the **entire cache**

Check hit/miss

Tag hit, we have the data in cache, but we will write this piece of cache

Update LRUs

Cache line with tag = 0x149 is updated and be the MRU, set it to 0

Assuming Write-back policy in this cache

LRUs

2	1	0x10F	…	…	…	…
1	1	0x178	…	…	…	…
0	1	0x209	…	…	…	…
	0	…	…	…	…	…

access 0x178

1. After load word at 0x5E0

2	1	0x10F	…	…	…	…
0	1	0x178	…	…	…	…
1	1	0x209	…	…	…	…
	0	…	…	…	…	…

2. After load word at 0x524

3	1	0x10F	…	…	…	…
1	1	0x178	…	…	…	…
2	1	0x209	…	…	…	…
0	1	0x149	…	…	…	…

→ add all other LRUs

← access 0x149, set LRU=0

Cache is full, let's consider eviction policy in next a few steps

1	3	0x10F	…	…	…	…
1	1	0x178	…	…	…	…
1	2	0x209	…	…	…	…
1	0	0x149	…	…	…	…

LRU = 3 - evicted. and all other LRU add by 1

3. After load byte at 0x972

1	0	0x25C	…	…	…	…
1	2	0x178	…	…	…	…
1	3	0x209	…	…	…	…
1	1	0x149	…	…	…	…

access 0x149

4. After storing byte at 0x524

1	0	1	0x25C	…	…	…	…
1	0	2	0x178	…	…	…	…
1	0	3	0x209	…	…	…	…
1	1	0	0x149	…	…	…	…

→ LRU +1

set LRU=0

Assuming write-back policy in this cache

Set **dirty bit** to **1**, indicating we have **write the cache**, but **not write the memory** yet

Conclusion for Fully associative cache

Data can be stored anywhere in the cache, Search/place data only using the **tag field in add**

Pros:

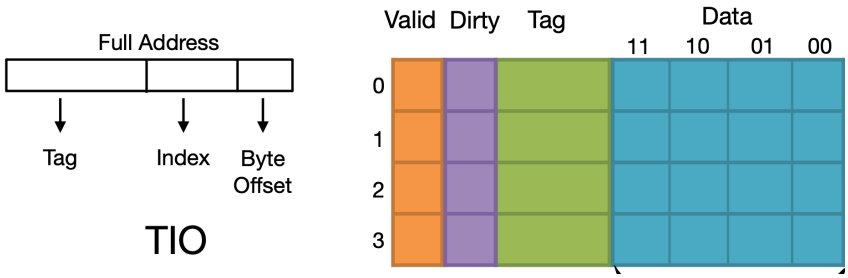
- **Flexibility of placing memory block in any cache line, and fully utilize the cache**
- **Full utilization offers better hit rate**
- Flexibility of various replacement algorithms

Cons:

- **Complexity make it slow, large, and power-consuming**
- **As each entry requires a comparator to check the tag**

3. Direct Mapped Caches

The data can **ONLY be stored in one location**: which is specified by the **index field** in **addre**



1. Address partitioning for Direct mapped cache

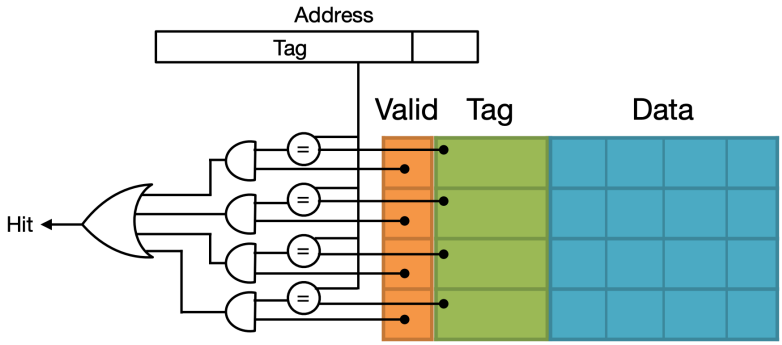
- **# byte offset bits = log2(line size)** *=> for 4 byte word*: **# offset bits = log2(4) = 2**
- **# index bits = log2(# of cache lines) =>** *for this example, a 4 entry cache*: **#index bits =**
- **# tag bits = # addr bits - # index bits - # offset bits**

- Note that byte offset bits can be larger, assume **line size** is **4-words, 16 bytes**, then **#**
  - In this case, as a **word is still 4 byte**, only **2 bits needed for byte offset**, then th

Example for direct mapped cache:

LHS: 1 word blocks, 4KB data RHS: 16 Byte (4 word)

dress



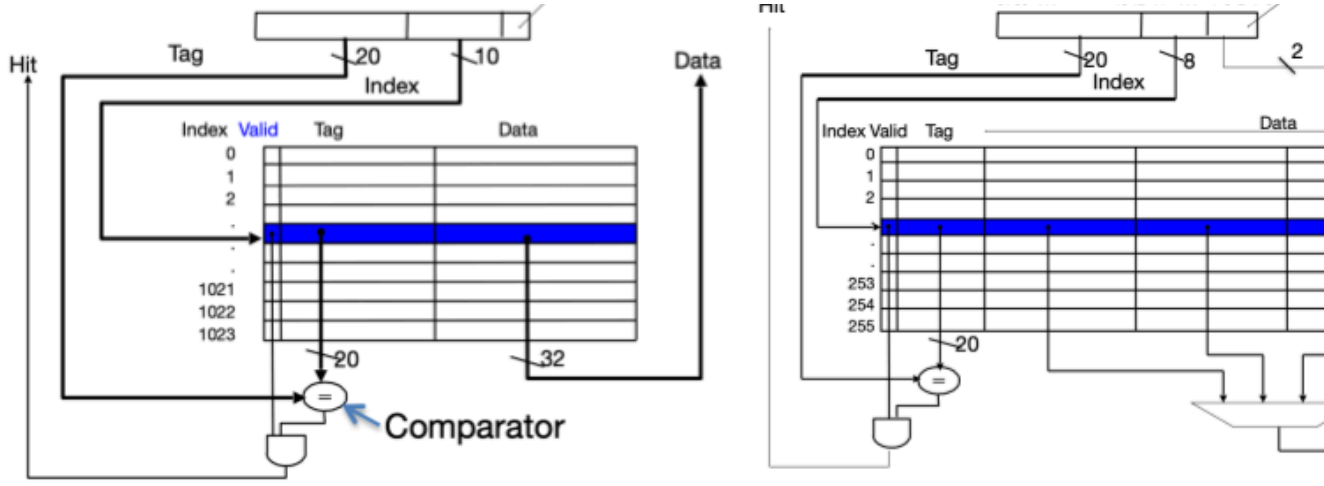
ss

= log2(4) = 2

offset bits = log2(16) = 4
e upper 2 bits is actually the word offset

block size, 4KB data

offset



2. Step by step demo : Direct mapped cache

1. Load byte at 0xFE2

   Address partitioning

   0xFE2 = 12'b1111 1110 0010:

   where tag = 8'b1111 1110 = 0xFE; index = 2'b00 ; byte offset = 2'b10

   Check hit/miss ?

   Of course we get a miss here, bring the entire line into entry with index=2'b00

2. Load byte at 0xFE8 and then load 0xFE9

   a. Load 0xFE8

   Address partitioning

   0xFE2 = 12'b1111 1110 1000:

   where tag = 8'b1111 1110 = 0xFE; index = 2'b10 ; byte offset = 2'b10

   Check hit/miss ?

   Of course we get a miss here, bring the entire line into entry with index=2'b10

   b. Load 0xFE9

   Address partitioning

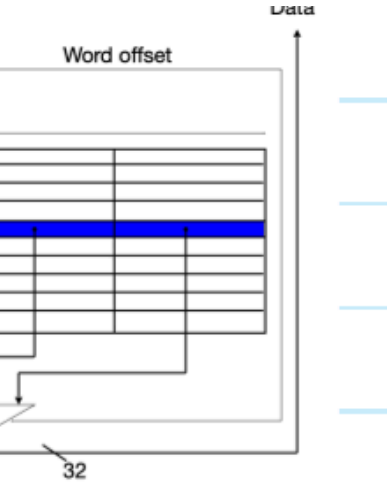
   0xFE2 = 12'b1111 1110 1001:

   where tag = 8'b1111 1110 = 0xFE; index = 2'b10 ; byte offset = 2'b11

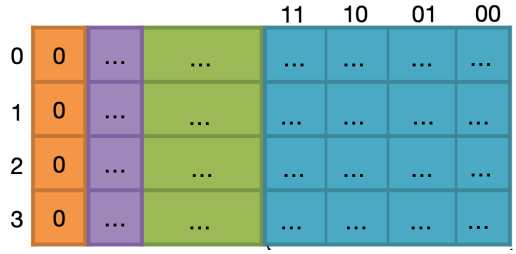
   Check hit/miss ?

   We have load contents into index 2'b10, and tag matches, we get a hit. Then load byt

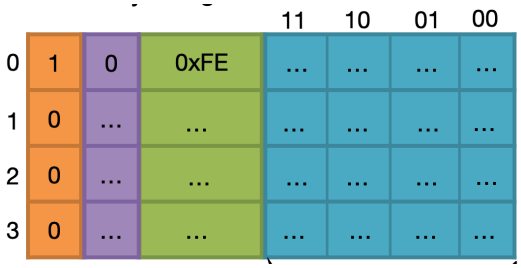
Some cache line might be replaced then



0. Initial State



1. After Load 0xFE2



2. After load 0xFE8 and Trying to load 0xFE9

Here, we need to consider Direct mapped cache replacement

e from byte=2'b11

3. After Load byte at 0xDF9

3. Load byte at 0xDF9

Address partitioning

0xDF9 = 12'b1101 1111 1001:
where tag = 8'b1111 1110 = 0xDF; index = 2'b10 ; byte offset = 2'b01

Check hit/miss ?

Valid at entry 2'b10 but **Tag doesn't match, it's a miss**. The entire cache line will be e
Next the word from 0xDF8 will be placed in entry with index =2'b10

4. Load byte at 0xFE8

Address partitioning

0xDF9 = 12'b1111 1110 1000:
where tag = 8'b1111 1110 = 0xFE; index = 2'b10 ; byte offset = 2'b00

Check hit/miss ?

Valid at entry 2'b10 but **Tag doesn't match, it's still a miss**. The entire cache line will
Next the word from 0xFE8 will be placed in entry with index =2'b10

Conclusion for Direct Mapped Cache

Different from fully associate cache, data can **only be stored at the certain entry** (Cache lin
As the address partitioning has: **Tag, index, and offsets**. **Index** will **constrain where to store**

- Pros: Cheap and fast from the perspective of circuit design
- Cons:
  - More likely to have cache conflict's, and higher miss rates
  - No flexibility of replacement algorithm

4. Set-Associative Caches

A combination of two types of caches. In fact, we get multiple "associative" caches, each is

- **The data can only be stored at one index, but there are multiple slots to store it in**

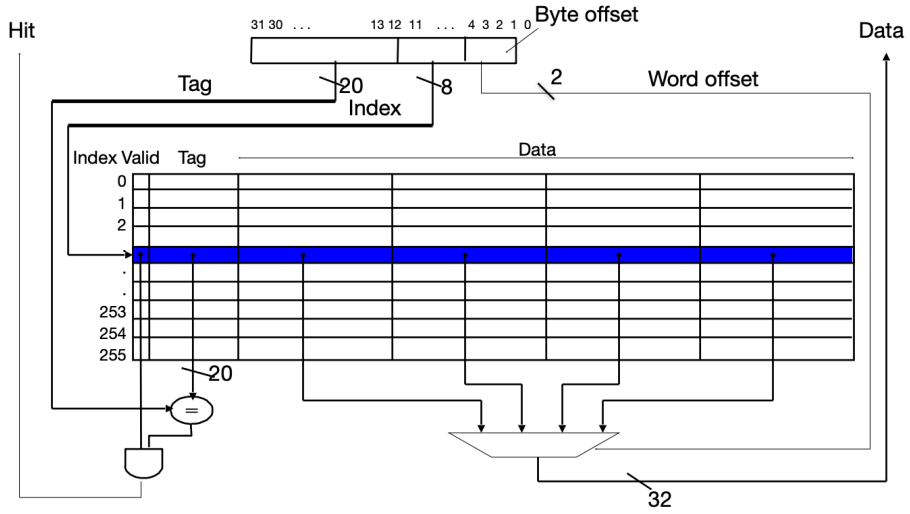
0	1	0	0xFE	…	…	…	…
1	0	…	…	…	…	…	…
2	1	0	0xDF	…	…	…	…
3	0	…	…	…	…	…	…

victed,

4. After load byte at 0xFE8

				11	10	01	00
0	1	0	0xFE	…	…	…	…
1	0	…	…	…	…	…	…
2	1	0	0xFE	…	…	…	…
3	0	…	…	…	…	…	…

be evicted,

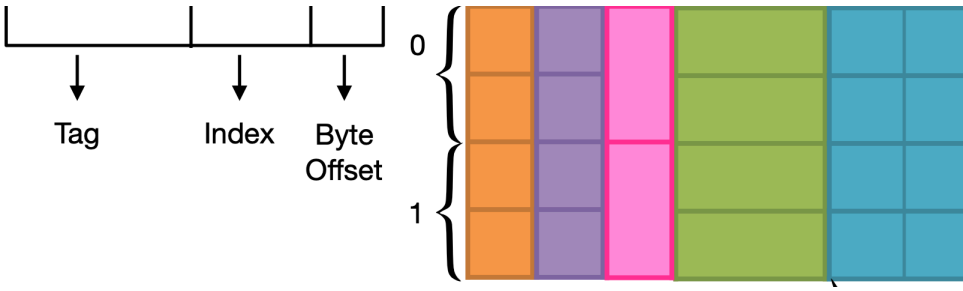


e)
e the data

labeled with a certain index

Data

		01	00



1. Address partitioning for Set-Associative Caches

Partition is similar to direct mapped cache**, the ONLY difference is the index field. Index is**

- **# byte offset bits = log2(line size)** *=> for line size that just fits a 4 byte word*: **# offset**
- **# index bits = log2(# of cache lines) =>** *for this example, a 2 set set-associative cache*:
- **# tag bits = # addr bits - # index bits - # offset bits => In this example, # tag bits = 12**

2. Step by step demo : Direct mapped cache

1. Load byte at 0x8E2

   Address partitioning

   0xFE2 = 12'b1000 1110 0010:

   where tag = 9'b1 0001 1100 = 0x11C; index = 1'b0 ; byte offset = 2'b10

   Check hit/miss ?

   Of course we get a miss here, bring the entire line into the first empty entry with inde

2. Load byte at 0x8E8 and then load 0x8E9

   a. Load 0x8E8

   Address partitioning

   0xFE2 = 12'b1000 1110 1000:

   where tag = 9'b1 0001 1101 = 0x11D; index = 1'b0 ; byte offset = 2'b00

   Check hit/miss ?

   Of course we get a miss here, bring the entire line into the second entry with index 1'

   b. Load 0x8E9

   Address partitioning

   0xFE2 = 12'b1000 1110 1001:

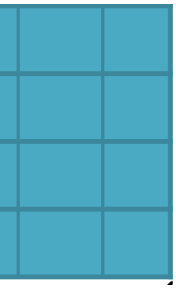
   where tag = 9'b1 0001 1101 = 0x11D; index = 1'b0 ; byte offset = 2'b01

   Check hit/miss ?

   We have load contents into index 1'b0, and tag matches, we get a hit. Then load byte

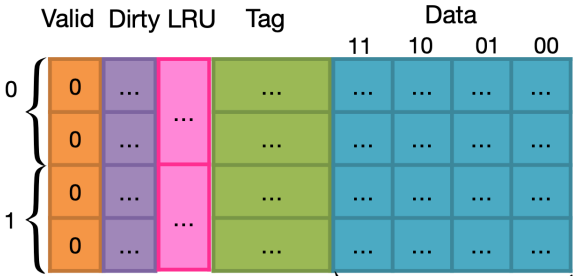
   Some cache line might be replaced then

3. Load byte at 0xDF7



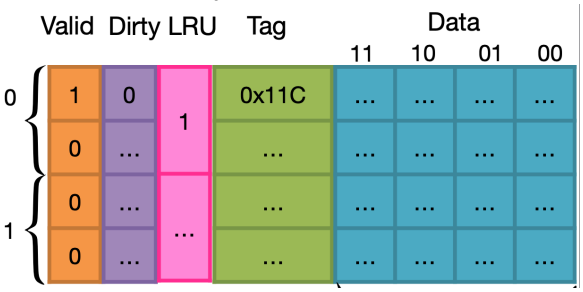
an index of sets
bits = log2(4) = 2
: # index bits = log2(2) = 1
- 2 - 1 = 9

0. Initial State



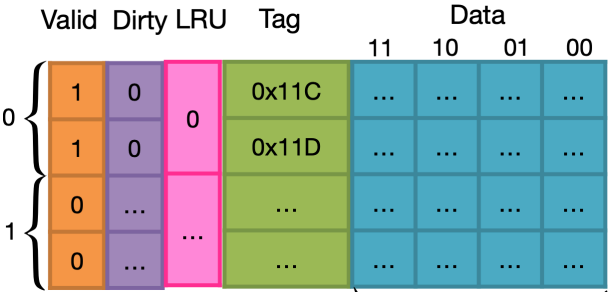
ex=1'b0

1. Read Byte 0x8E2



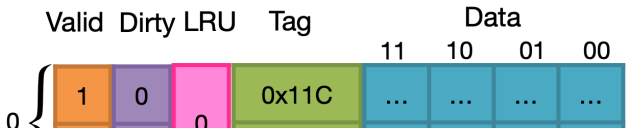
'b0

2. Read Byte 0x8E8 and then read 0x8E9



from byte=2'b01

3. Read Byte 0xDF7



Address partitioning

0xDF9 = 12'b1101 1111 0111:

where tag = 9'b1 1011 1110 = 0x1BE; index = 1'b1 ; byte offset = 2'b01

Check hit/miss ?

Valid is 0, it's a miss. Bring the entire line into the first empty entry with index=1'b1

4. Load byte at 0xAB8

Address partitioning

0xAB9 = 12'b1010 1011 1001:

where tag = 9'b1 0101 0111 = 0x157; index = 1'b0 ; byte offset = 2'b01

Check hit/miss ?

Valid at index 1'b0 but tag doesn't match, it's a miss. Evict the cache line (oldest), in t

Load from 0xAB8, and put it in the cache line

Set-Associative Cache Summary

Basically somewhere in between direct-mapped and fully associate

- Instead of constrain location of data in cache to **a certain cache line**. The **index navig**
- For a **4-way set-associative** cache, that cache **allows storing 4 data with the same in**

In addition to being a trade-off between direct-mapped and fully associative cache, this pla

- **Pros: Allows flexibility of placement policies**
- **Cons: Still suffers from conflict misses**

5. Comparisons between Cache placement policies

0. Terminology Recap

0. Valid

- When start a new program, **caches doesn't have valid information for this program**
- **Valid bit** is the indicator to tell if each entry is valid for this program. **1 for valid, and**

A. Tag

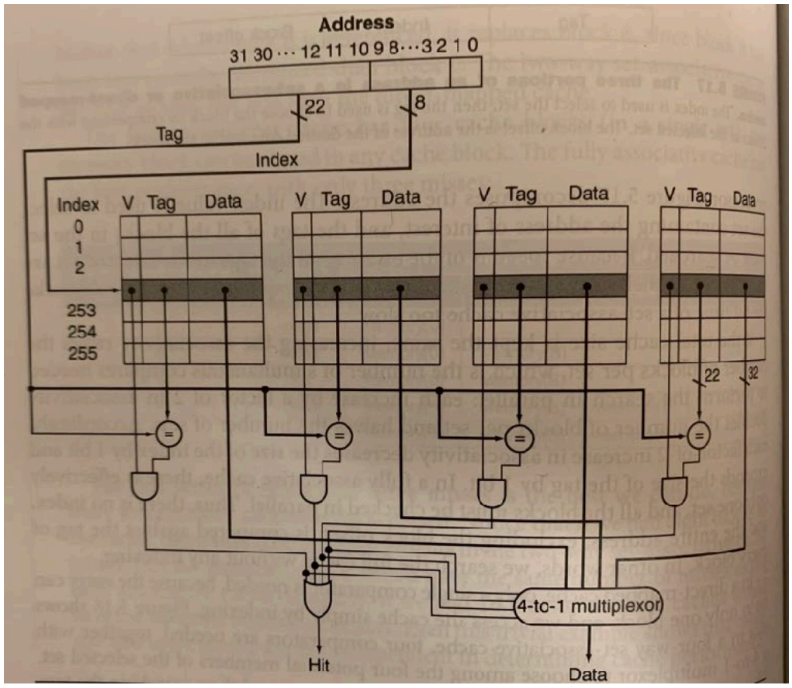
- Every line the cache has a tag which helps **identify if the memory address that we ar**
- **To check if our address matches a given line, we need to verify: (assume the index**

	Valid	Dirty	LRU	Tag	Data 11	10	01	00
	1	0		0x11D	…	…	…	…
1	1	0	1	0x1BE	…	…	…	…
	0	…		…	…	…	…	…

4. Read Byte 0xAB8

	Valid	Dirty	LRU	Tag	Data 11	10	01	00
0	1	0	1	0x157	…	…	…	…
	1	0		0x11D	…	…	…	…
1	1	0	1	0x1BE	…	…	…	…
	0	…		…	…	…	…	…

his case, it's entry0 in index0



ates to several sets.
dex
cement policy also features

0 for non-valid

e trying to access is stored in cache
is good)

- Valid bit is 1
- Tag in cache is equal to tag in address

B. Write through vs. Write-back

- **Write-through**
  - **Write to the cache and the memory at the same time**
    - Pros: Simple to implement
    - Cons: Write to memory will take longer, and increase the traffic to the m
- **Write-back**
  - **Write data in caches, and set the dirty bit to 1 (additional data needed)**
  - **When this line gets evicted from the cache, write it to memory**
    - Pros: Lower traffic to memory, because you may write to something mult
    - Cons: Additional control logics

C. Write allocate vs No write-allocate

- **Write-allocate**
  - **On a write miss,** you bring the **line into the cache**, and then **update the line**
- **No write-allocate**
  - On a write miss, **do not** bring the line into the cach**e, you only update memory**

1. Comparison

	Fully Associative	**Direct Mapped**
Possible location of data in cache	A specific line of data can be stored in any line of the cache	A specific line of data can only be sto in one index of the cache
Address Partitioning	• **Tag** • **Byte (word offset)**	• **Tag** • **Index: for the ONLY slot** • **Byte/word offset**
Replacement policy	Flexible, user need to choose one accordingly	If the line you want to store the data already occupied, then you evict that and load new data in. No option for replacement policy

2. 3Cs for misses

- **Compulsory Miss**

emory

tiple times before you evict it from the cache

	Set Associative
red	A specific line of data can be stored at only one index of the cache (but there should be multiple lines in each index, #ways > 1)
	• **Tag** • **Index: for possible slots (# of "ways")** • **Byte/word offset**
in is line	Flexible, user need to choose one accordingly

    - Caused by the first access to a block that has never been in the cache
- **Capacity Miss**
    - Caused when the cache cannot contain all the blocks needed during the execut
    - Occurs when blocks were in the cache, replaced, and later retrieved
- **Conflict Miss**
    - Occurs in set-associative or direct mapped caches when multiple blocks compe
    - Never appear in a fully-associative cache

tion of a program

te for the same set