CS 223 Computer Organization & Architecture

Lecture 34 [04.05.2020]

Cache Memory-Organization and Design Issues

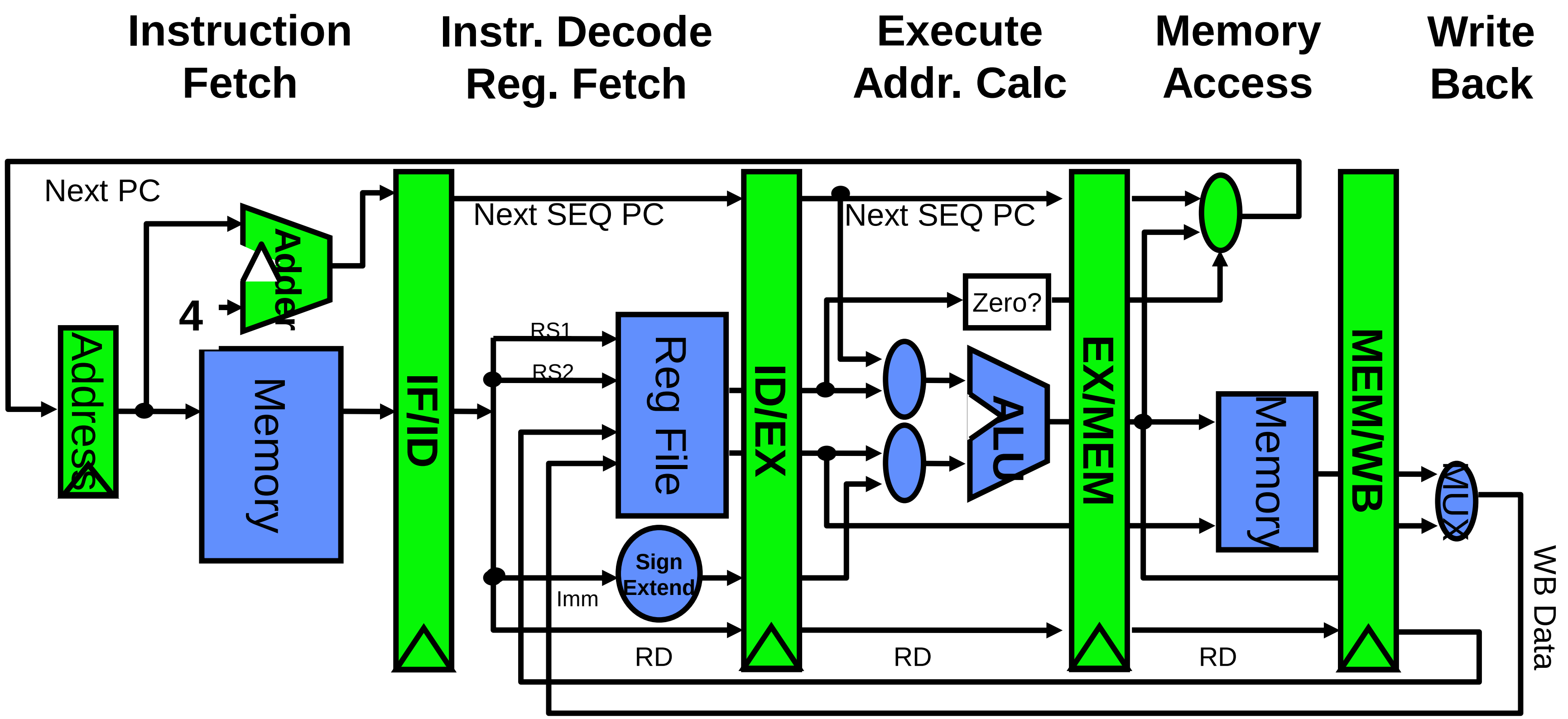
John Jose

Assistant Professor

Department of Computer Science & Engineering

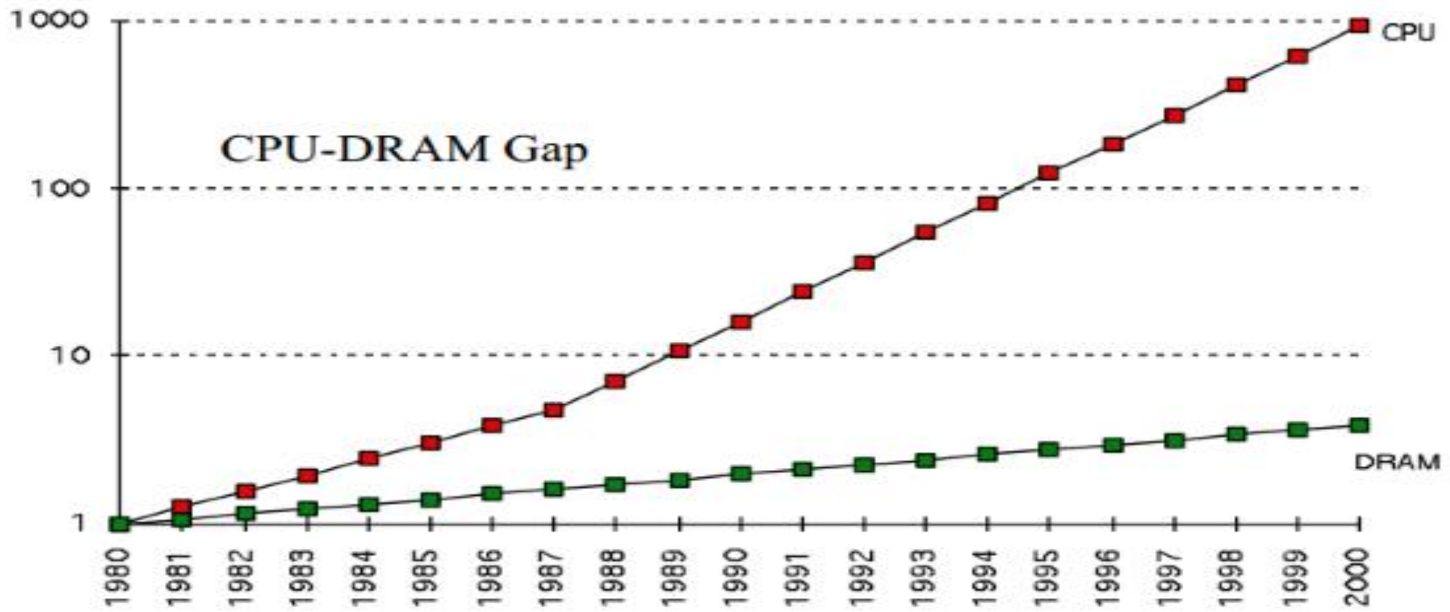
Indian Institute of Technology Guwahati, Assam.

Pipelined RISC Data path



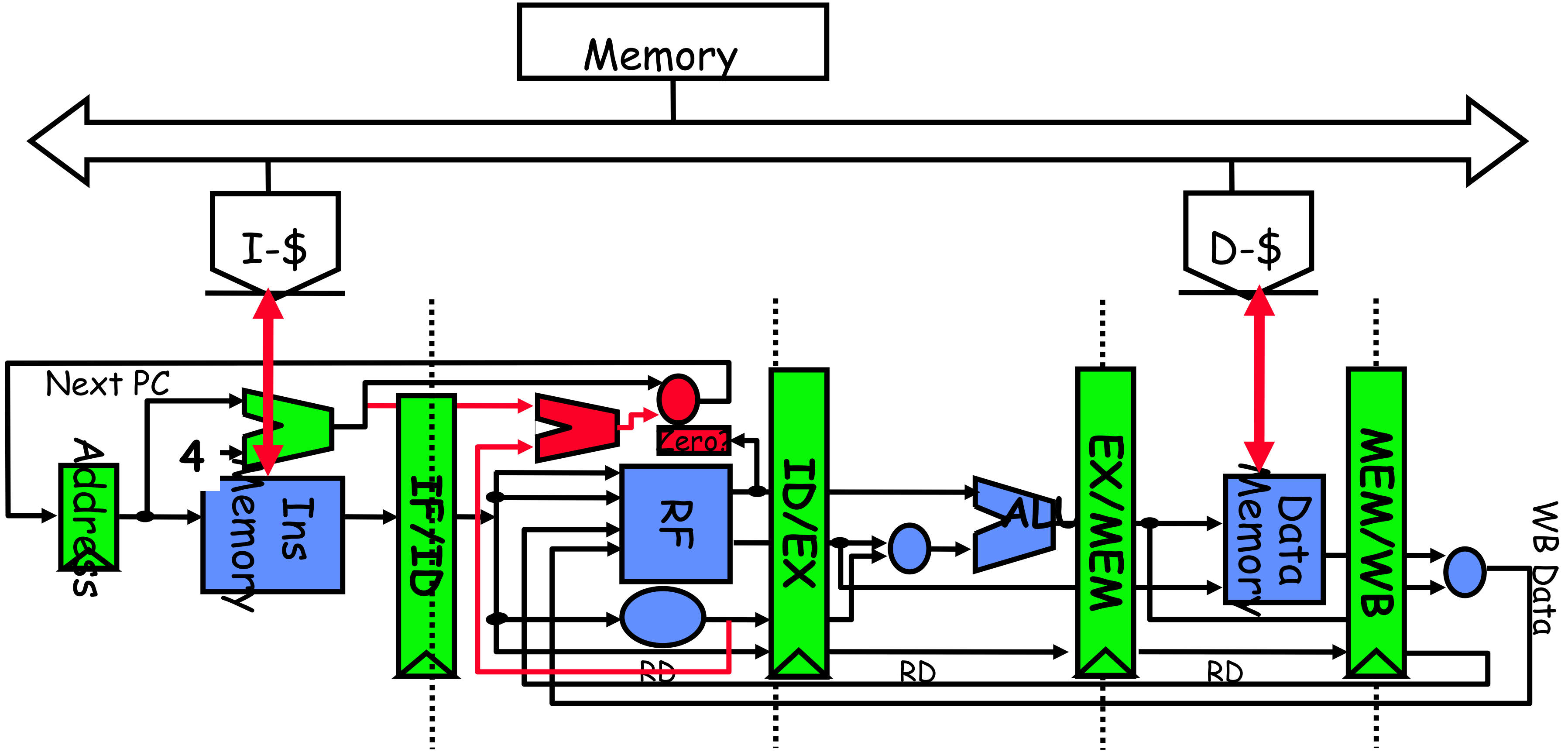
Processor Memory Performance Gap

- Processor vs Memory Performance



1980: no cache in microprocessor;
1995 2-level cache

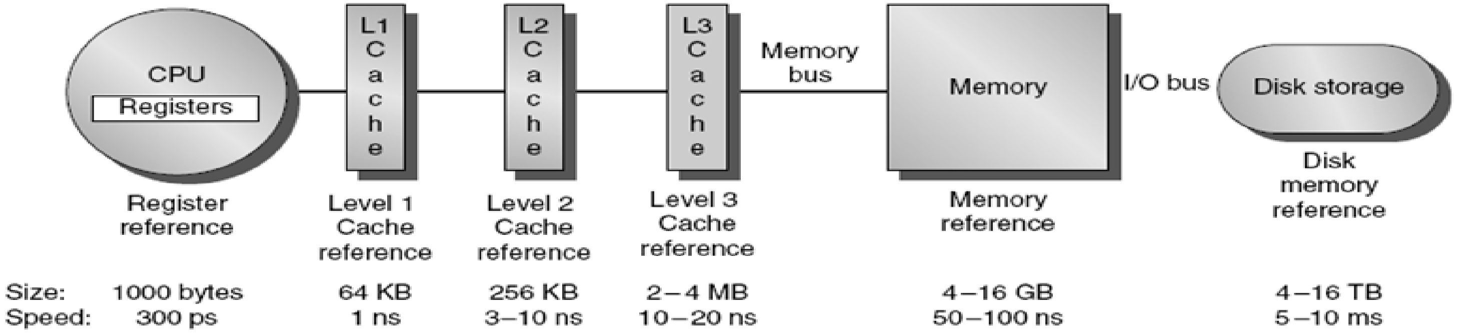
Relationship of Caches and Pipeline

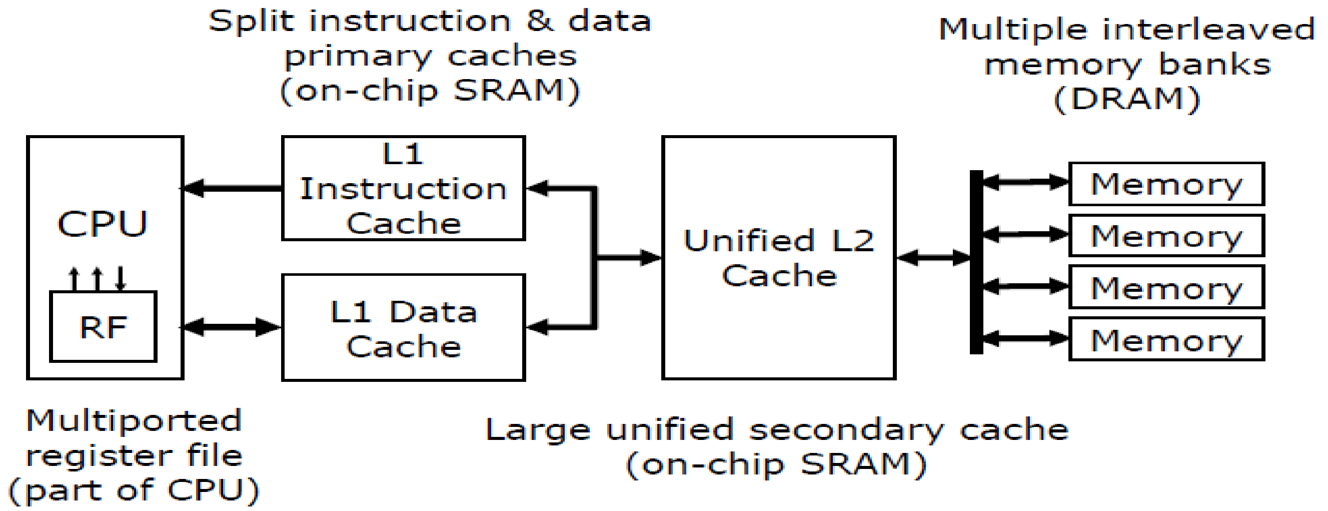


Role of memory

- Programmers want unlimited amount of fast memory
- Create the illusion of a very large and fast memory
- Implement the memory of a computer as a hierarchy
- Multiple levels of memory with different speeds and sizes
- Entire addressable memory space available in largest, slowest memory
- Keep the smaller and faster memories close to the processor and the slower, large memory below that.

Memory Hierarchy

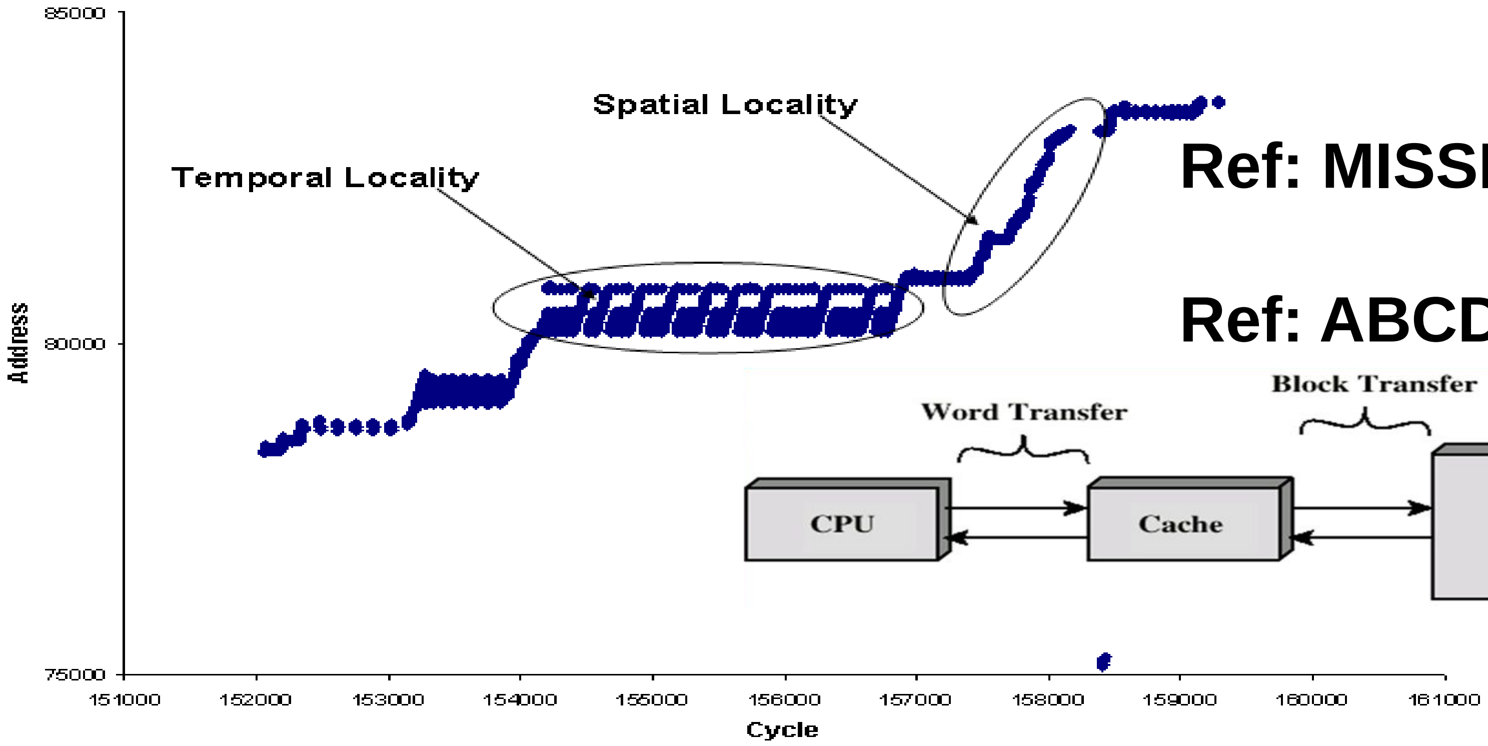



Split instruction & data primary caches (on-chip SRAM)
Multiple interleaved memory banks (DRAM)
CPU
RF
L1 Instruction Cache
L1 Data Cache
Unified L2 Cache
Memory
Memory
Memory
Memory
Multiported register file (part of CPU)
Large unified secondary cache (on-chip SRAM)

Cache Memory - Introduction

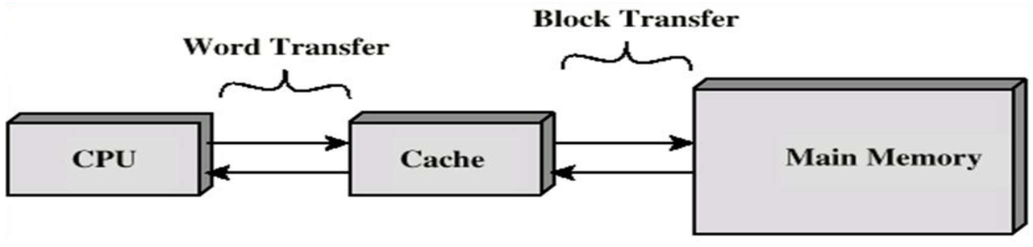
- Cache is a small, fast buffer between processor and memory
- Old values will be removed from cache to make space for new values
- **Principle of Locality :** Programs access a relatively small portion of their address space at any instant of time
- **Temporal Locality :** If an item is referenced, it will tend to be referenced again soon
- **Spatial Locality :** If an item is referenced, items whose addresses are close by will tend to be referenced soon

Access Patterns



Ref: MISSISSIPPI

Ref: ABCDEFAGHI



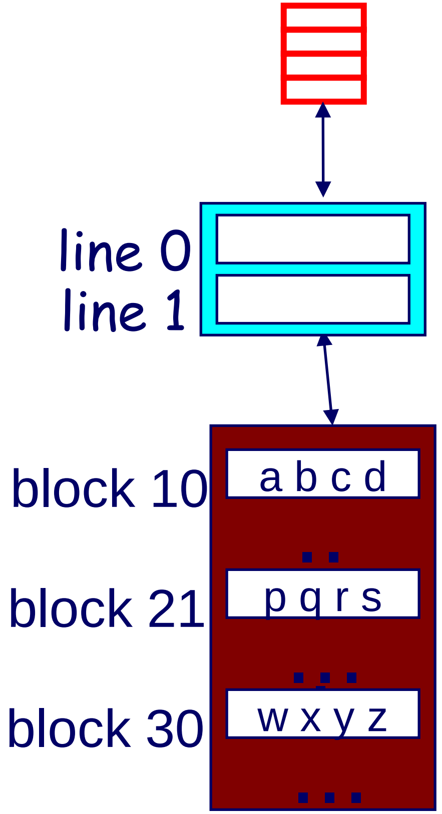
Cache Fundamentals

- **Block/Line :** Minimum unit of information that can be either present or not present in a cache level
- **Hit :** An access where the data requested by the processor is present in the cache
- **Miss :** An access where the data requested by the processor is not present in the cache
- **Hit Time :** Time to access the cache memory block and return the data to the processor.
- **Hit Rate / Miss Rate:** Fraction of memory access found (not found) in the cache
- **Miss Penalty :** Time to replace a block in the cache with the corresponding block from the next level.

CPU – Cache Interaction

❖The transfer unit between the CPU register file and the cache is a 4-byte word

❖The transfer unit between the cache and main memory is a 4-word block (16B)



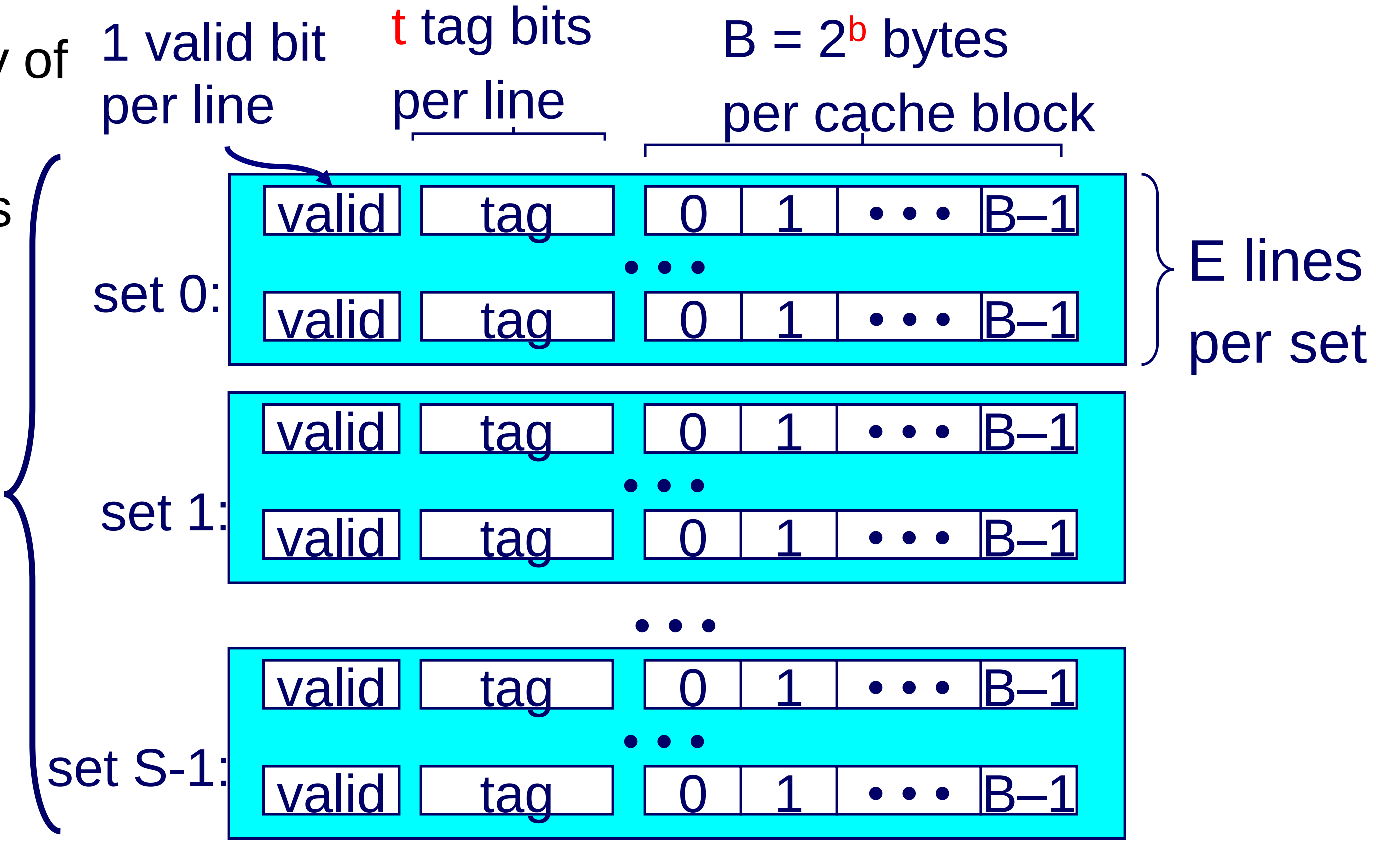
❖The tiny, very fast CPU register file has room for four 4-byte words

❖The small fast **L1 cache** has room for two 4-word blocks

❖The big slow main memory has room for many 4-word blocks

General Organization of a Cache

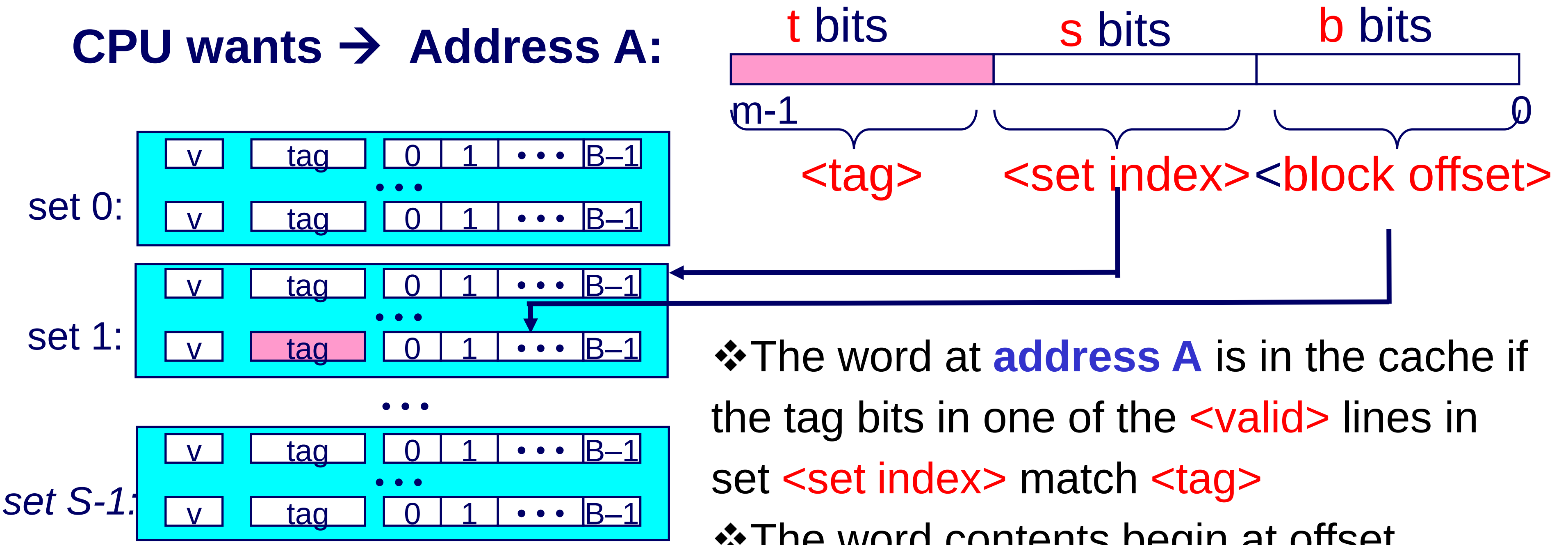
- ❖Cache is an array of sets
- ❖Each set contains one or more lines
- ❖Each line holds a block of data



Cache size: C = B x E x S data bytes

Addressing Caches

CPU wants → Address A:

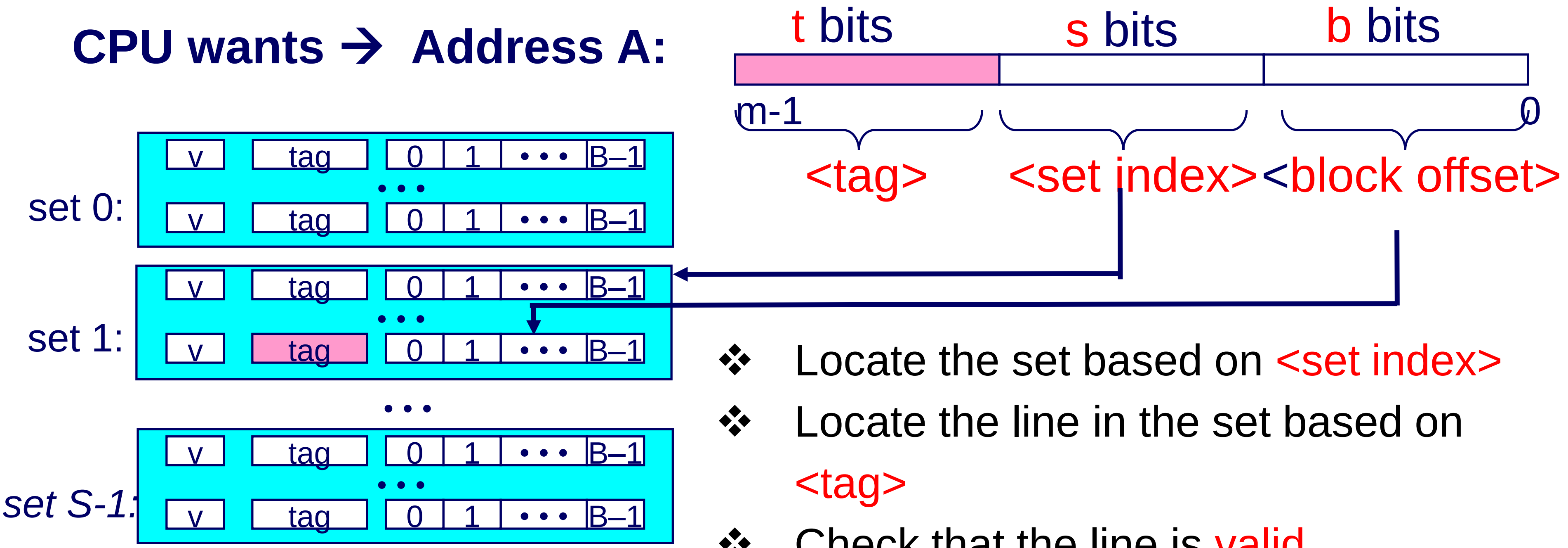


❖The word at **address A** is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>

❖The word contents begin at offset <block offset> bytes from the beginning of the block

Addressing Caches

CPU wants → Address A:



- Locate the set based on <set index>
- Locate the line in the set based on <tag>
- Check that the line is valid
- Locate the data in the line based on <block offset>

Index and Offset Calculations

A cache has 512 KB capacity, 4B word, 64B block size and 8-way set associative. The system is using 32 bit address. Given the address 0X ABC89984, which set of cache will be searched and specify which word of the selected cache block will be forwarded if it is a hit in cache?

# sets = CS/(BSxA) = $2^{19}/(2^6 \times 2^3) = 2^{10}$ = 1024 sets

1 word = 4B , Hence 64 byte block has 16 words

Tag = 16	**Index =10**	**Offset=6 (4+2)**

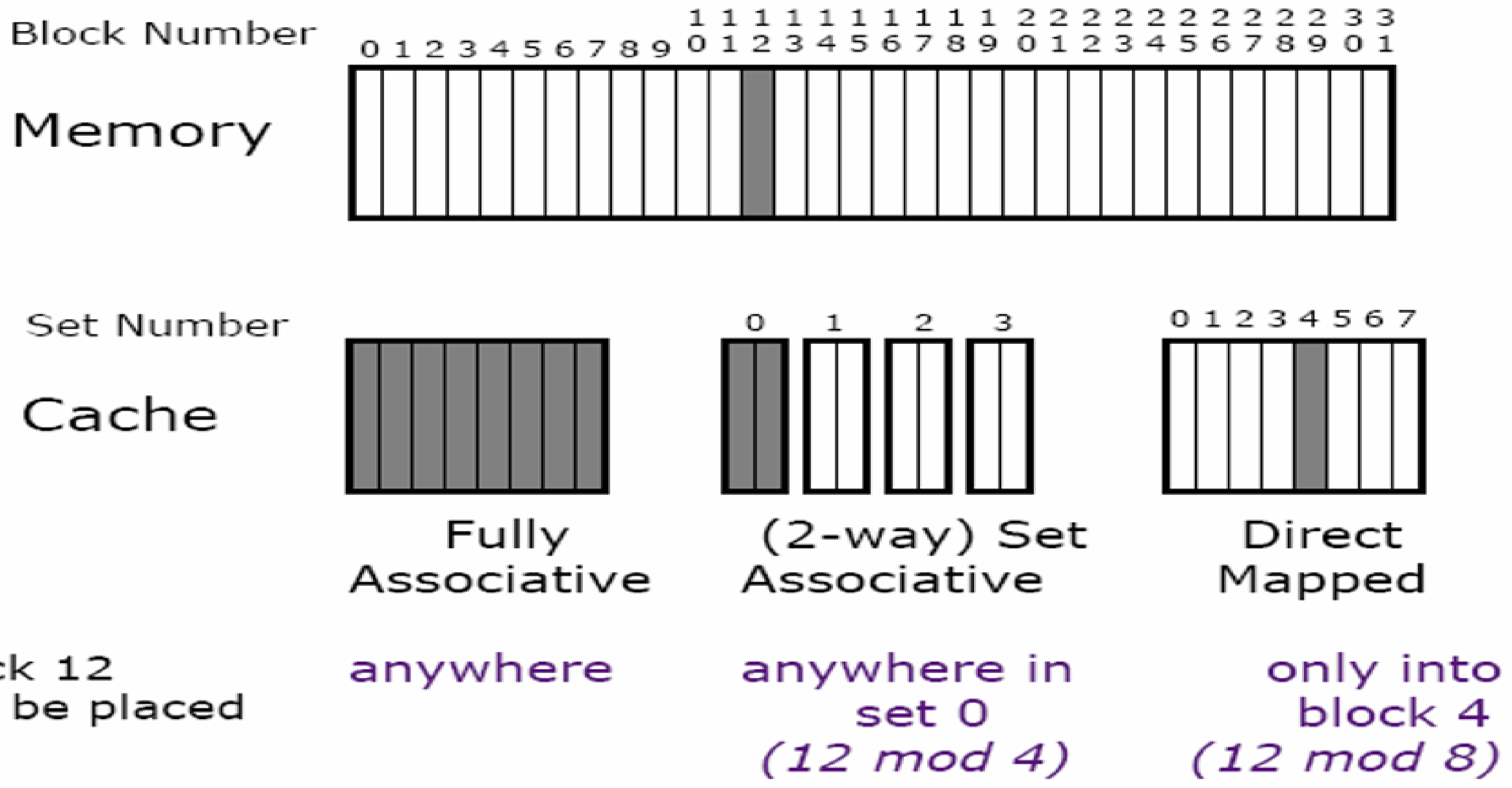
0x ABC89984 = 1001 1001 1000 0100 → Set 614, word 1

0x 485669AC = 0110 1001 1010 1100 → Set 422, word 11

Four cache memory design choices

- ❖ Where can a block be placed in the cache?
  - – **Block Placement**
- ❖ How is a block found if it is in the upper level?
  - – **Block Identification**
- ❖ Which block should be replaced on a miss?
  - – **Block Replacement**
- ❖ What happens on a write?
  - – **Write Strategy**

Block Placement

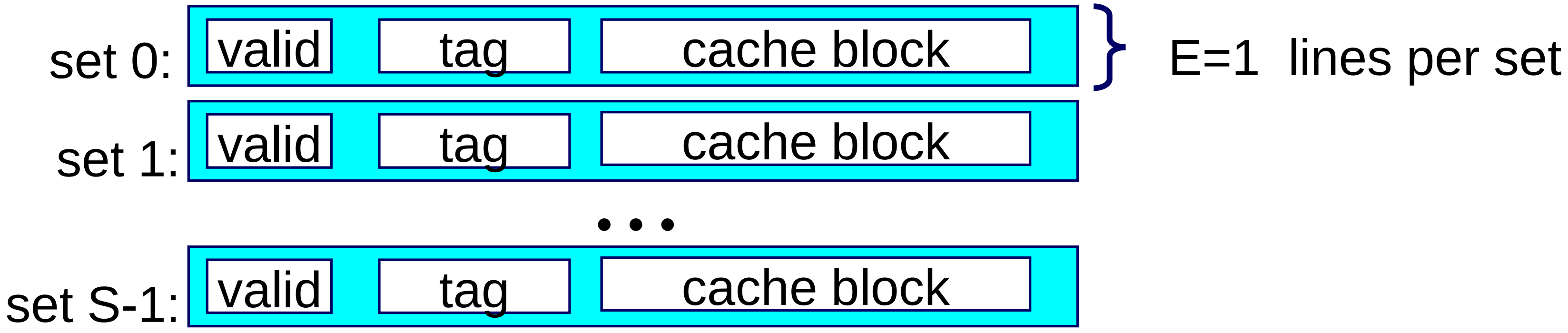


Cache Mapping / Block Placement

- **Direct mapped**
  - Block can be placed in only one location
  - (Block Number) **Modulo** (Number of blocks in cache)
- **Set associative**
  - Block can be placed in one among a list of locations
  - (Block Number) **Modulo** (Number of sets)
- **Fully associative**
  - Block can be placed anywhere

Direct-Mapped Cache

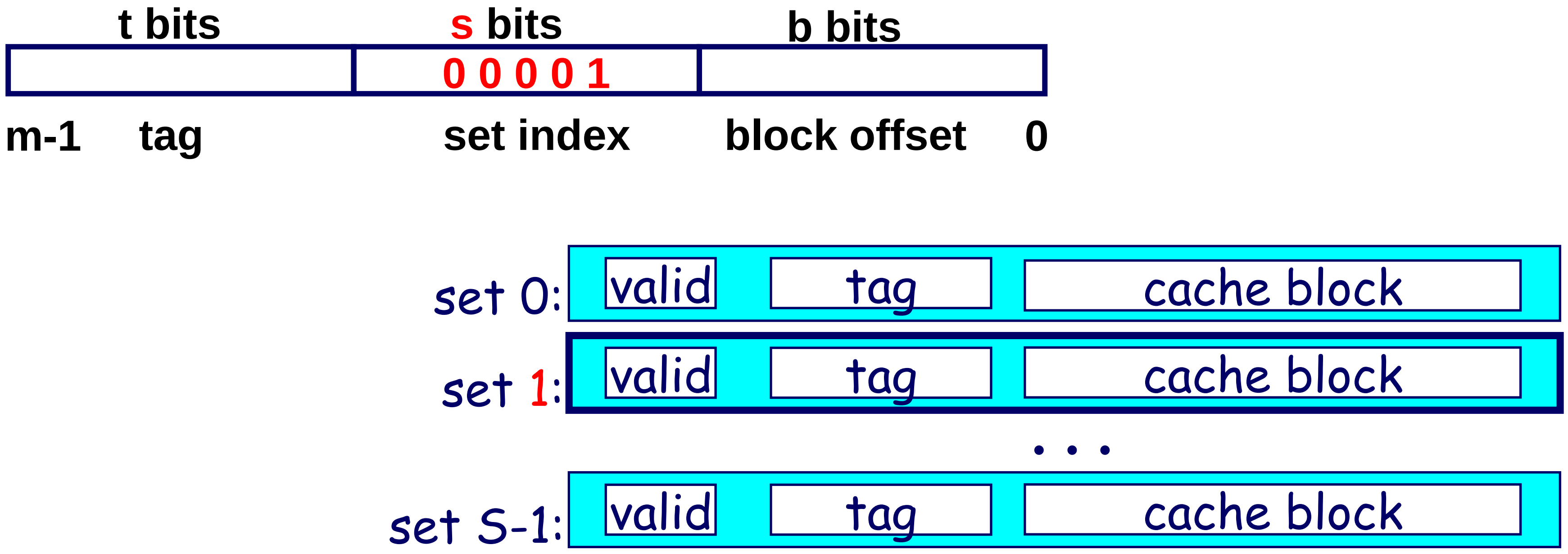
- Simplest kind of cache, easy to build
- Only 1 tag compare required per access
- Characterized by exactly one line per set.



Cache size: C = B x S data bytes

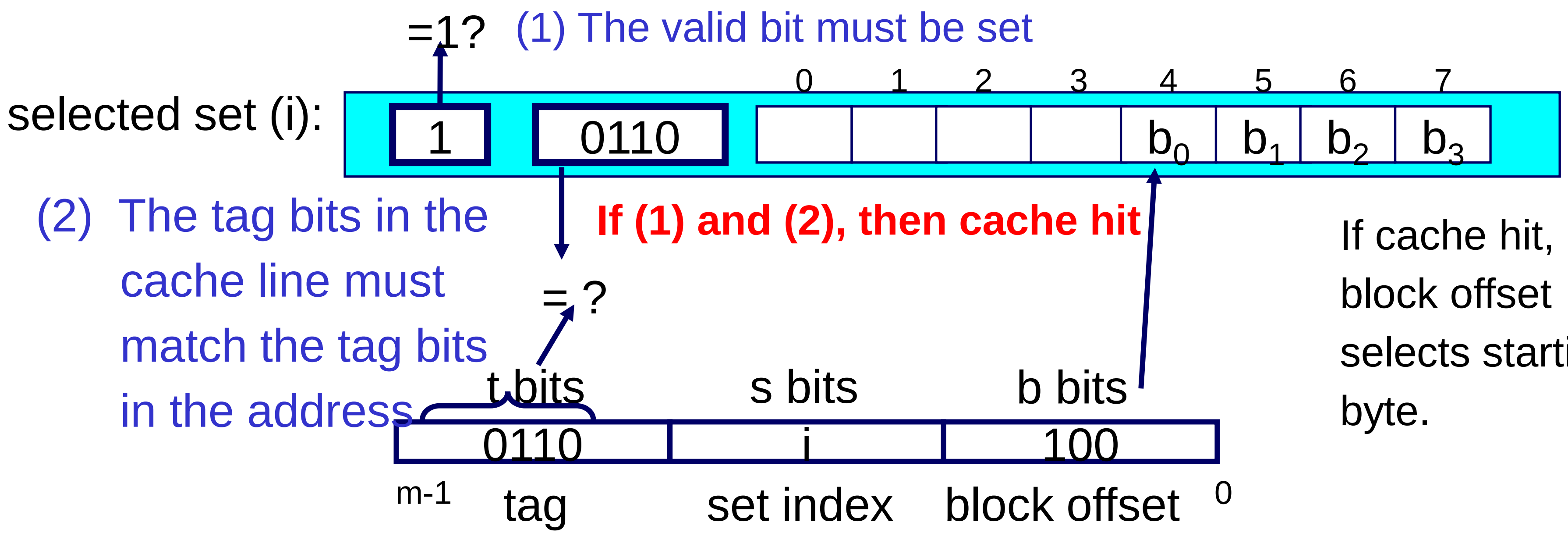
Accessing Direct-Mapped Caches

- Set selection is done by the set index bits



Accessing Direct-Mapped Caches

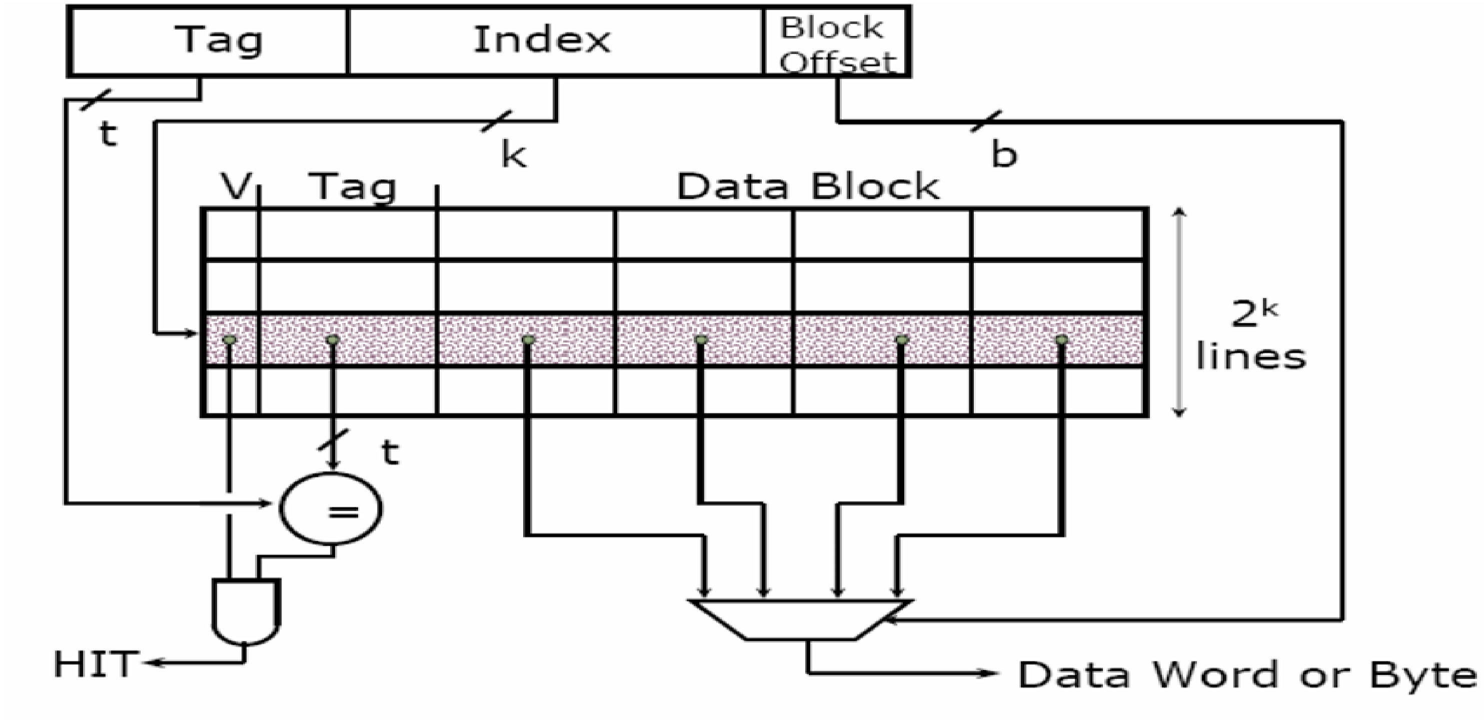
- Block matching: Find a valid block in the selected set with a matching tag
- Word selection: Then extract the word



=1?
(1) The valid bit must be set
selected set (i):
0
1
2
3
4
5
6
7
1
0110
b_0
b_1
b_2
b_3
(2) The tag bits in the cache line must match the tag bits in the address
If (1) and (2), then cache hit
= ?
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

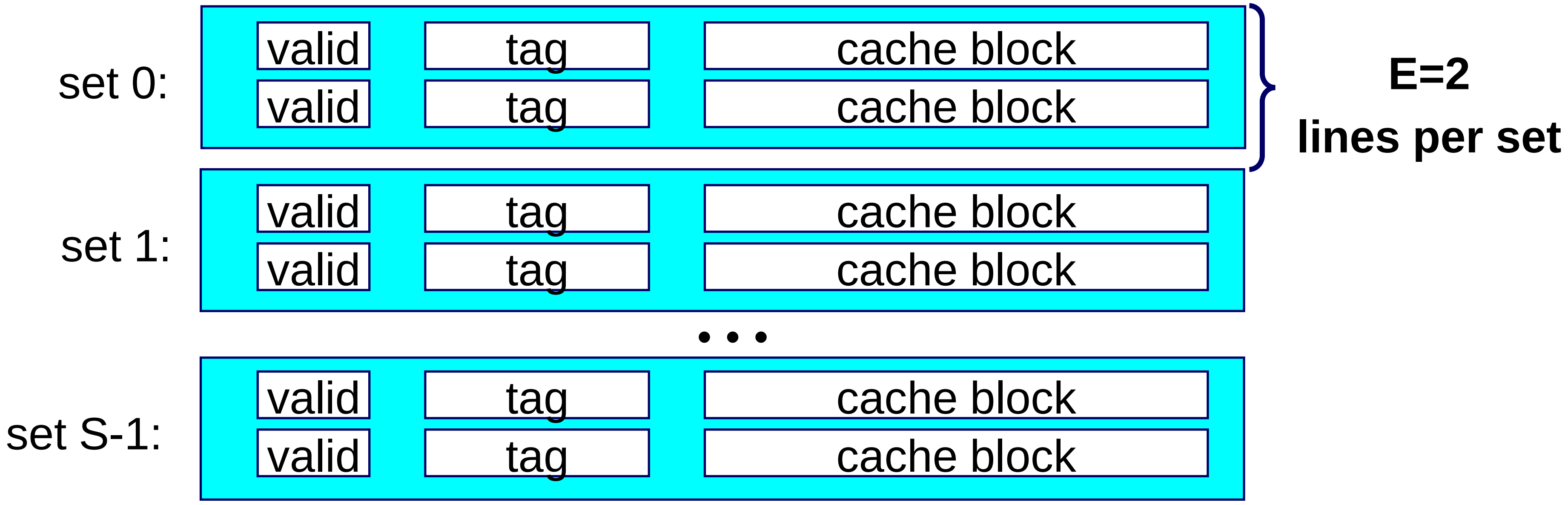
If cache hit, block offset selects starting byte.

Block Identification – Direct mapped



Set Associative Cache

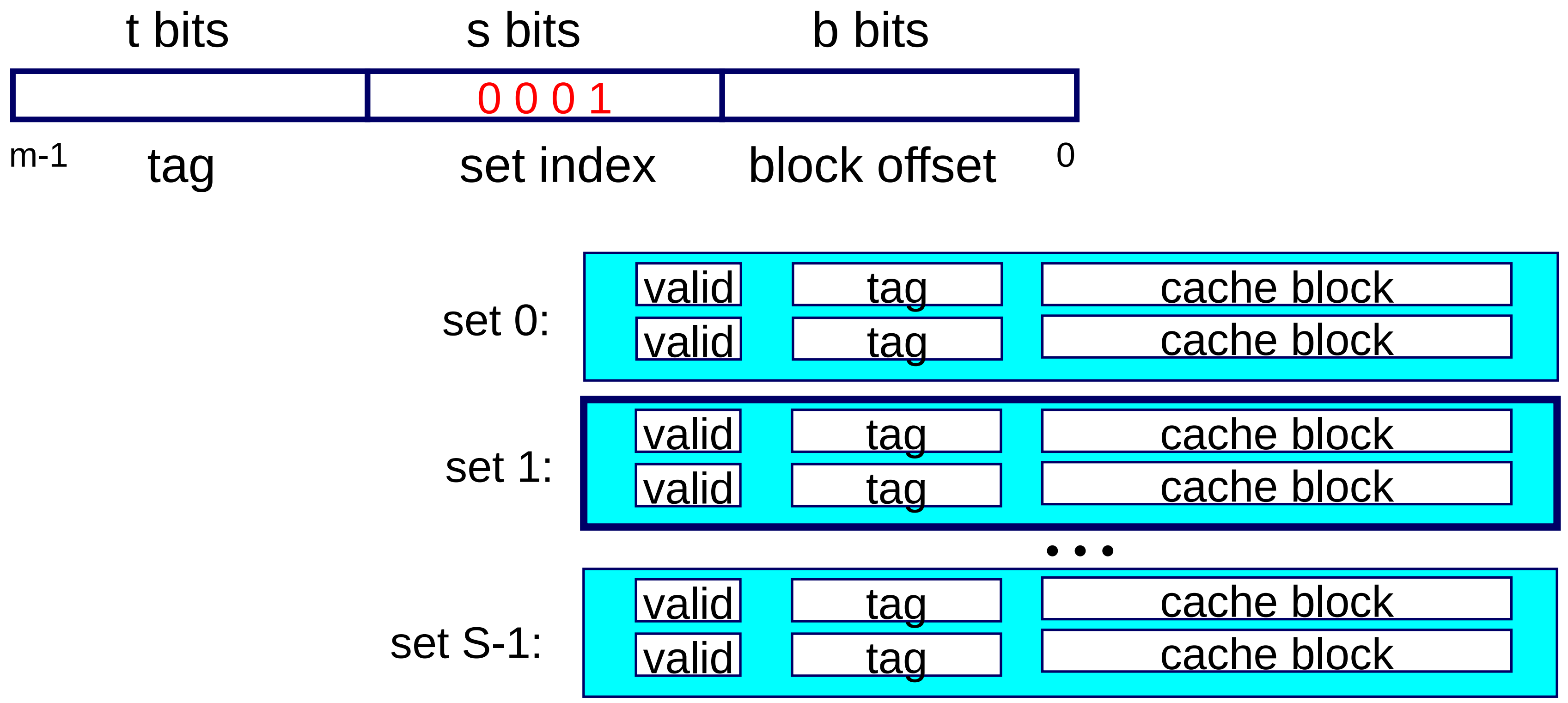
- Characterized by more than one line per set



2-way associative cache

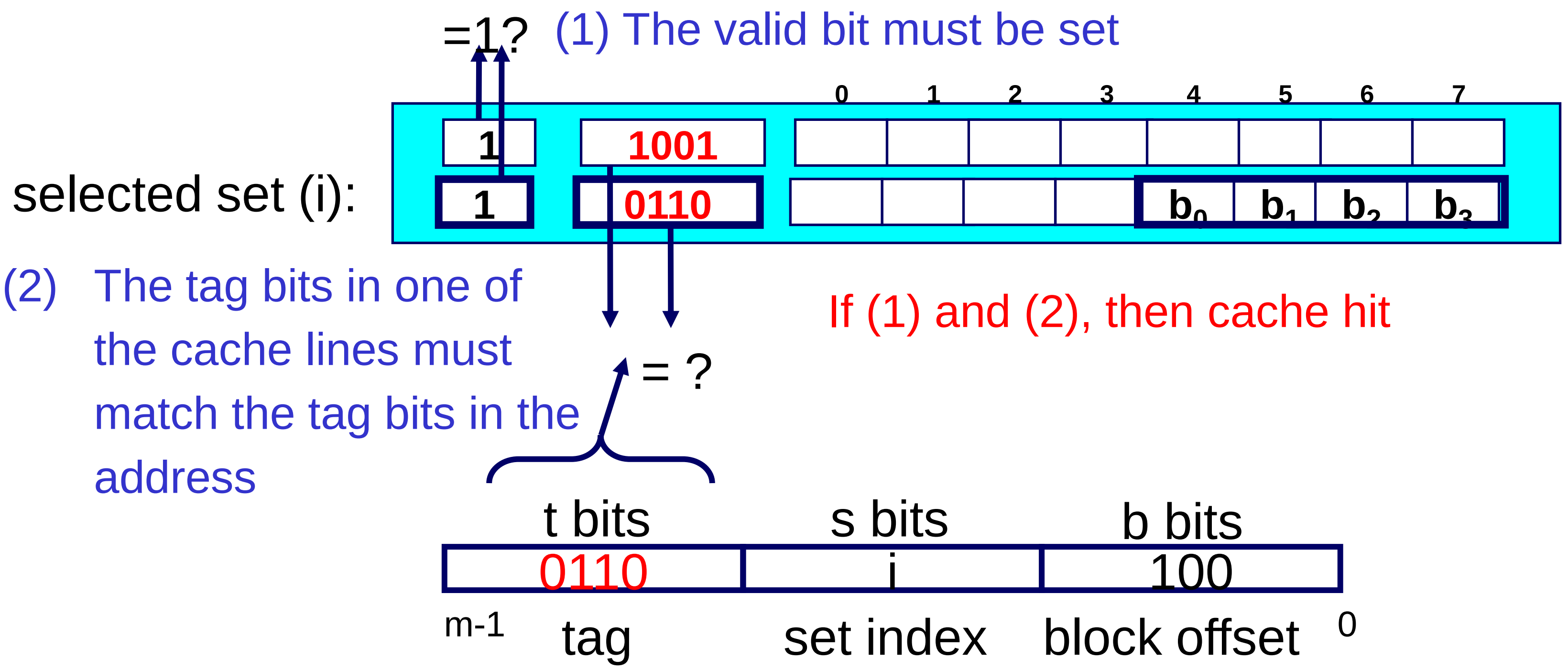
Accessing Set Associative Caches

- Set selection is identical to direct-mapped cache



Accessing Set Associative Caches

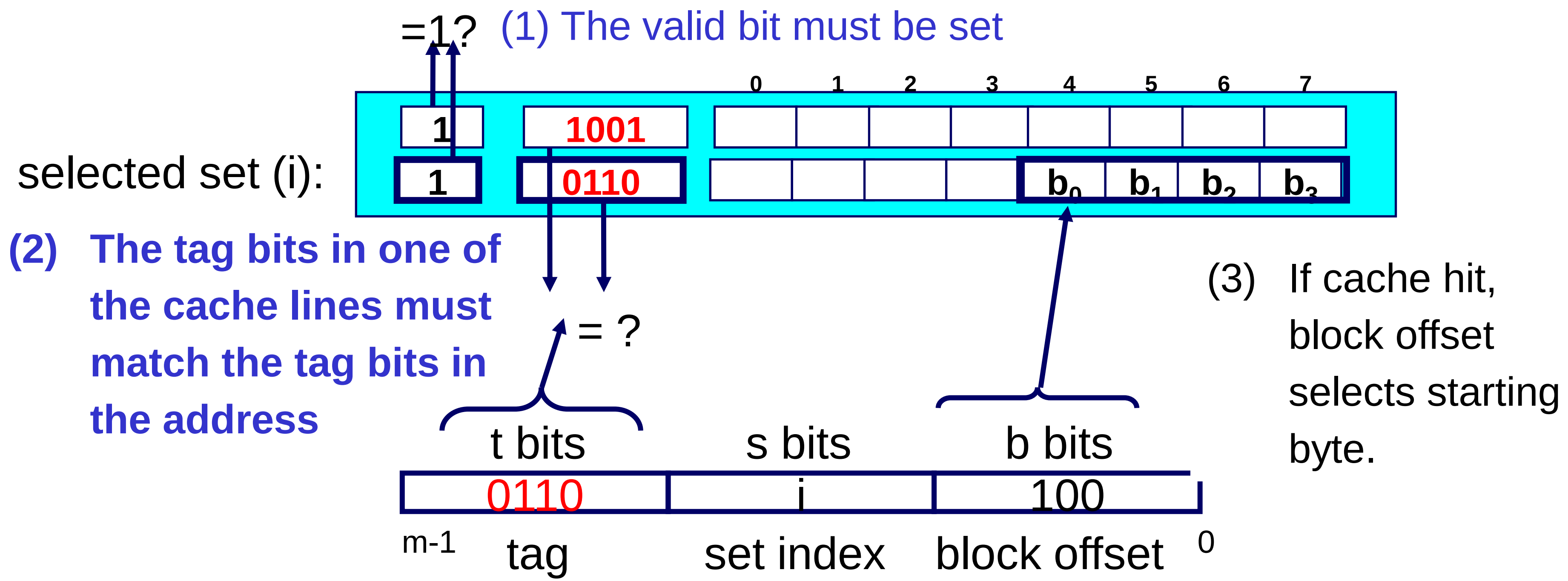
- Block matching is done by comparing the tag in each valid line in the selected set.



=1?
(1) The valid bit must be set
0
1
2
3
4
5
6
7
1
1001
selected set (i):
1
0110
b_0
b_1
b_2
b_3
(2) The tag bits in one of the cache lines must match the tag bits in the address
If (1) and (2), then cache hit
= ?
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

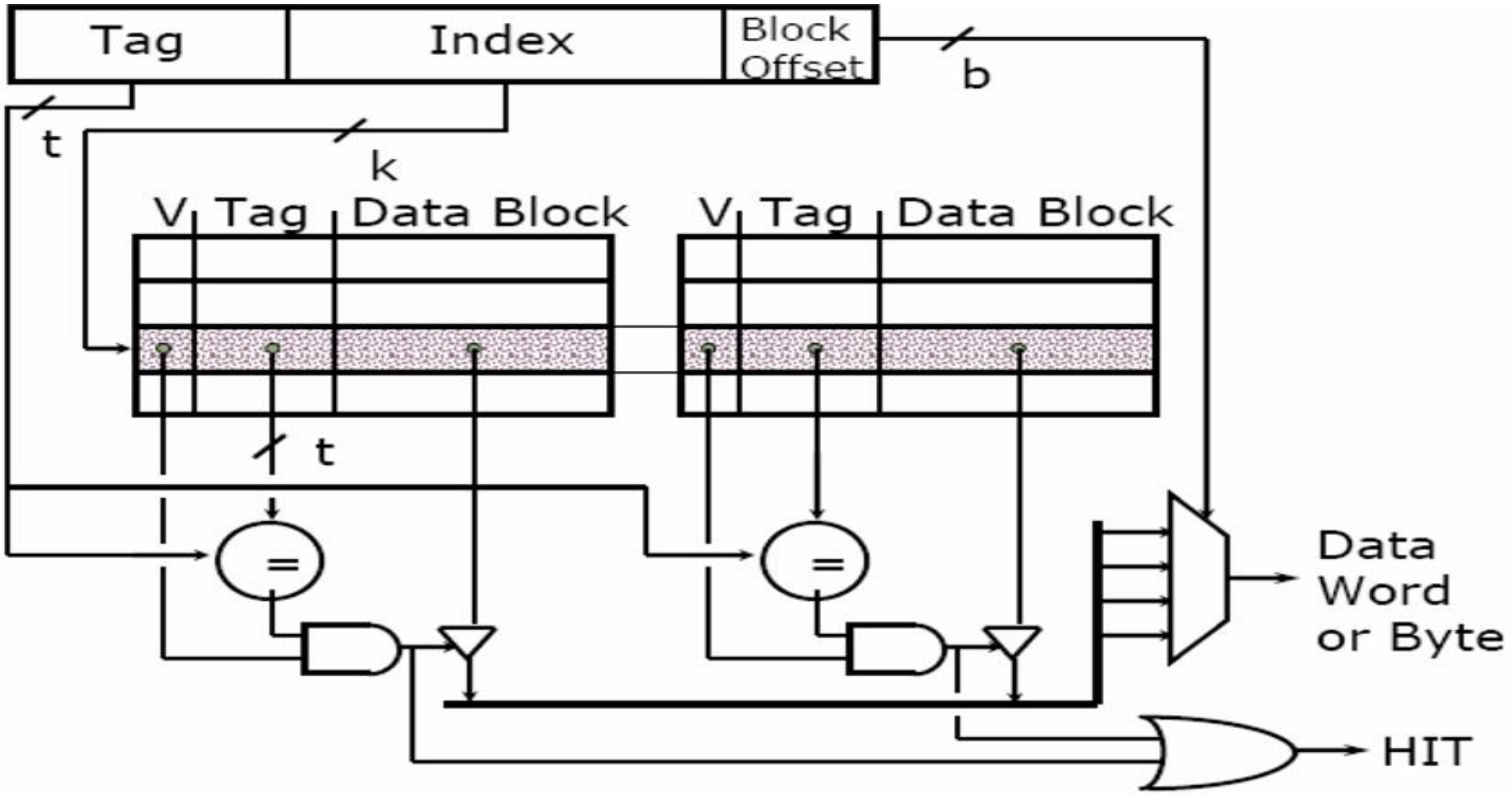
Accessing Set Associative Caches

❖ Word selection is done same as direct mapped cache but chosen only on the line has produced a hit.

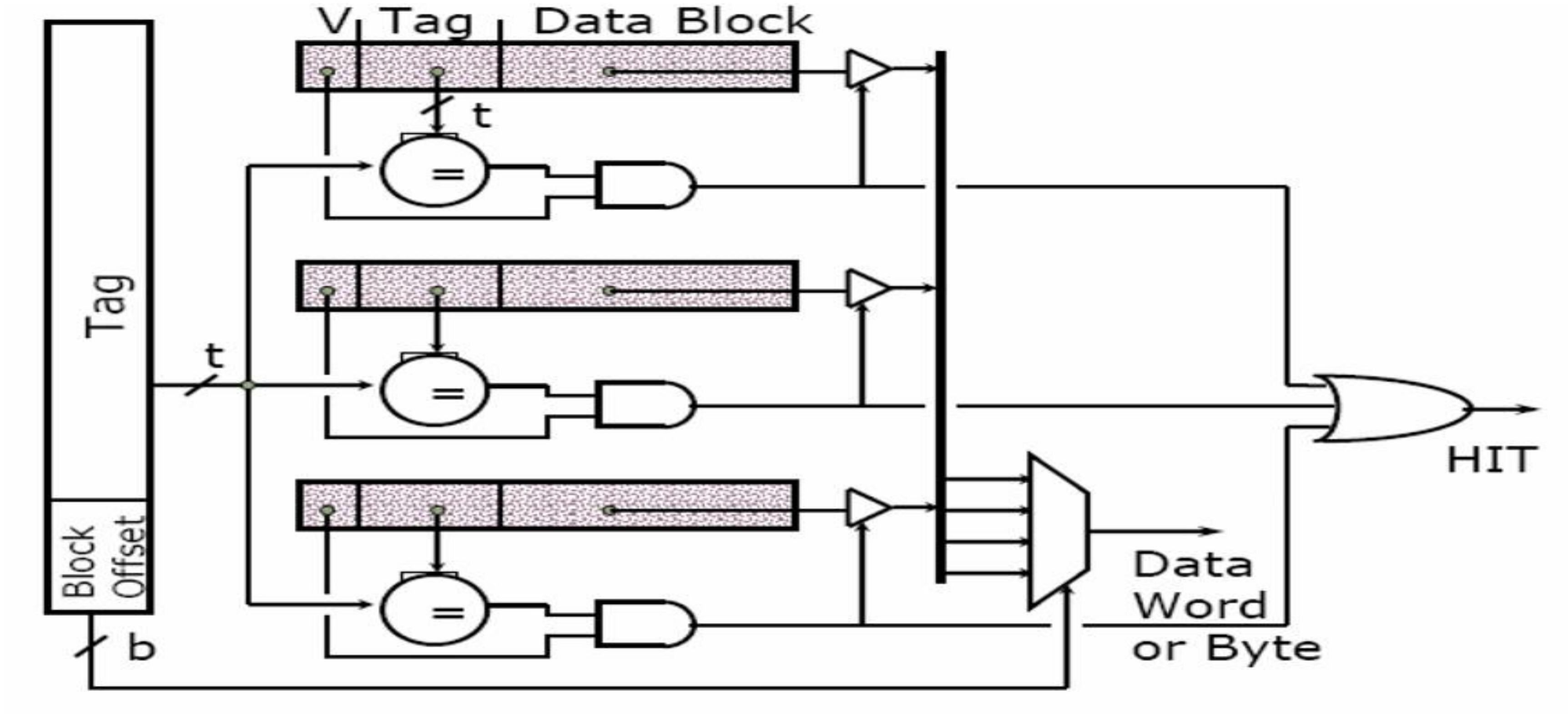


=1?
(1) The valid bit must be set
0
1
2
3
4
5
6
7
1
1001
selected set (i):
1
0110
b_0
b_1
b_2
b_3
(2) The tag bits in one of the cache lines must match the tag bits in the address
= ?
(3) If cache hit, block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

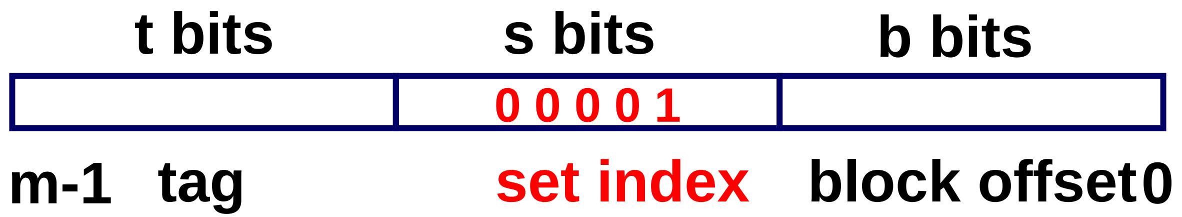
Block Identification – Set Associative



Block Identification – Fully Associative

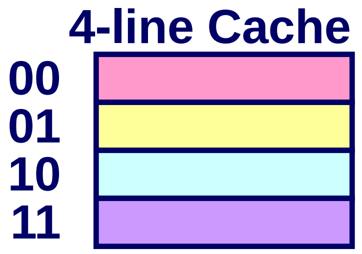


Cache Indexing



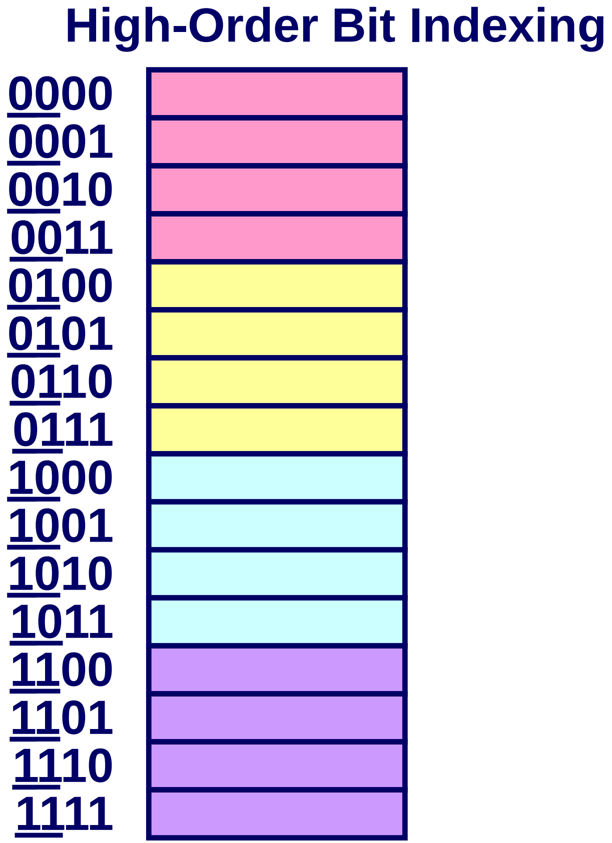
- Decoders are used for indexing
- Indexing time depends on decoder size (s: 2^s)
- Smaller number of sets, less indexing time.

Why Use Middle Bits as Index?

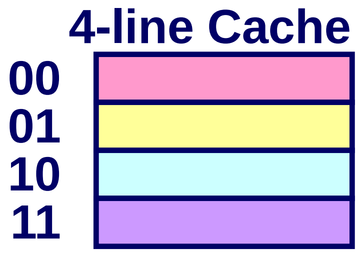


High-Order Bit Indexing

- ❖ Adjacent memory lines would map to same cache entry
- ❖ Poor use of spatial locality

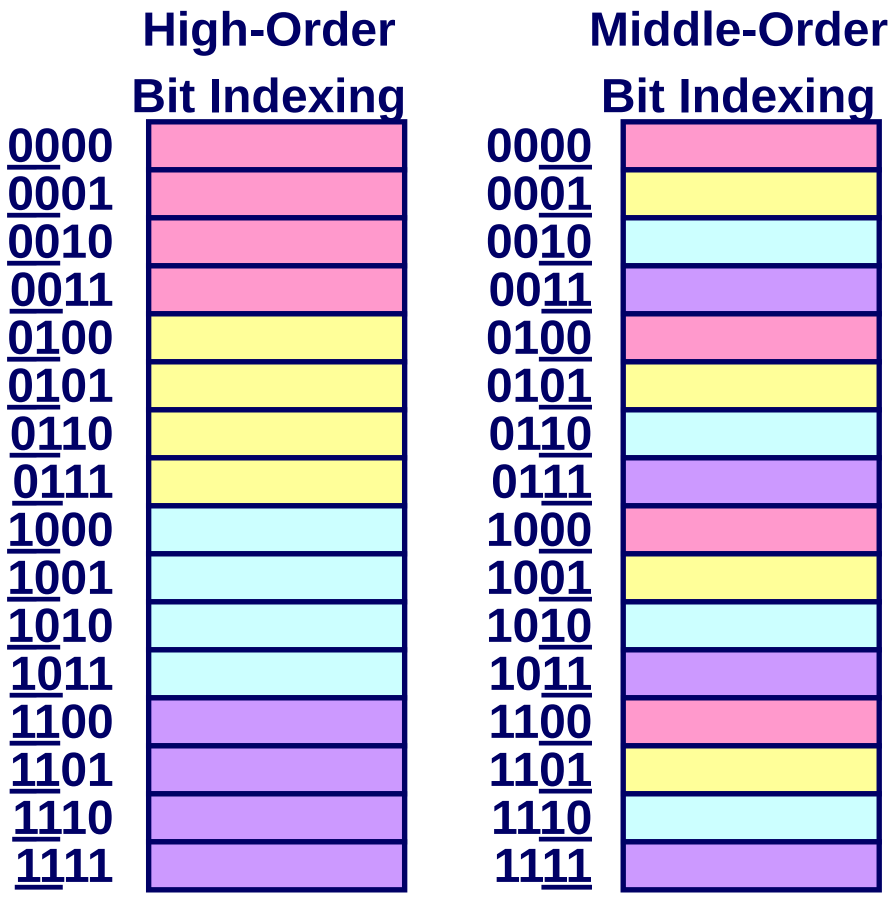


Why Use Middle Bits as Index?



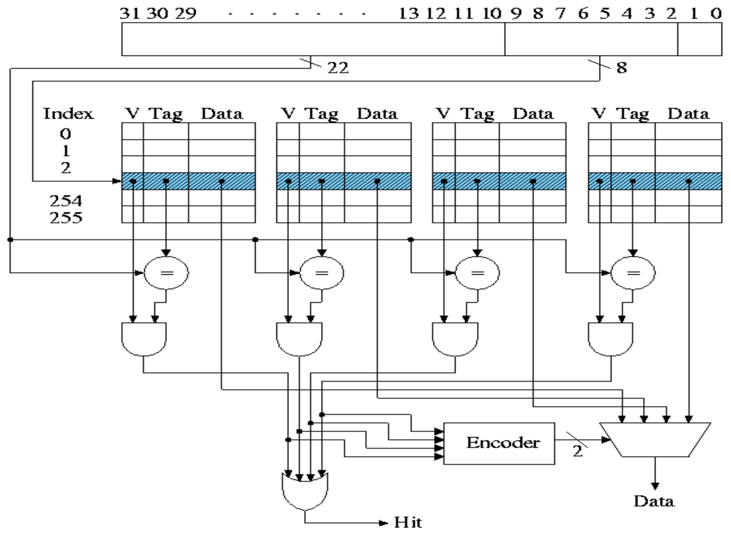
Middle-Order Bit Indexing

- ❖Consecutive memory lines map to different cache lines
- ❖Better use of spacial locality without replacement



Block Replacement

- Cache has finite size. What do we do when it is full?
- Direct Mapped is Easy
- Which block to be replaced for a set associative cache?



Block Replacement Algorithms

- Random
- First In First Out (FIFO)
- Last In First Out (LIFO)
- Least Recently Used (LRU)
- Pseudo-LRU (PLRU)
- Not Recently Used (NRU)
- Least Frequently Used (LFU)
- Re-Reference Interval Prediction (RRIP)
- Optimal

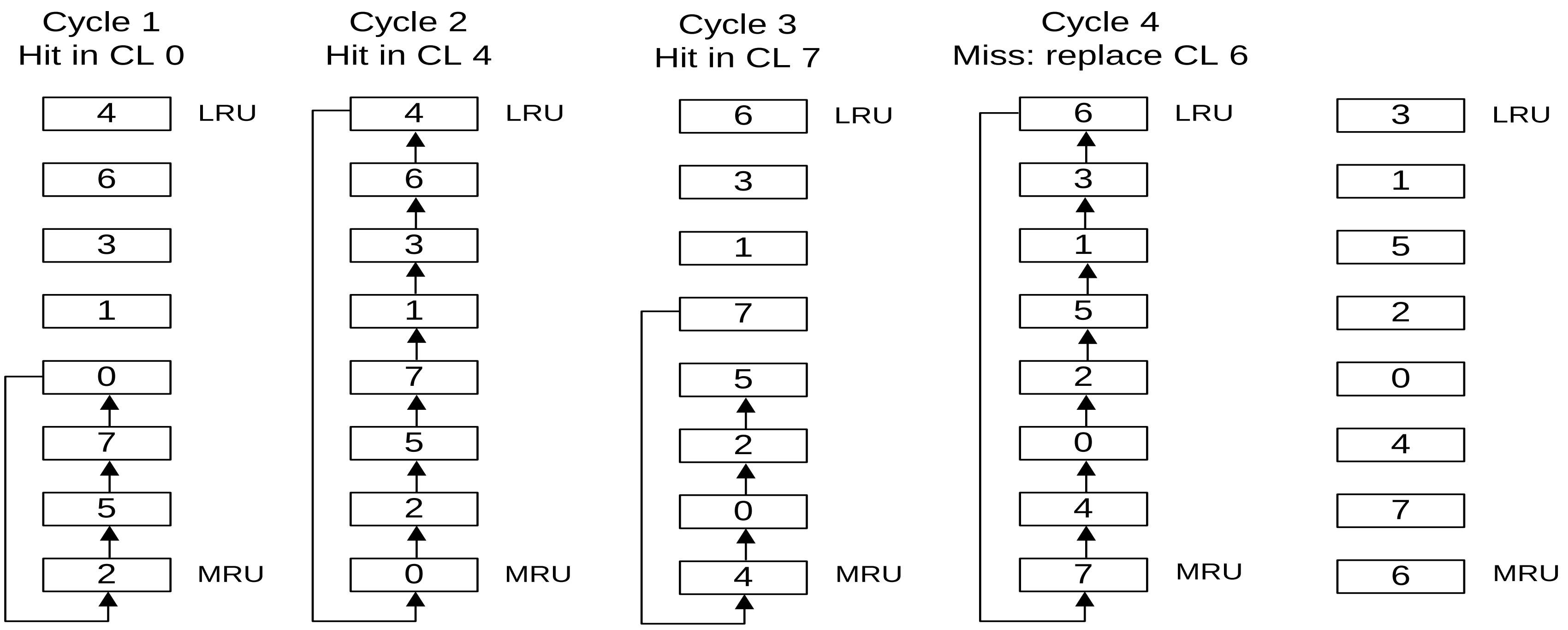
Random & FIFO Replacement Policy

- **Random policy** needs a pseudo-random number generator
- Makes no attempt to take advantage of any temporal or spatial localities
- **First-in, First-out(FIFO) policy** evict the block that has been in the cache the longest
- It requires a queue Q to store references
- Blocks are enqueued in Q, dequeue operation on Q to determine which block to evict.

Least-Recently Used

- For associativity =2, LRU is equivalent to NMRU
  - Single bit per line indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Record Timestamps? How many bits?
  - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
  - Shift register implementation

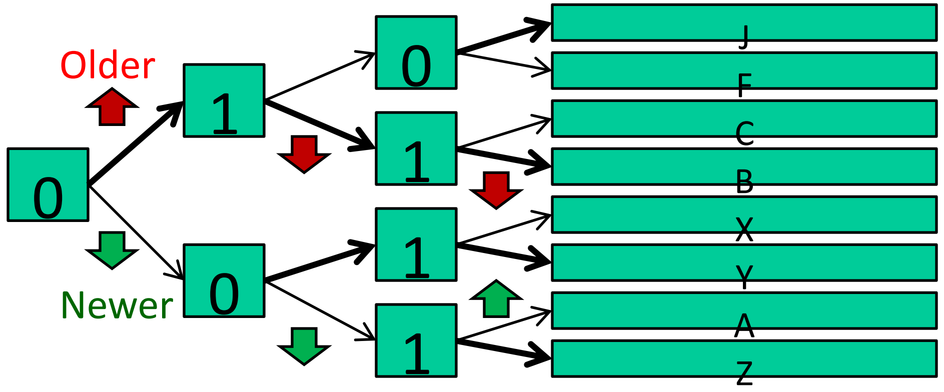
LRU Implementation



Optimal Replacement Policy

- ❖ Evict block with longest reuse distance
  - ❖ i.e. next reference to block is farthest in future
  - ❖ Requires knowledge of the future!
- ❖ Can’t build it, but can model it with trace
- ❖ Useful, since it reveals opportunity
- ❖ Optimal better than LRU
  - ❖ (X,A,B,C,D,X): LRU 4-way SA cache, 2nd X will miss

Practical Pseudo-LRU



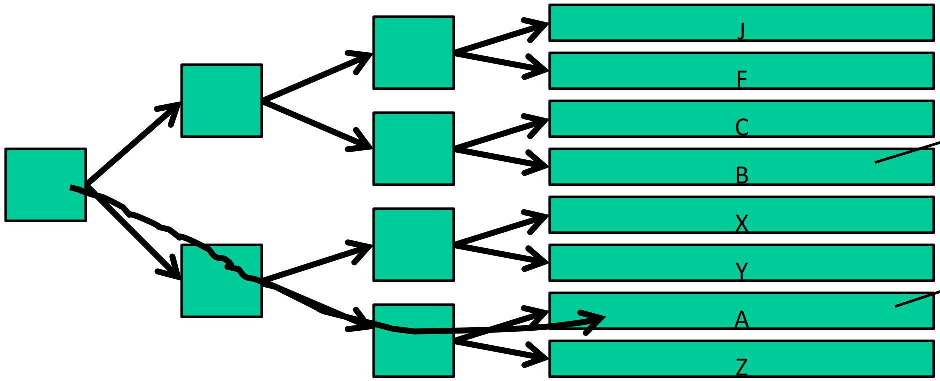
❖ Rather than true LRU, use binary tree

❖ Each node records which half is older/newer

❖ Update nodes on each reference

❖ Follow older pointers to find LRU victim

Practical Pseudo-LRU



J Y X Z B C F A

011: PLRU Block B is here

110: MRU block is here

Partial Order Encoded in Tree:

Z < A

Y < X

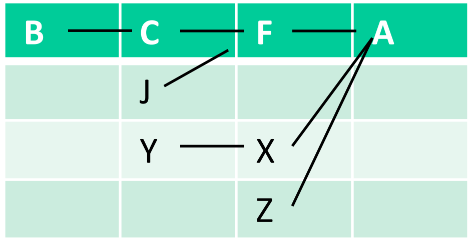
B < C

J < F

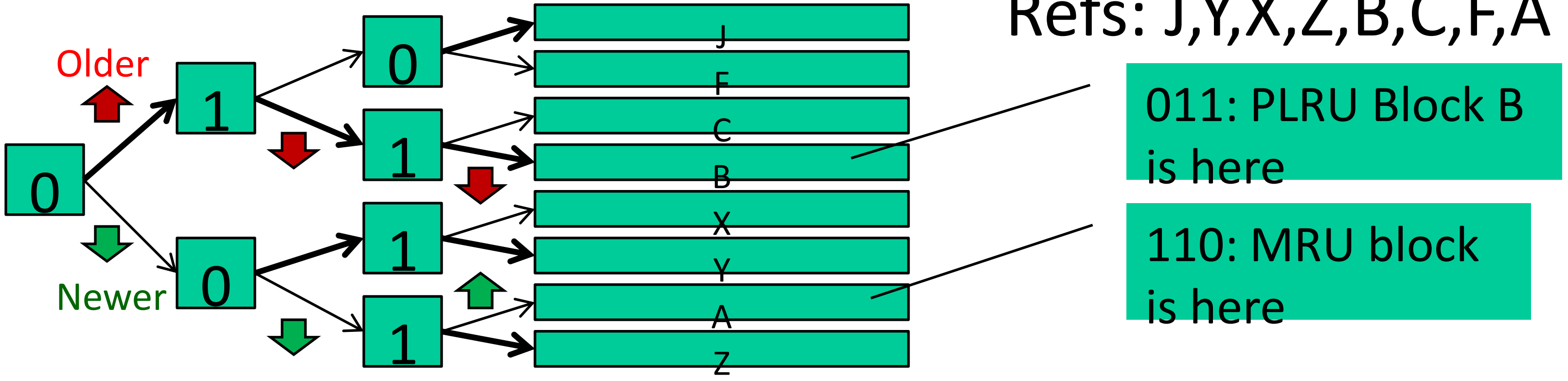
A > X

C < F

A > F



Practical Pseudo-LRU



- Binary tree encodes PLRU partial order
- At each level point to LRU half of subtree
- Each access: flip nodes along path to block
- Eviction: follow LRU path
- Overhead: *(a-1)/a* bits per block

Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is reset to 0 when installed / re referenced
  - Bit is set to 1 when it is not referenced and other block in the same set is referenced
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0 / 1 then pick by random
- Provides some scan and thrash resistance
- Randomizing evictions rather than strict LRU order

Re-reference Interval Prediction

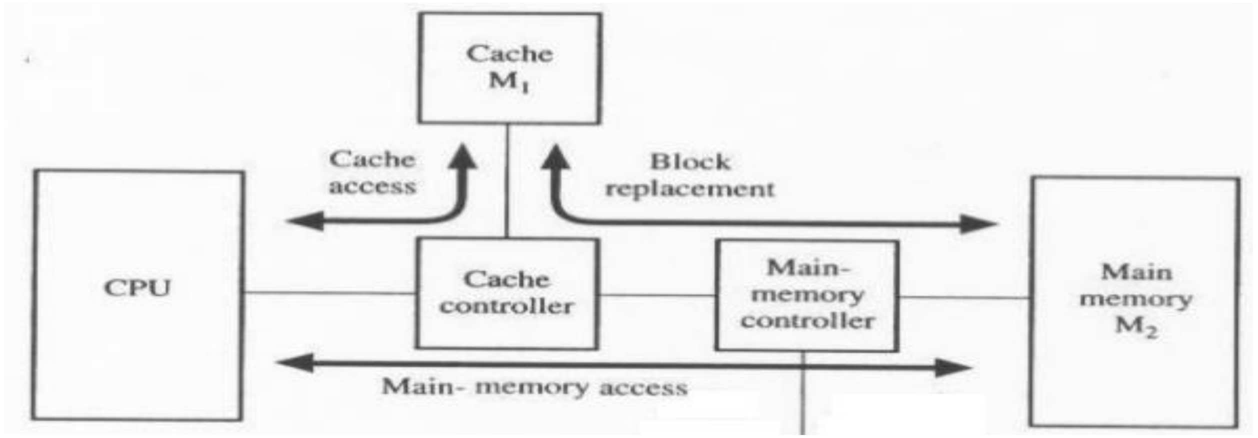
- RRIP
- Extends NRU to multiple bits
  - Start in the middle
  - promote on hit
  - demote over time
- Can predict near-immediate, intermediate, and distant re-reference

Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
- Logic not trivial (a^2 comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?

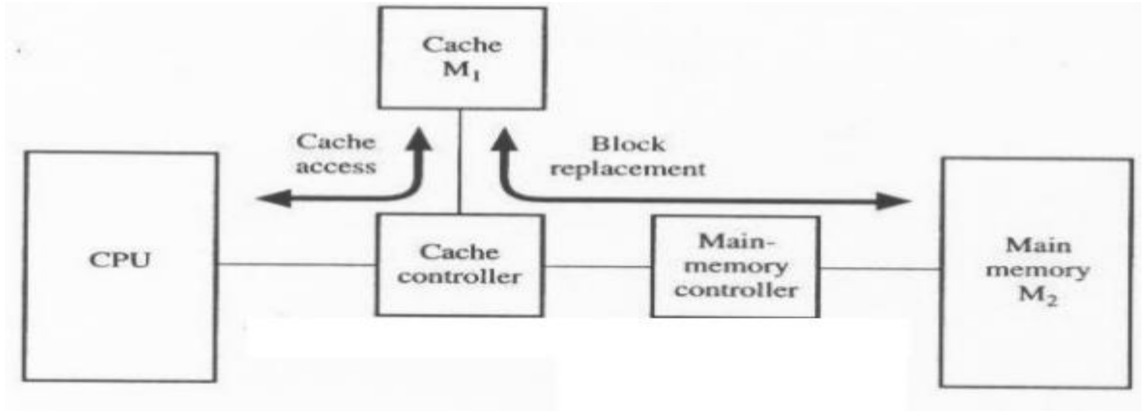
Look-aside vs Look through caches

- **Look-aside cache:** Request from processor goes to cache and main memory in parallel
- Cache and main memory both see the bus cycle
- On cache hit→ processor loaded from cache, bus cycle terminates; On cache miss: processor & cache loaded from memory in parallel



Look-aside vs Look through caches

- ❖ **Look-through cache:** Cache checked first when processor requests data from memory
- ❖ On hit→ data loaded from cache: On miss→ cache loaded from memory, then processor loaded from cache

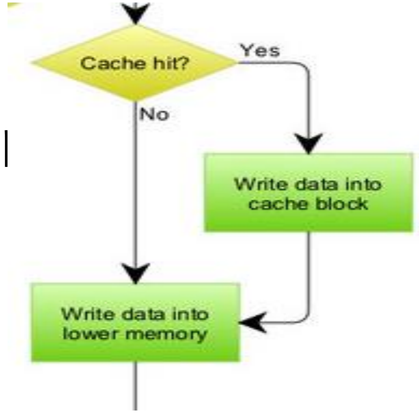


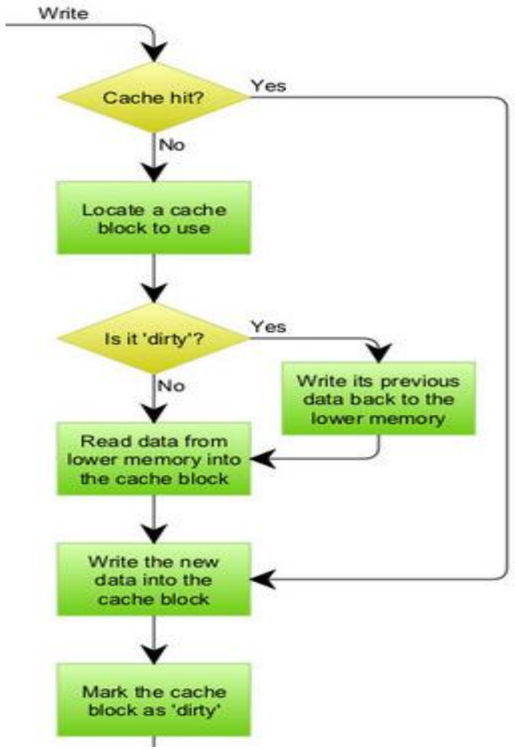
Write strategy

- **Write Hits → Write through vs Write back**
- **Write Miss→ Write allocate vs No-Write allocate**
- **Write through:** The information is written to both the block in the cache and to the main memory
- Read misses do not need to write back evicted line contents
- **Write back:** The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
- Have to maintain whether block clean or dirty. No extra work on repeated writes; only the latest value on eviction gets updated in main memory.

Write strategy

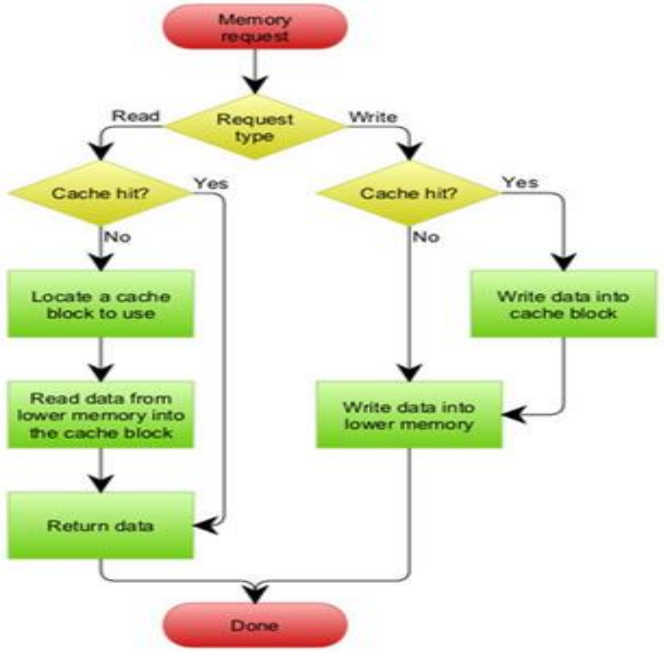
- **Write allocate:** The block is loaded into cache on a write miss.
- Used along with write back caches
- **No-Write allocate:** The block is modified in the main memory but not in cache
- Used along with write thr caches



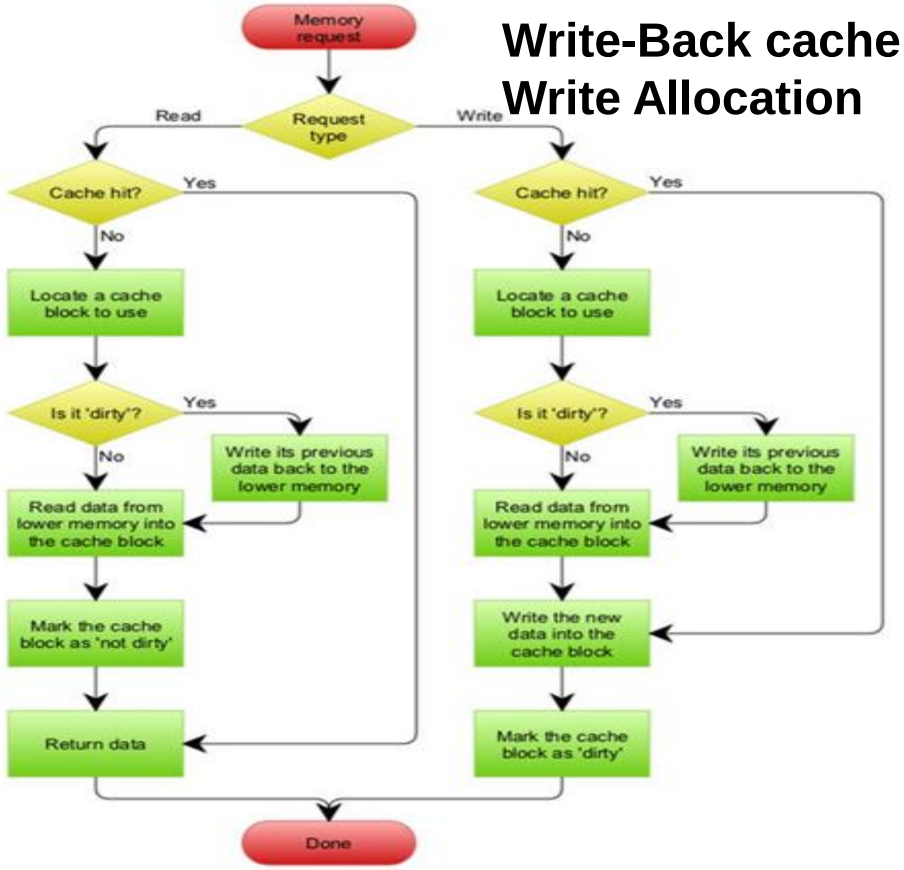

Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'

Write strategy

Write-Through cache with No-Write Allocation


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Read data from lower memory into the cache block
Return data
Cache hit?
Yes
No
Write data into cache block
Write data into lower memory
Done

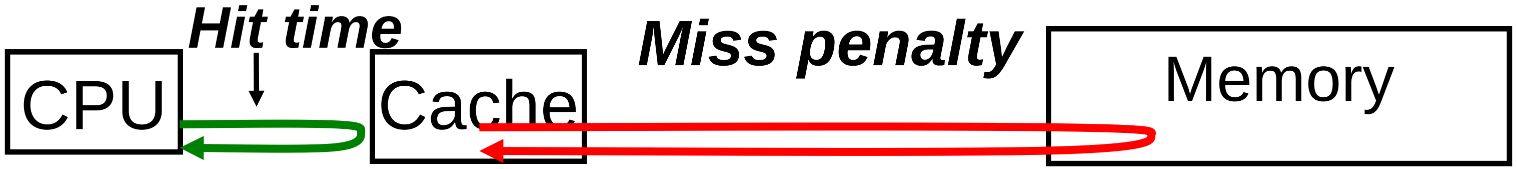
Write-Back cache with Write Allocation


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

Types of Cache Misses

- **Compulsory**
  - Very first access to a block
  - Will occur even in an infinite cache
- **Capacity**
  - If cache cannot contain all the blocks needed
  - Misses in fully associative cache (due to the capacity)
- **Conflict**
  - If too many blocks map to the same set
  - Occurs in associative or direct mapped cache

Accessing Cache Memory



Average memory access time = Hit time + (Miss rate × Miss penalty)

- **Hit Time:** Time to find the block in the cache and return it to processor *[indexing, tag comparison, transfer].*
- **Miss Rate:** Fraction of cache access that result in a miss.
- **Miss Penalty:** Number of additional cycles required upon encountering a miss to fetch a block from the next level of memory hierarchy.

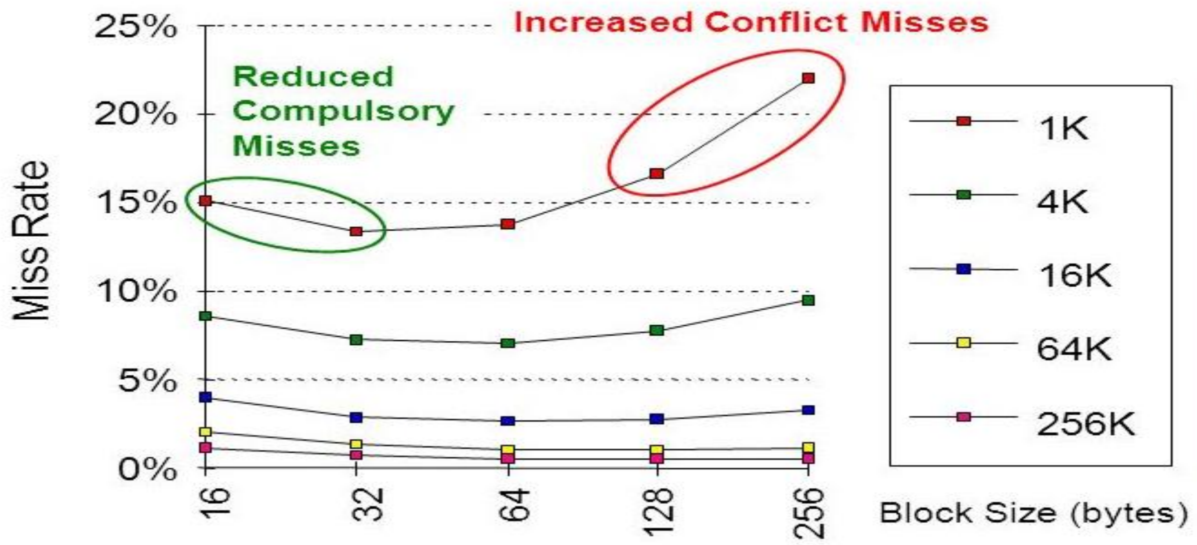
How to optimize cache ?

- Reduce Average Memory Access Time
- AMAT= Hit Time + Miss Rate x Miss Penalty
- Motives
  - Reducing the miss rate
  - Reducing the miss penalty
  - Reducing the hit time

Larger Block Size

- **Larger block size to reduce miss rate**
- **Advantages**
  - Utilize spatial locality
  - Reduces compulsory misses
- **Disadvantages**
  - Increases miss penalty
  - More time to fetch a block to the cache [bus width issue]
  - Increases conflict misses
  - More number of blocks will be mapped to the same location
  - May bring useless data and evict useful data [pollution]

Larger Block Size

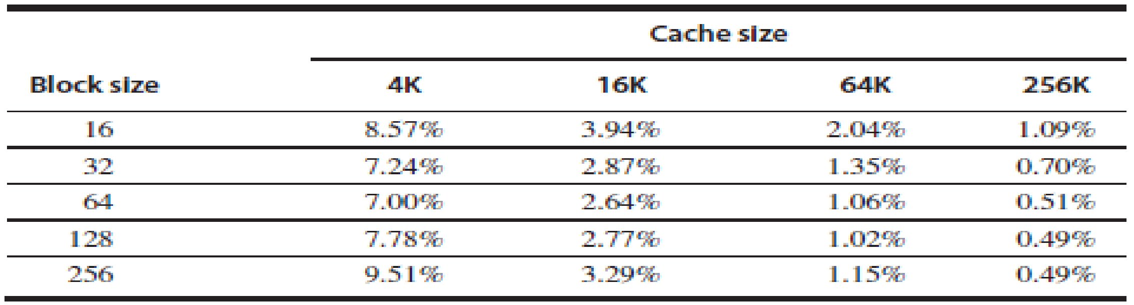


64-byte blocks are common in L1 caches

128-byte block are common in L2 caches

Larger Caches

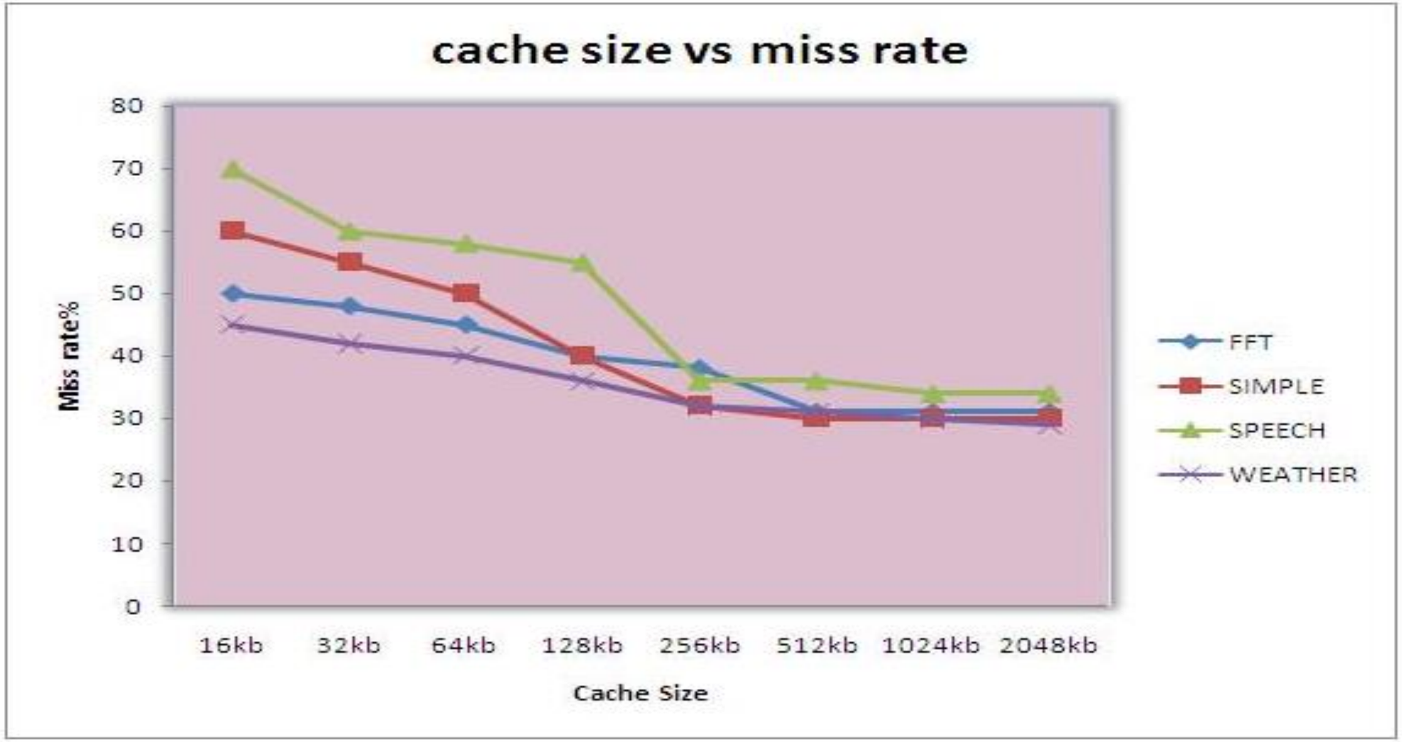
- ❖ Larger cache to reduce miss rate
- ❖ Advantages
  - ❖ Reduces capacity misses
  - ❖ Can accommodate larger memory footprint



	Cache size			
Block size	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

- ❖ Drawbacks
  - ❖ Longer hit time
  - ❖ Higher cost, area and power

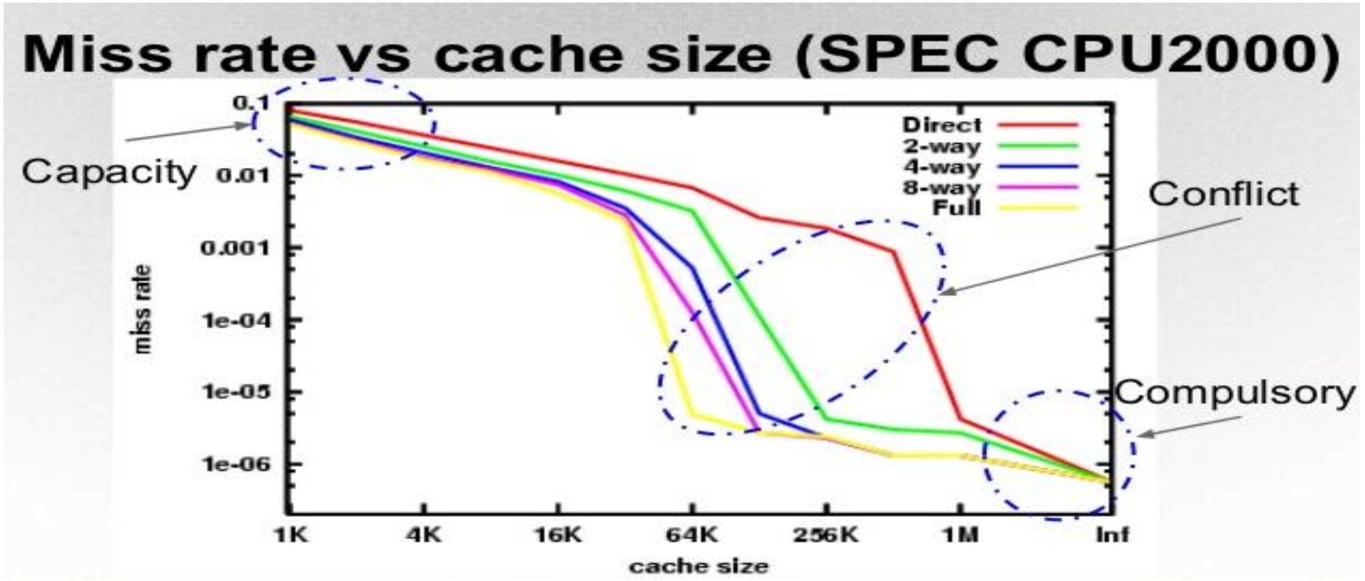
Larger Caches



Higher Associativity

- **Higher associativity to reduce miss rate**
  - Fully associative caches are the best, but high hit time.
  - So increase the associativity to an optimal possible level
- **Advantages**
  - Reduce conflict miss
  - Reduce miss rate and eviction rate
- **Drawbacks**
  - Increase in the hit time
  - Complex design than direct mapped
  - More time to search in the set (tag comparison time)

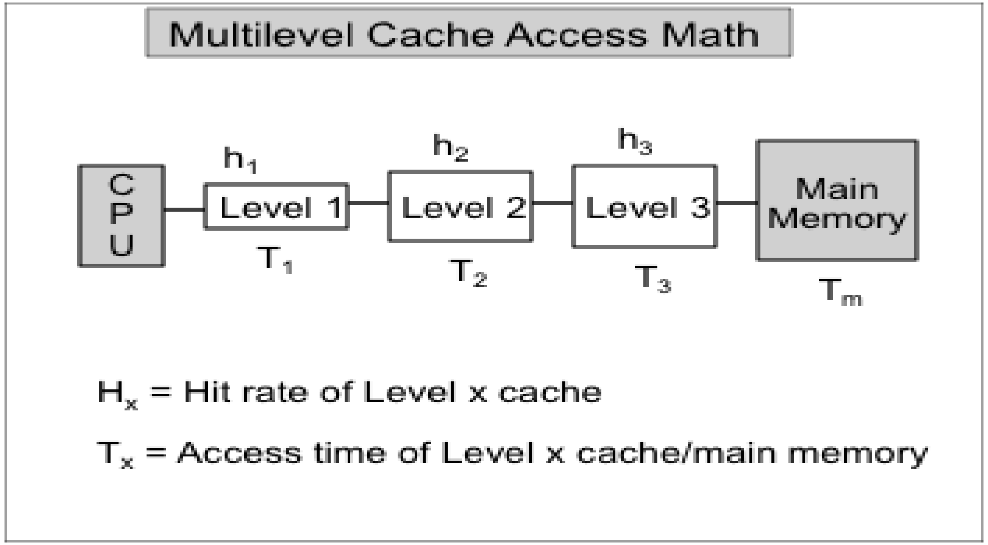
AMAT vs cache associativity



Multilevel caches

- **Multilevel caches to reduce miss penalty**
- **Caches should be faster** to keep pace with the speed of processors, **AND** **cache should be larger** to overcome the widening gap between the processor and main memory
- Add another level of cache between the cache and memory.
- The first-level cache (L1) can be small enough to match the clock cycle time of the fast processor. [Low hit time]
- The second-level cache (L2) can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty. [Low miss rate]

Multilevel caches


Multilevel Cache Access Math
CPU
Level 1
Level 2
Level 3
Main Memory
h_1
h_2
h_3
T_1
T_2
T_3
T_m
H_x = Hit rate of Level x cache
T_x = Access time of Level x cache/main memory

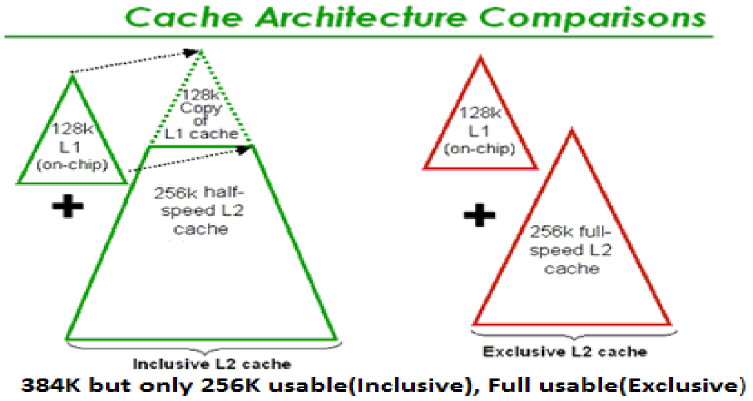
$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

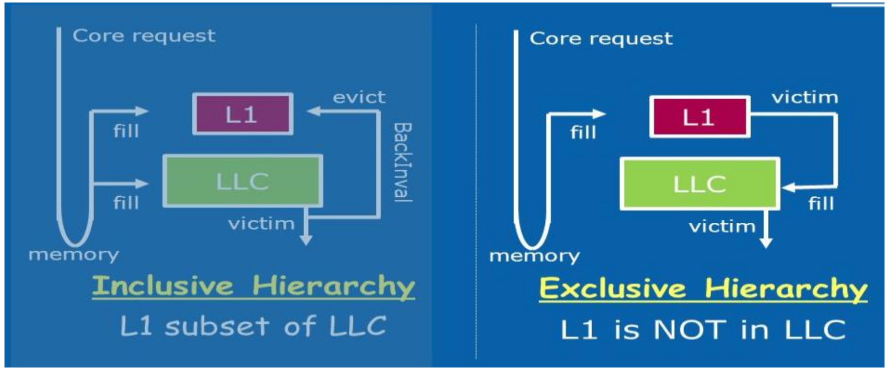
$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

Multilevel caches

- **Multilevel caches to reduce miss penalty**
- **Local miss rate:** Number of misses in a cache level divided by number of memory access to this level.
- **Global miss rate:** Number of misses in a cache level divided by number of memory access generated by the CPU.
- **Inclusive and Exclusive caches**





Prioritize read miss over writes

- **Prioritize read misses to reduce miss penalty**
- If a read miss has to evict a dirty memory block, the normal sequence is write the dirty block to memory and read the missed block

```
SW R3, 512(R0)      ;M[512] ← R3      (cache index 0)
LW R1, 1024(R0)     ;R1 ← M[1024]     (cache index 0)
LW R2, 512(R0)      ;R2 ← M[512]      (cache index 0)
```

- Optimization: copy the dirty block to a buffer, read from memory and then write the block - reduces CPU’s waiting time on read miss

Way Prediction

- **Predict the way in a set to reduce hit time**
- To improve hit time, predict the way to pre-set multiplexer
- Extra bits are set to predict the block with in the set
- Remember the MRU way of a given set
- Using the prediction bits, power gating can be done on unused ways for reducing power.

johnjose@iitg.ac.in

http://www.iitg.ac.in/johnjose/