Cache Memory

The Four Questions

Computer System Architecture (CS5202)
IIT Tirupati
March 2020
Jaynarayan t tudu
jtt@iittp.ac.in

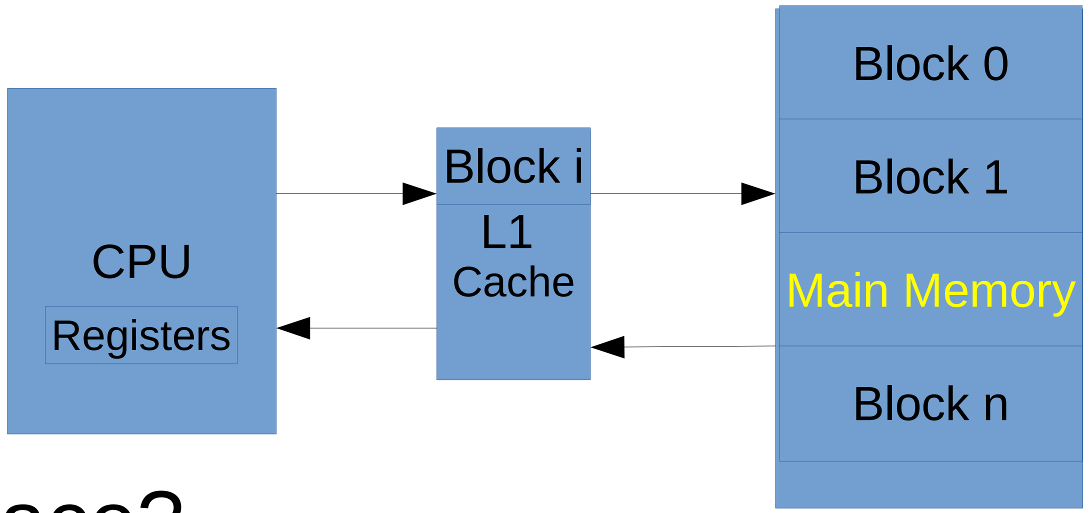
Last Lecture

- Memory Design
- Concept of Memory block
- Addressing a block
- Mapping between main and cache
- Four important questions

Four Questions

- Where to place a block in the upper level?
  - Mapping mechanism
- How to find the block already placed?
  - Block location
- How to accommodate or place a blocks on miss?
  - Block replacement
- What happen when a block is updated
  - Write policies

Cache Mapping and Placement



Where to place?

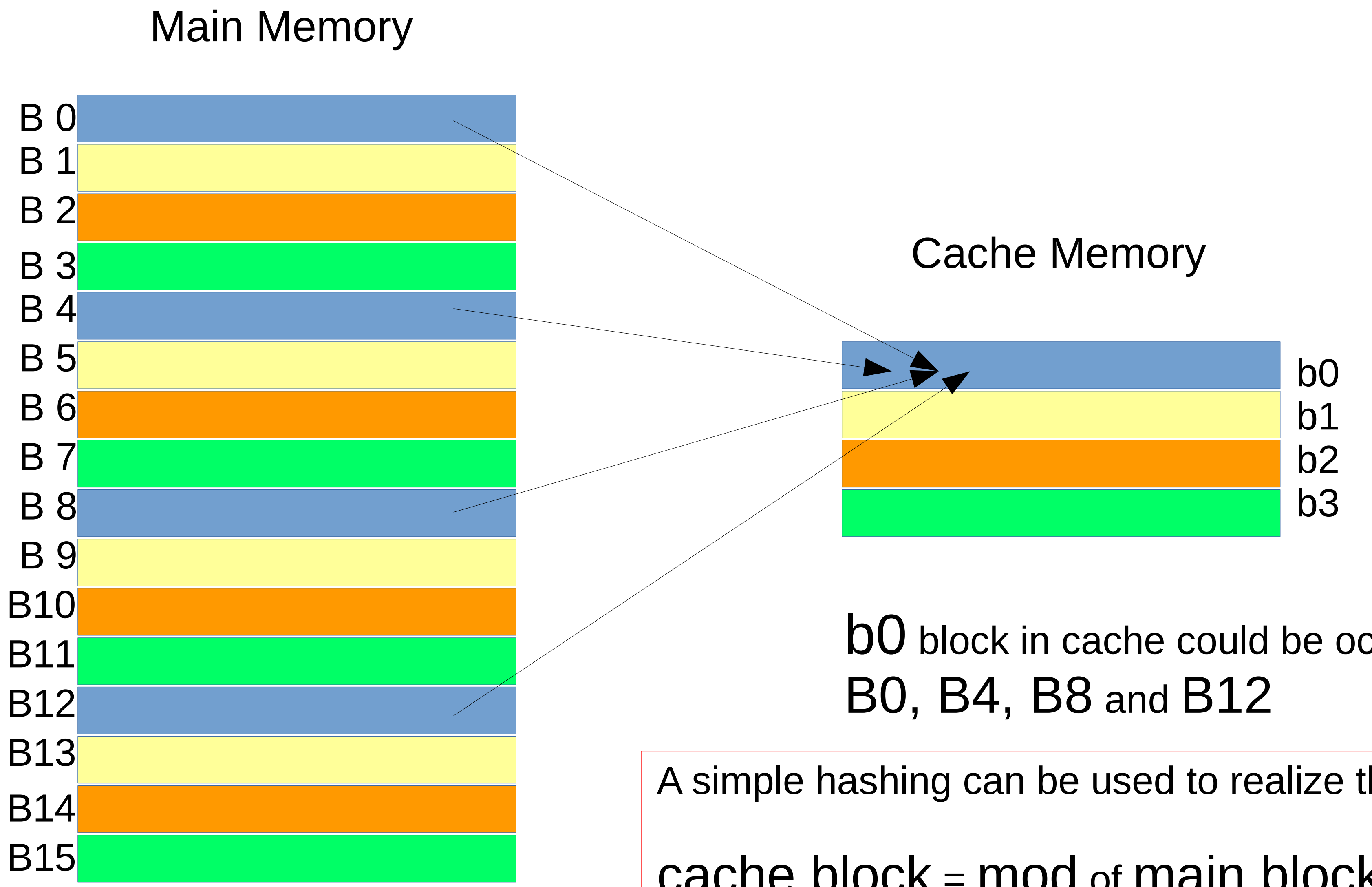
General thoughts:

Thought 1: Let any main memory bock occupy any cache block
Thought 2: Only a selected main memory block occupy a designated set of cache block
Thought 3: Let a designated set of main block occupy a fixed cache block
Thought 4: Any other possibilities? Looks like no possibility! World is changing (think ML)

Standard name:

Thought 1: Fully associative mapping
Thought 2: Set associative mapping
Thought 3: Direct mapping

Direct Mapping



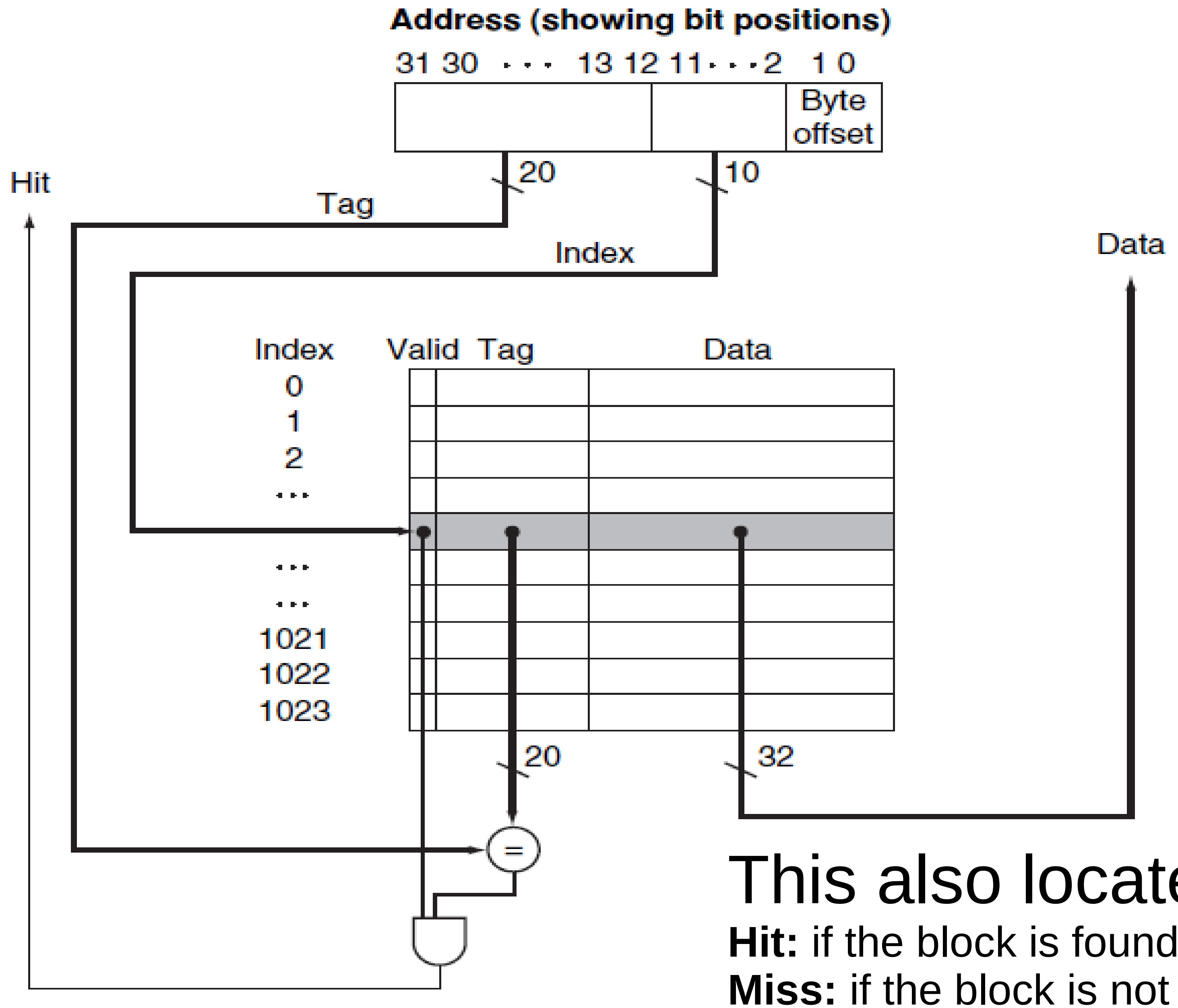
b0 block in cache could be occupied by B0, B4, B8 and B12

A simple hashing can be used to realize this mapping!

cache block = mod of main block

- Cache memory block also called as Line or cache line

Direct Mapping Design



- The simple indexing will work as a mod

- Tag matching is needed because multiple main blocks map to single cache block

This also locates a block!

Hit: if the block is found

Miss: if the block is not found

Direct Mapping

Points to be noted:

- Looks simple to implement
- Multiple main blocks are mapped to single location

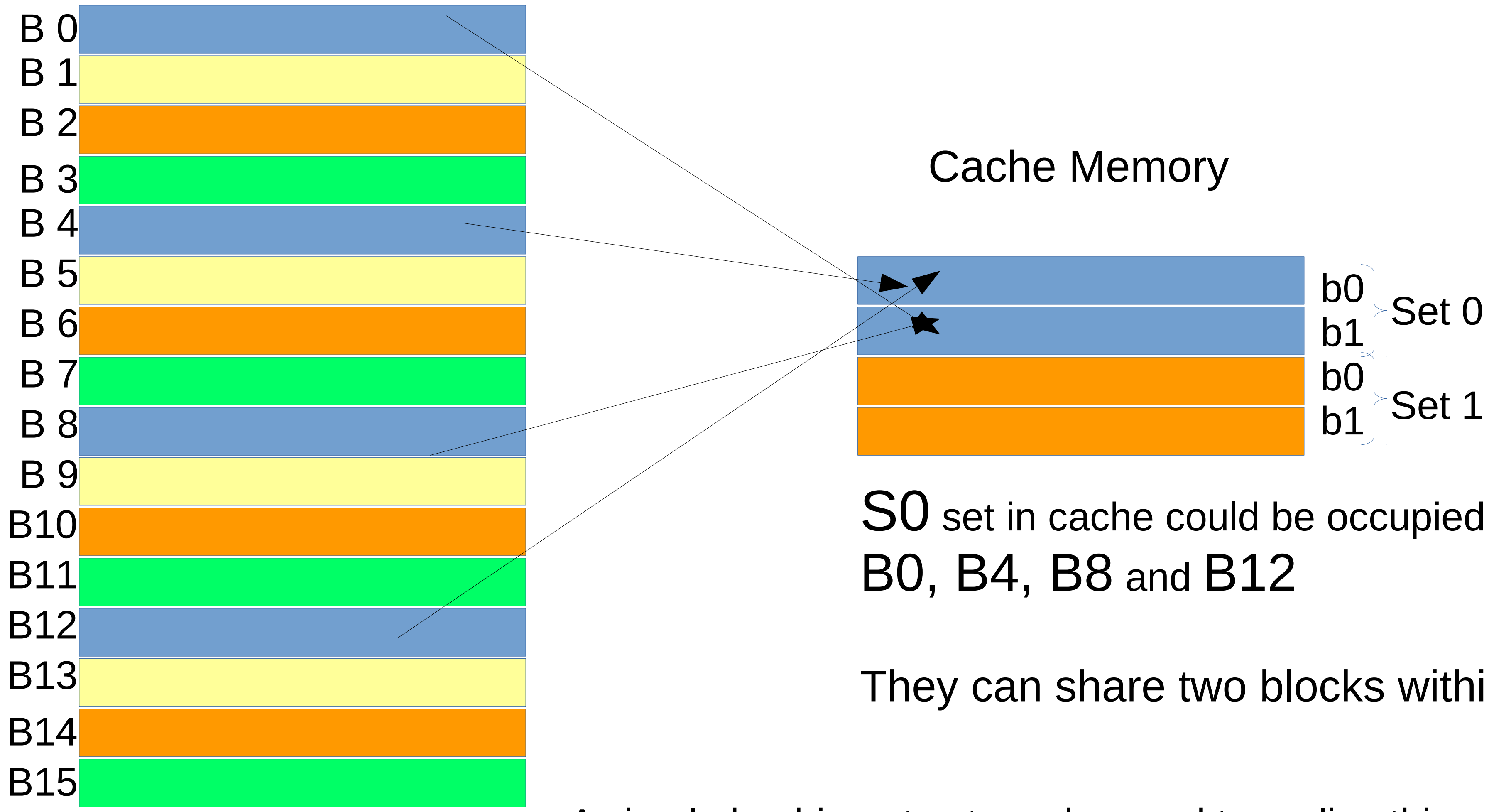
Scenario:

There are few cache blocks which are currently empty, but the program has requested only $B_{(i+jn)}$ blocks from the main, what would happen to b_i in cache? (here $j = 0$ to m, n is size of cache memory in blocks)

The b_i would witness frequent miss!

How to avoid this?

Set Mapping



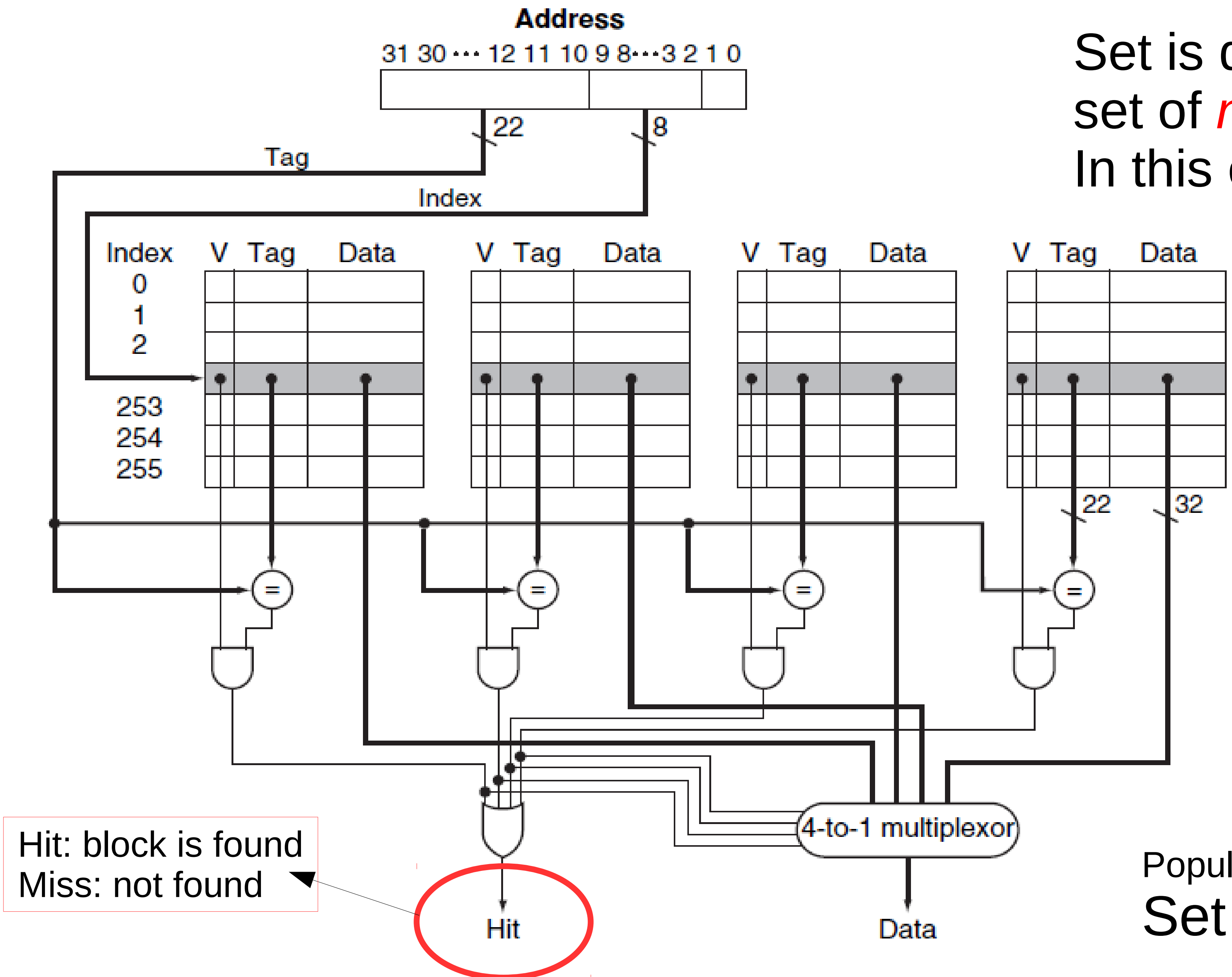
S0 set in cache could be occupied by B0, B4, B8 and B12

They can share two blocks within

A simple hashing at set can be used to realize this mapping!

cache set = mod of main block

Set Mapping

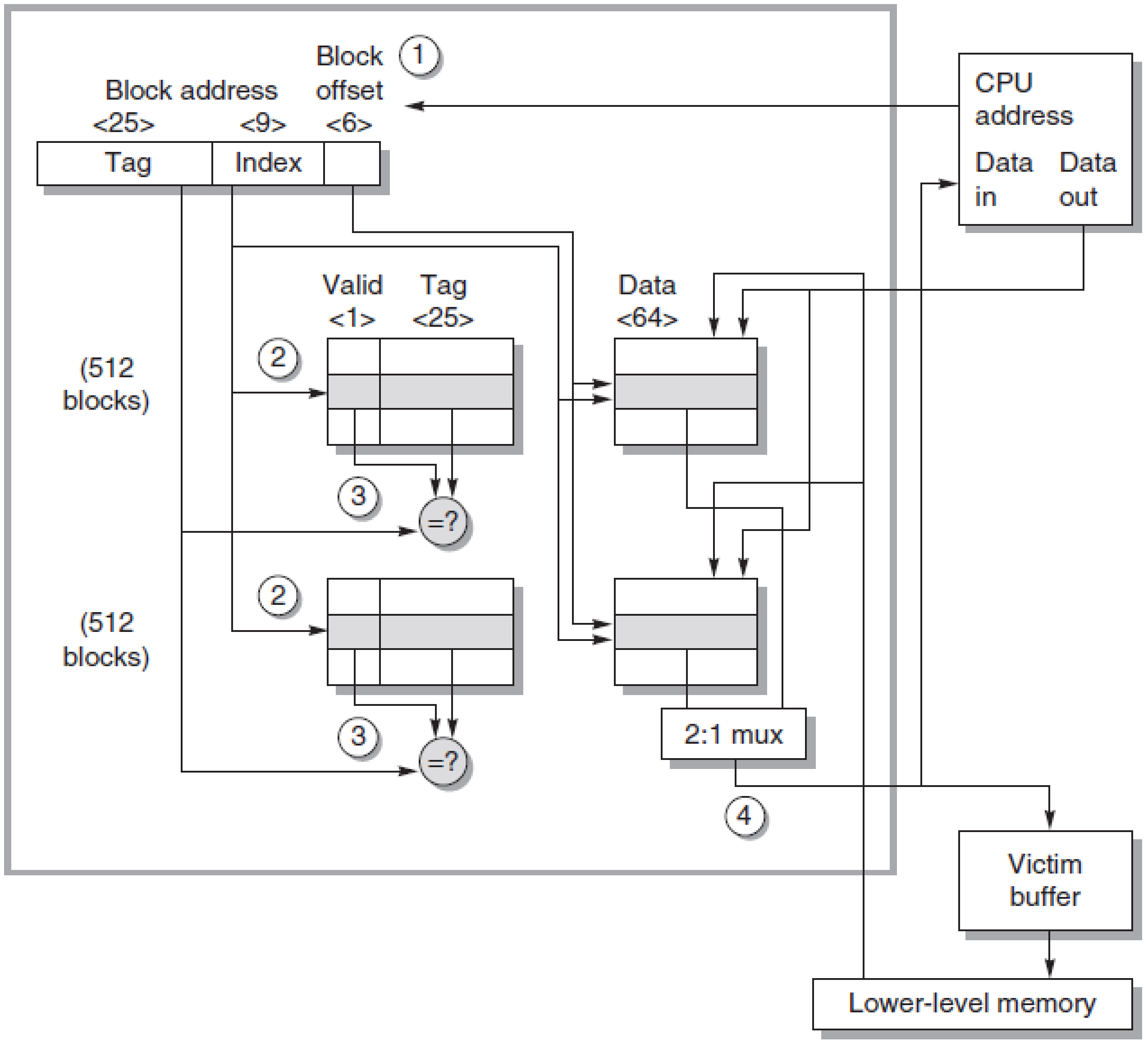


Hit: block is found
Miss: not found

Set is defined as a set of *n* blocks.
In this example *n = 4*

Popular name:
Set associative

Set Associative Mapping



Case Study:
AMD Opteron Data Cache

Cache size = 64 KB

Organisation = 2-way set
(2 blocks per Set)

Block size = 64 Byte
Offset = log (64) = 6 bits

Total set = 64 KB / 2 X 64 B
= 512

Index = log(512) = 9 bits

Tag = remaining bits = 25

Physical address size = 40 bits

Set Mapping: Points to be Noted

- Looks improved from direct mapped in terms of avoiding miss.
  Effectively managing the blocks!

- What about hardware complexity?
  - Number of comparator increased!
  - Number of TAG bits also increased!

- No free food here! You have to pay price as hardware complexity to avoid misses!

Best and Worst Scenarios:

Fully Associative Mapping

General thoughts:

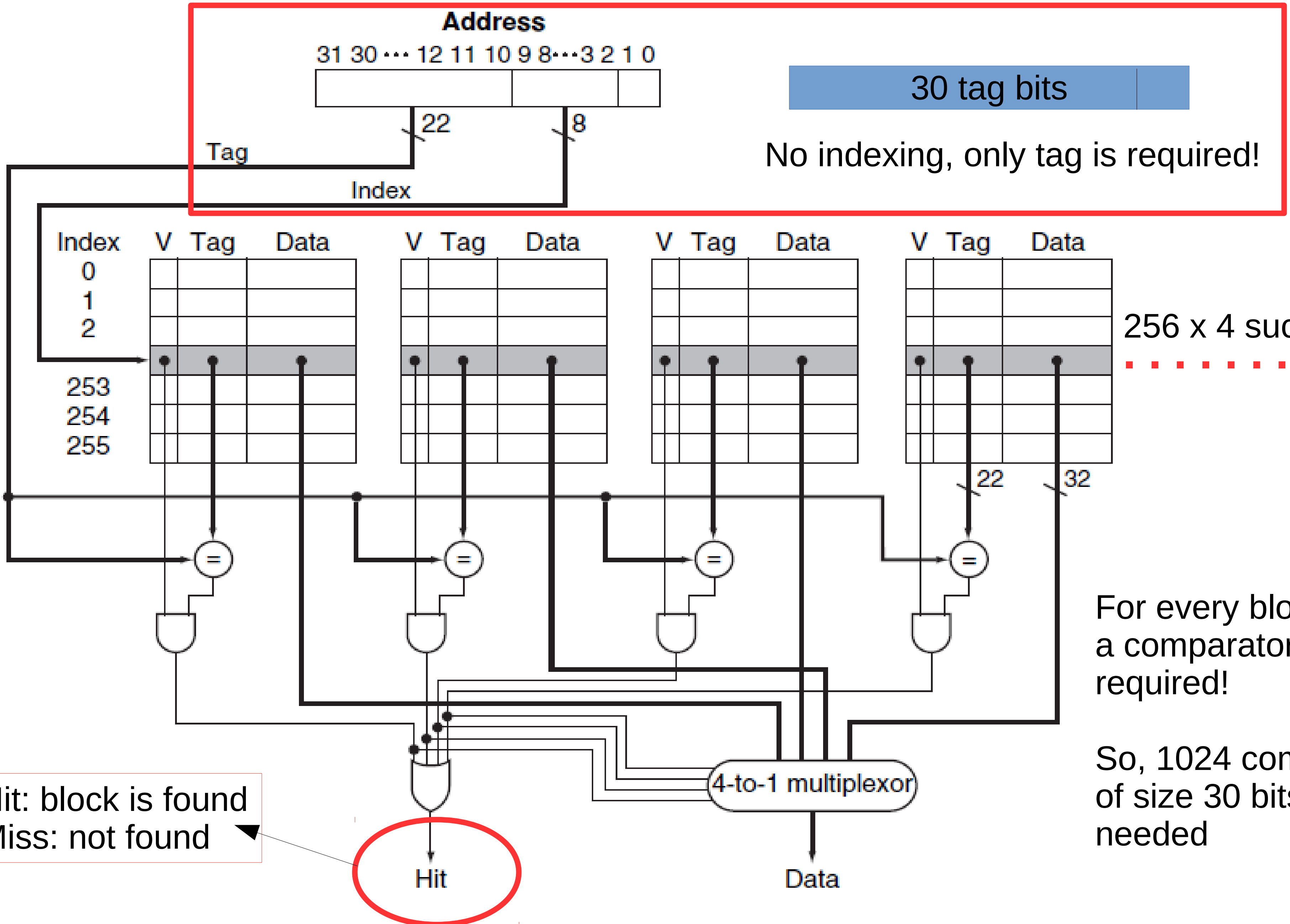
Thought 1: Let any main memory bock occupy any cache block
Thought 2: Only a selected main memory block occupy a designated set of cache block
Thought 3: Let a designated set of main block occupy a fixed cache block
Thought 4: Any other possibilities? Looks like no possibility! World is changing (think ML)

Standard name:

Thought 1: Fully associative mapping
Thought 2: Set associative mapping
Thought 3: Direct mapping

What about just having only one set?

Set Associative Design



No indexing, only tag is required!

256 x 4 such blocks

For every block a comparator is required!

So, 1024 comparators of size 30 bits are needed

Hit: block is found
Miss: not found

Food for Thought

- How do you say which mapping scheme is better?
- When one scheme would be chosen over other?
- If performance is the crucial which one would you chose?
- If hardware cost is the crucial which one would you chose?
- What kind of application domain demands performance?
- What kind of application domain demands hardware cost?
- What about power dissipation? How does it looks like for each scheme?

The Four Questions

- Where to place a block in the upper level?
  - Mapping mechanism
- How to find the block already placed?
  - Block location
- How to accommodate or place a blocks on miss?
  - Block replacement
- What happen when a block is updated?
  - Write policies

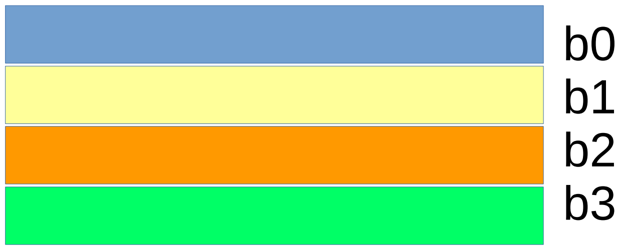
Block Replacement

- How to accommodate or place a blocks on miss?

Since the size of cache is much smaller than the main memory, not all the blocks could be accommodated. Miss is bound to occur reason being the requested word in not present in any of the cache block.

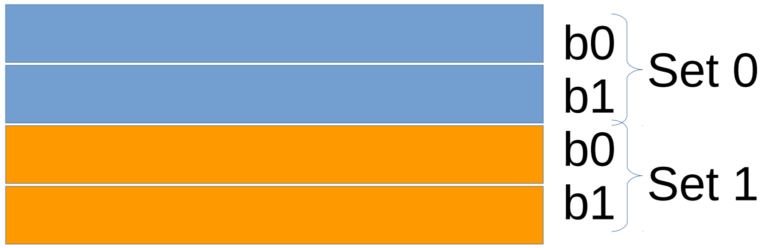
This situation needs to be analyzed with respect to placement scheme!

Direct Mapping Cache



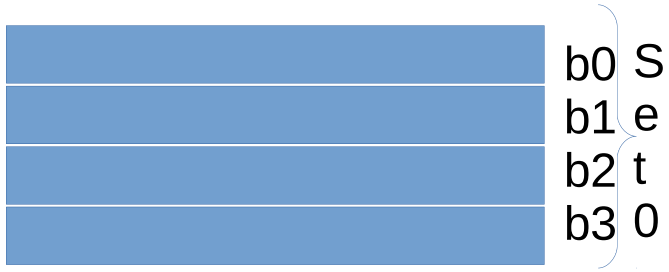
Misses are restricted to blocks

Set Associative Cache



Misses are restricted to Set

Fully Associative Cache



Misses are not restricted

Block Replacement

Replacement in Direct Map Cache:

This is trivial!

- No additional hardware resources for replacement!

Block Replacement

Set Associative and Fully Associative Cache:

How to decide on which block to replace from a set of blocks?

The best policy is: If we can know the block which is not going to be referred in the near future.

- Random policy
  - The idea is to be uniform to all the blocks.
  - Simple to implement, just have pseudo random generator.
- Least Recent Used (LRU)
  - Based on the idea of temporal locality
  - The block which has not been used for long time need to replaced.
  - Implementation requires sophisticated hardware support with timer and counter.
- First In First Out (FIFO)
  - The block which have stayed longer to be replaced
  - Hardware is simpler than LRU, just requires counter.

Only way to know the future is by understanding the past!

Experimental Results on SPEC

Two-way Associativity

Size	LRU	Random	FIFO
16 KB	114.1	117.3	115.5
64 KB	103.4	104.3	103.9
256 KB	92.2	92.1	92.5

Four-way Associativity

Size	LRU	Random	FIFO
16 KB	111.7	115.1	113.3
64 KB	102.4	102.3	103.1
256 KB	92.1	92.1	92.5

Eight-way Associativity

Size	LRU	Random	FIFO
16 KB	109.0	111.8	110.4
64 KB	99.7	100.5	100.3
256 KB	92.1	92.1	92.5

Data Cache
misses per
1000 instructions

SPEC2000 Int
SPEC200 fp

Block size = 64 byte
Architecture: Alpha

Write Policies

When should the modified block be updated in lower-level memory?

We have only two choices:

Either update as and when the cache block is updated, OR
Update later on whenever the block is replaced from cache

We have two policies accordingly:

Write-through

and

Write-back

Write Policies

Write-through:

- The update is done in both the block in the cache as well as to the block in lower-level memory.

- There is no need to keep track of update status of block.

Write-back:

- The update is done when the block is replaced.
- Update status need to be recorded. This done using 1 bit, called dirty bit (you can call update bit).
- On read miss, the block has to be written back to lower-level. (this is not needed in write-through)

Write Policies

Impact on Performance:

- Write stall: processor need to wait a write-through to complete
  - Solution: have a buffer to keep the updated block which can be written to the lower-blocks without halting processor.

- Handling Write miss: Data are not required (processor produces data)
  - Write allocate: allocate a block and then write-hit (same as read miss)
  - No-write allocate: Don’t do anything at higher-level cache, rather update directly at lower-level.

Exercises and Practice Task

- Implement the LRU, FIFO and Random replacement policy using a simulator and run the simulator on traces of your program to compare the results on misses.

- You can also implement the LRU and FIFO in Verilog and simulate check the access time, and miss rate for at least few instruction cycle.

Exercise from the Book (H&P):

Appendix B:
- B2
- B3

Reference

All the figures presented here are taken from the following text:

- Computer Architecture: Quantitative Approach, 5th Edition
- Computer Organisation and Design -HW/SW Interface, 5th Edition

Reading:

Appendix B of Quantitative Approach
Chapter 5 of HW/SW Interface