National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 11: Cache II

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

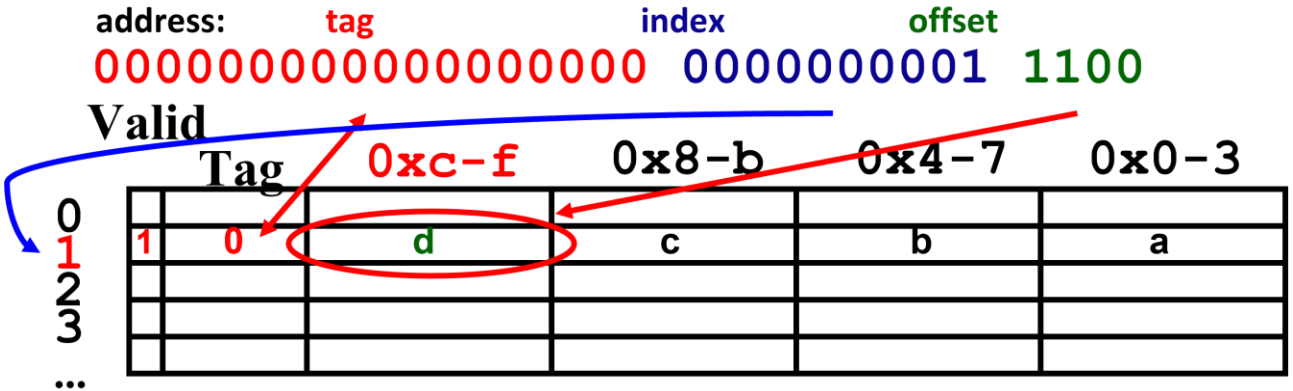
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS252 at ETHZ
    - https://safari.ethz.ch/digitaltechnik/spring2023
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/

Outline

- Associativity
- Fully Associative Cache
- N-way Set Associative Cache
- Cache Write Issue
- Cache Replacement Policy
- Multi-level Caches

Review: Cache Basics

- How to transparently move data among levels of a storage hierarchy
  - Address => index to set of candidates
  - Compare desired address with tag
  - Service hit or miss -> load new block and binding on miss

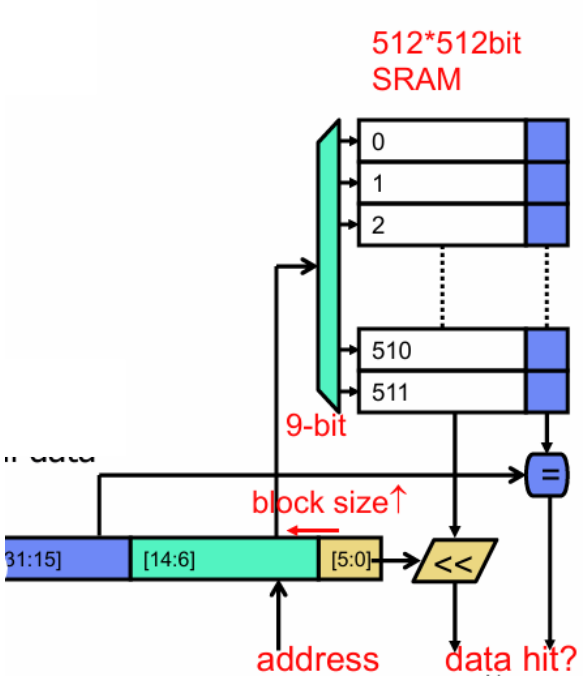


Review: Cache Basics

- 4-bit addresses -> 16 bytes memory
- 8 bytes cache, 2 bytes blocks
  - The number of sets: 4 (capacity / block size)
  - How address splits into offset/index/tag bits
    - Offset: least-significant $\log_2$(block size) = $\log_2(2)$ = 1-> 0000
    - Index: next $\log_2$(number-of-sets) = $\log_2(4)$ = 2 -> 0000
    - Tag: rest = 4 – 1 – 2 = 1-> 0000

Review: Cache Basics

- Given capacity, manipulate miss rate by changing cache organization
- One option: increase **block size**
  - Exploit spatial locality
  - Notice index/offset bits change
  - Tag remain the same
- Increasing cache block size
  - + reduce miss rate (up to a point)
  - + reduce tag overhead (why?)
  - - potentially useless data transfer



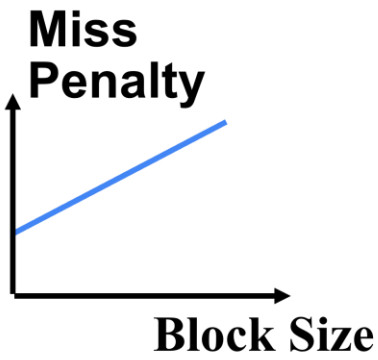
Review: Cache Basics

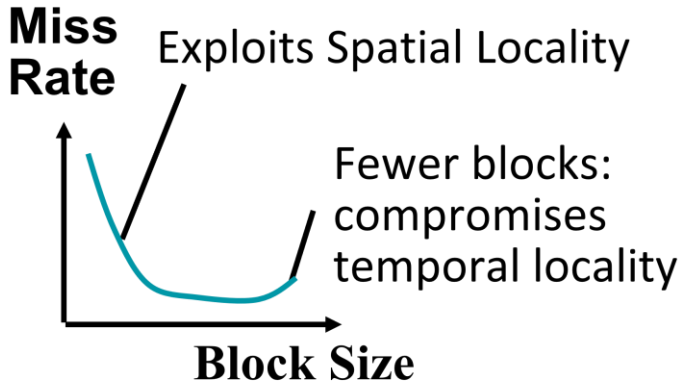
- **Hit Time**
  - Time to find and retrieve data from current level cache
- **Miss Penalty**
  - Average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- **Hit Rate**
  - % of requests that are found in current level cache
- **Miss Rate**
  - 1 - Hit Rate

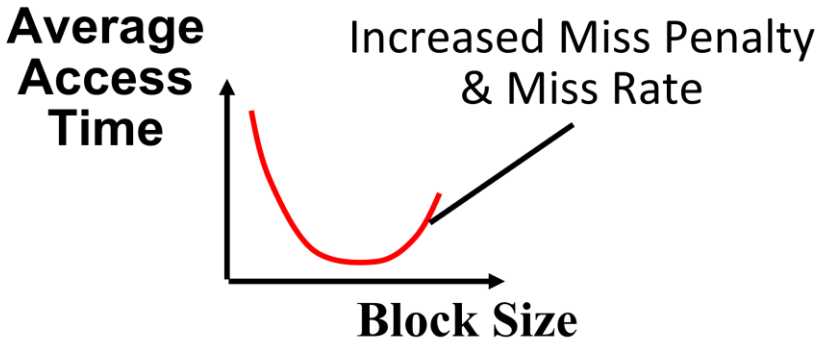
Review: Cache Basics

- Cache Size = 4 bytes, Block Size = 4 bytes
  - Only ONE entry (row) in the cache
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data before using it again
  - Nightmare for cache designer: **Ping Pong Effect**

Review: Cache Basics







Associativity

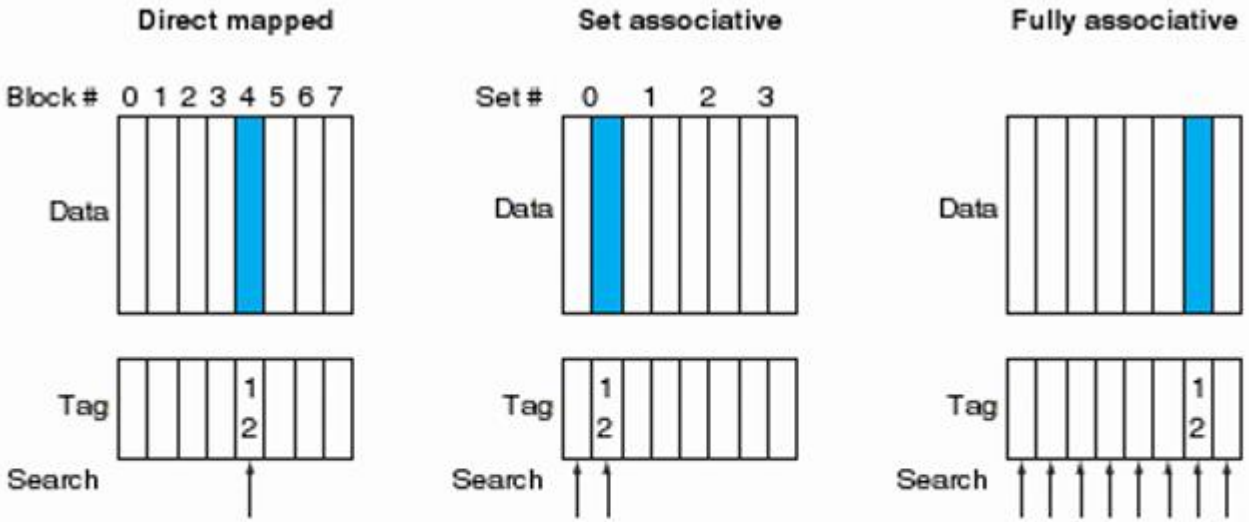


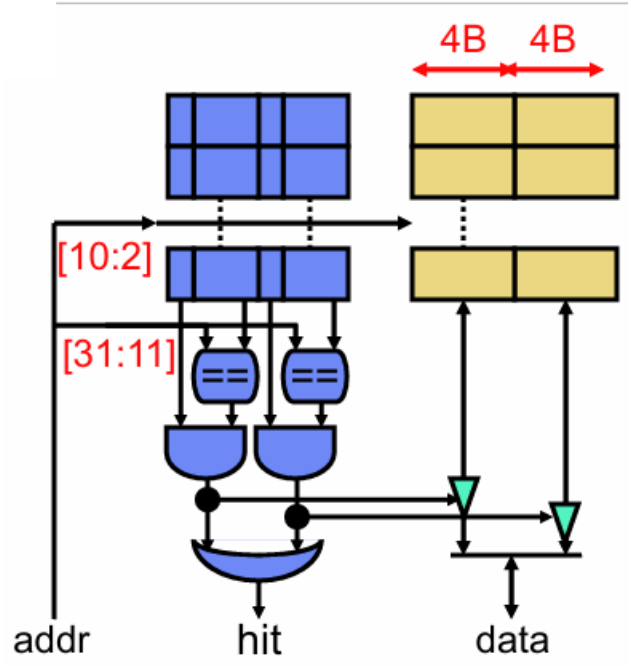
FIGURE 5.13 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped, set-associative, and fully associative placement. In direct-mapped placement, there is only one cache block where memory block 12 can be found, and that block is given by (12 modulo 8) = 4. In a two-way set-associative cache, there would be four sets, and memory block 12 must be in set (12 mod 4) = 0; the memory block could be in either element of the set. In a fully associative placement, the memory block for block address 12 can appear in any of the eight cache blocks.

Associativity

- **Direct-mapped cache**
  - Index completely specifies position which position a block can go in on a miss
- **N-Way Set associative**
  - Index specifies a set, but block can occupy any positions within the set on a miss
- **Fully associative**
  - Block can be written into any positions

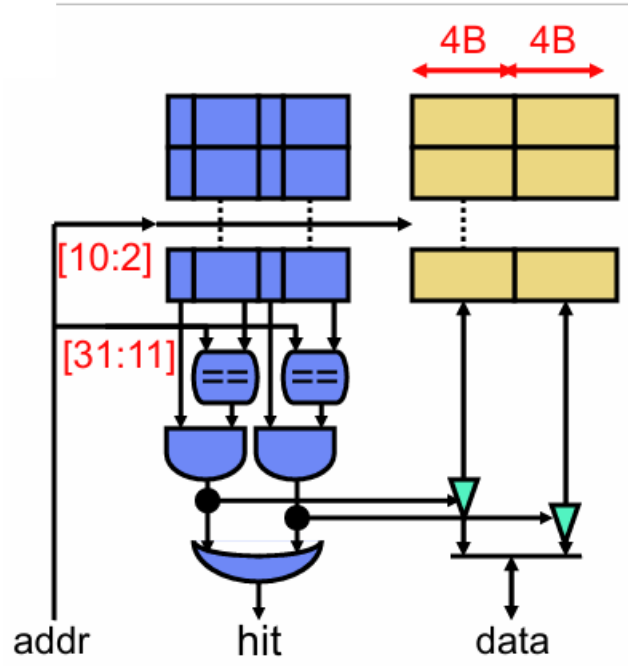
Associativity

- **Set-associativity**
  - Block can reside in one of few frames
  - Frame groups called sets
  - Each frame in set called a way
  - This is 2-way set-associative (SA)
  - 1-way -> directed-mapped (DM)
  - 1-set-> fully-associative (FA)
  - + Reduce conflicts
  - - Increase latency$_{hit}$
    - Additional tag match & muxing

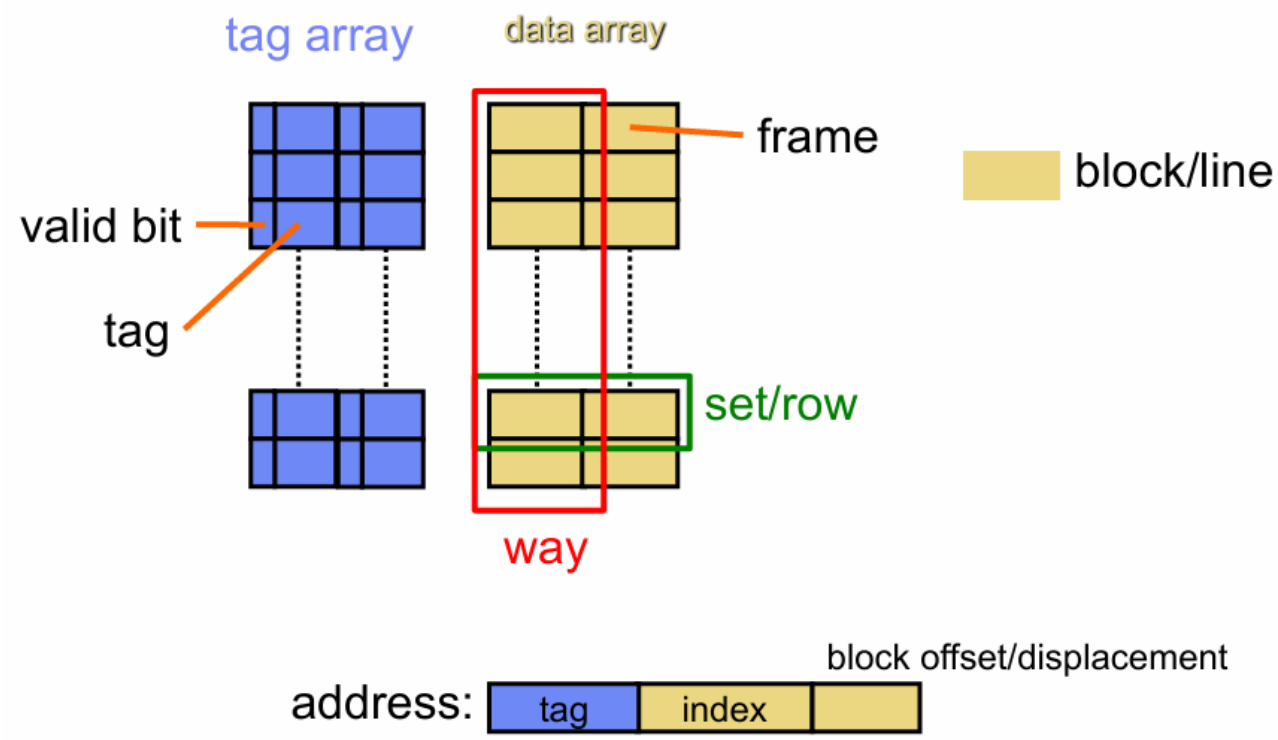


Associativity

- Lookup algorithm
  - Use index bits to find set
  - Read data/tags in all frames in parallel
  - Any (match and valid bit), Hit
  - Notice tag/index/offset bits
    - Only 9-bit index (versus 10-bit for direct mapped)

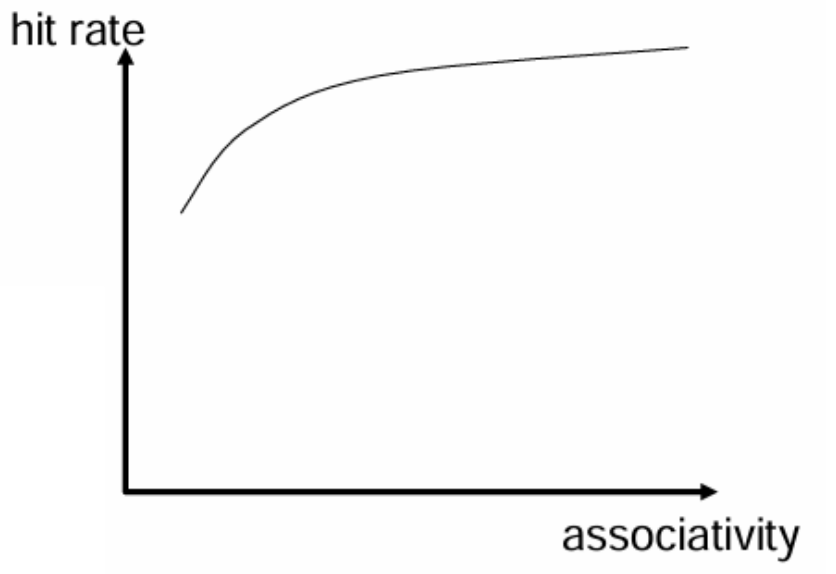


Associativity



Associativity

- How many blocks can be presented in the same index (i.e., set)?
- Larger associativity
  - Lower miss rate (reduced conflict)
  - Higher hit latency and area cost
- Smaller associativity
  - Lower cost
  - Lower hit latency
  - Especially important for L1 caches

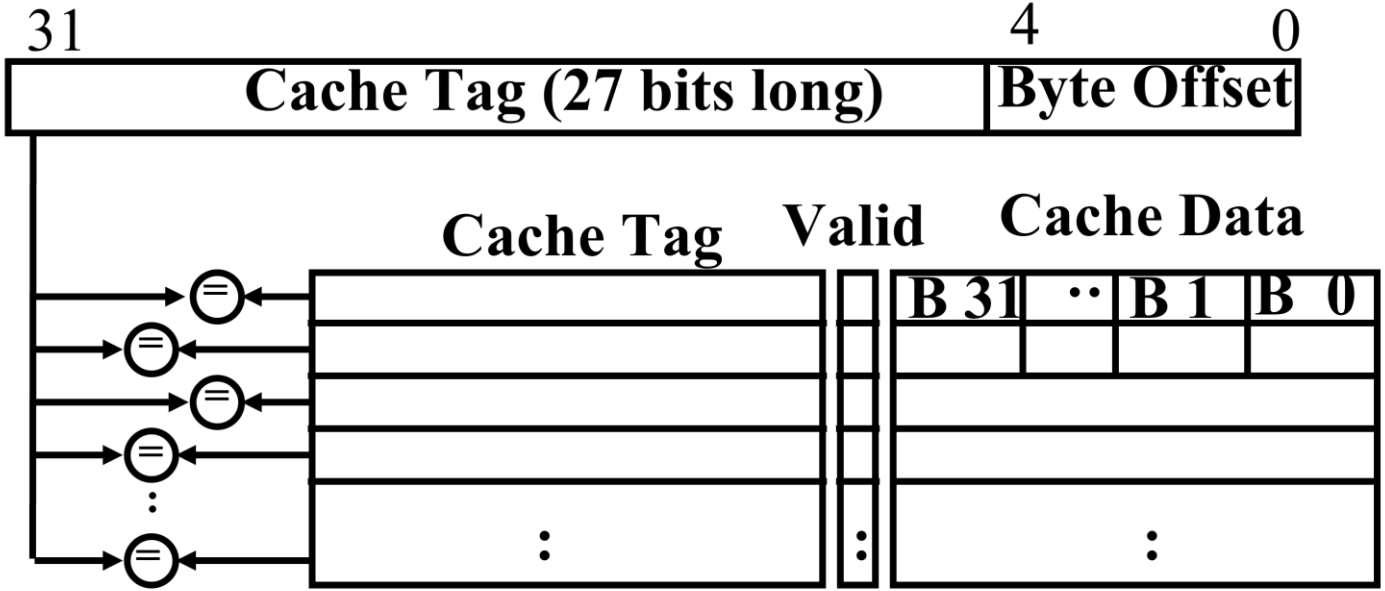


Fully Associative Cache

- Memory address fields
  - Tag: same as before
  - Offset: same as before
  - Index: Non-exist
- What does this mean?
  - No “rows”: any block can go anywhere in the cache
  - Must compare with all tags in entire cache to see if data is there

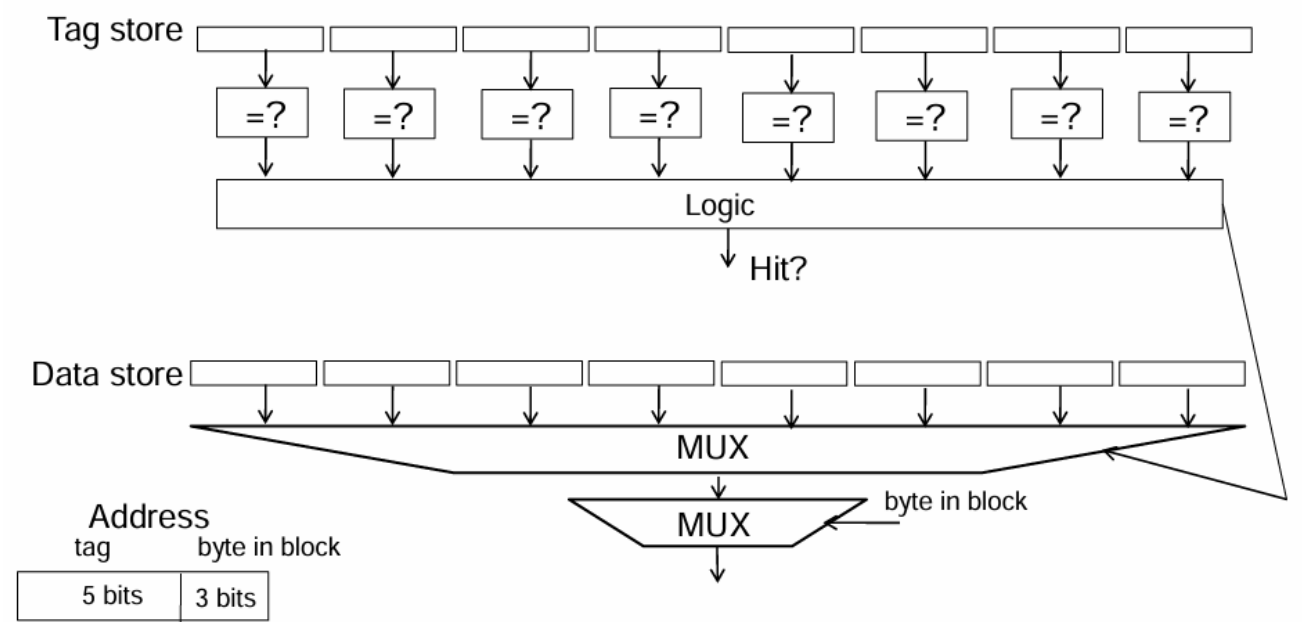
Fully Associative Cache

- Fully Associative Cache (e.g., 32 bytes block)
  - Compare tags in parallel



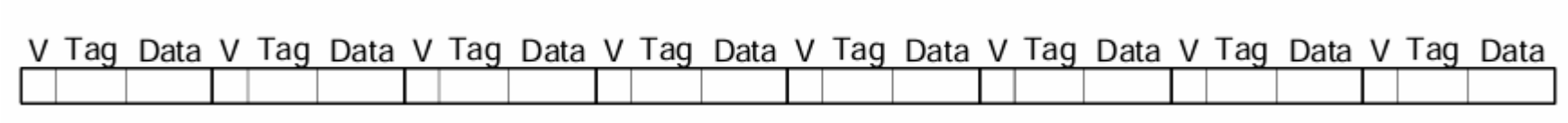
Fully Associative Cache

- Fully Associative Cache
  - A block can be placed in any cache location



Fully Associative Cache

- **Benefit of Fully Assoc Cache**
  - No conflict misses (since data can go anywhere)
- **Drawbacks of Fully Assoc Cache**
  - Need hardware comparator for every single entry
  - If we have a 64KB of data in cache with 4 bytes entries, we need 16K comparators
  - Expensive to build



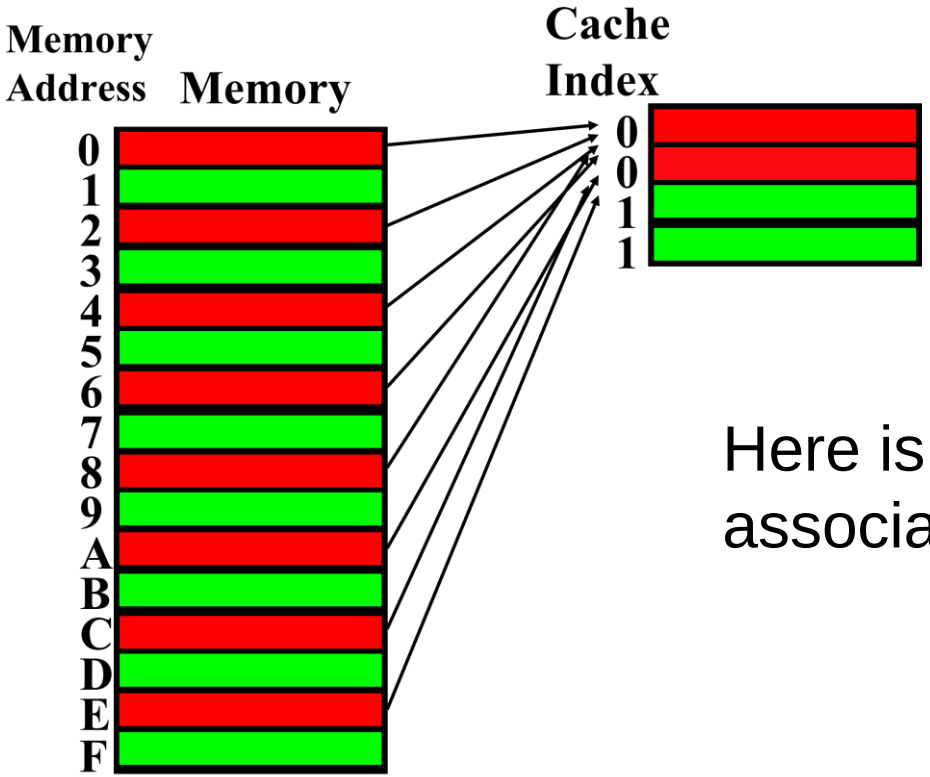
Types of Cache Misses

- **“Three Cs” Model of Misses**
- **3rd C: Capacity Misses**
  - Miss that occurs because the cache has a limited size
  - Miss that would not occur if we increase the size of the cache
  - This is the primary type of miss for Fully Associative cache

N-way Set Associative Cache

- Memory address fields:
  - **Tag**: same as before
  - **Offset**: same as before
  - **Index**: points us to the correct “row” (called a set in this case)
- What’s the difference?
  - Each set contains multiple blocks
  - Once we’ve found correct set, must compare with all tags in that set to find our data

N-way Set Associative Cache



Here is a simple 2-way set associative cache

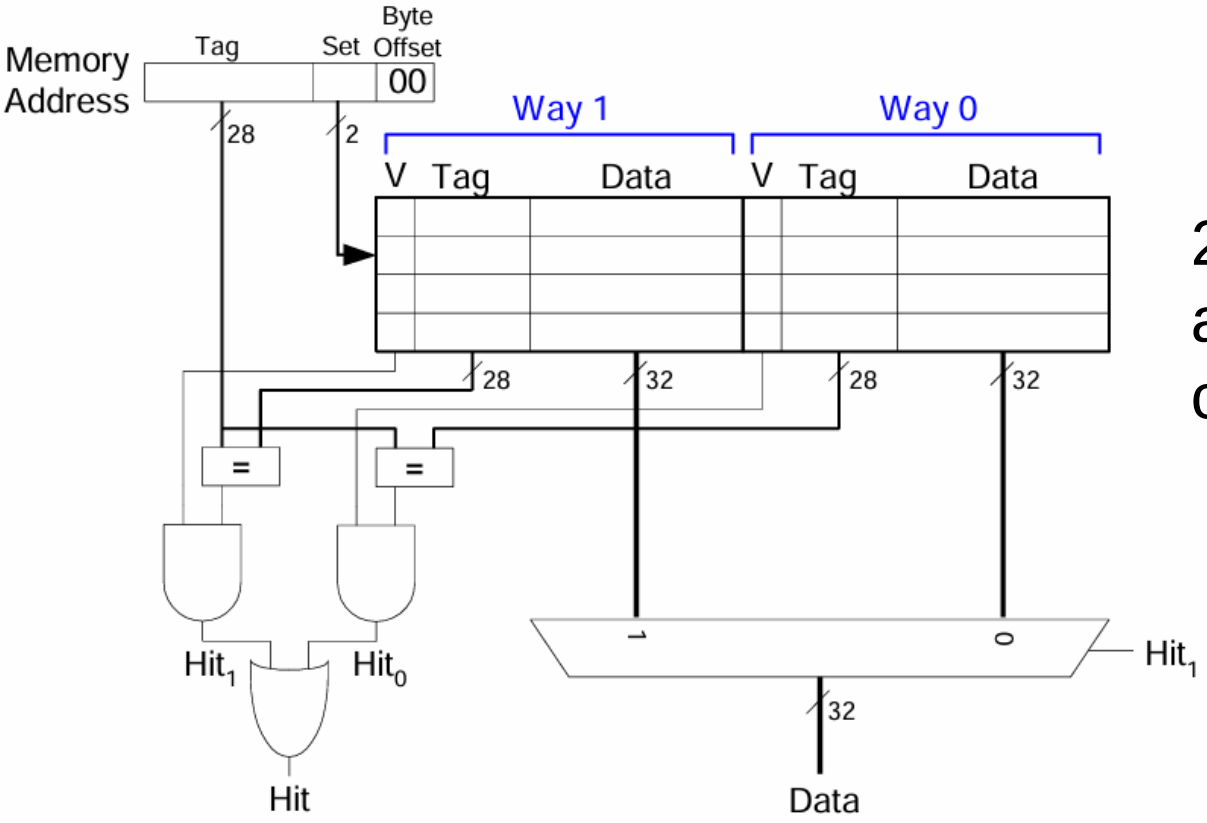
N-way Set Associative Cache

- **Basic idea**
  - Cache is directed-mapped w/respect to sets
  - Each set is fully associative with N blocks in it
- Given memory address
  - Find correct set using index value
  - Compare Tag with all Tag values in the determined set
  - If a match occurs, hit! Otherwise a miss
  - Finally, use the offset field as usual to find the desired data within the block

N-way Set Associative Cache

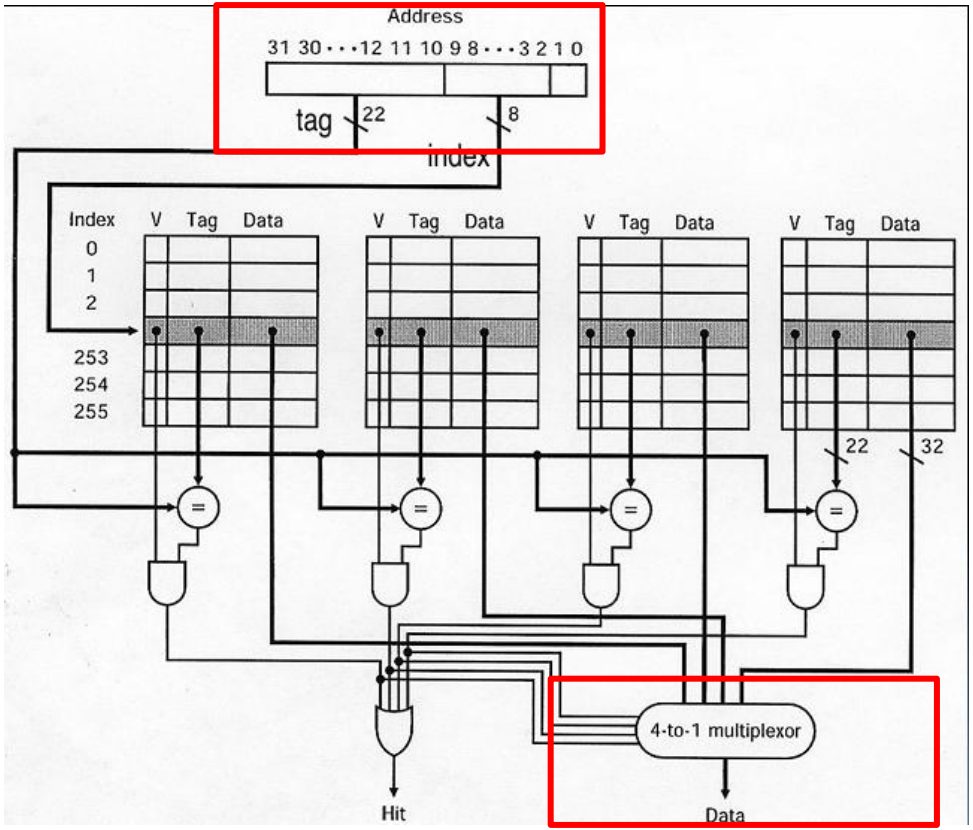
- What's so great about this?
  - Even a 2-way set associative cache avoids a lot of conflict misses
  - Hardware cost isn't that bad: only need N comparators
- In fact, for a cache with M blocks
  - It's Direct-Mapped if it's 1-way set associativity
  - It's Fully Associative if it's M-way set associativity
  - So these two are just special cases of the more general set associative design

N-way Set Associative Cache



2-way set associative cache

N-way Set Associative Cache



4-way set associative cache

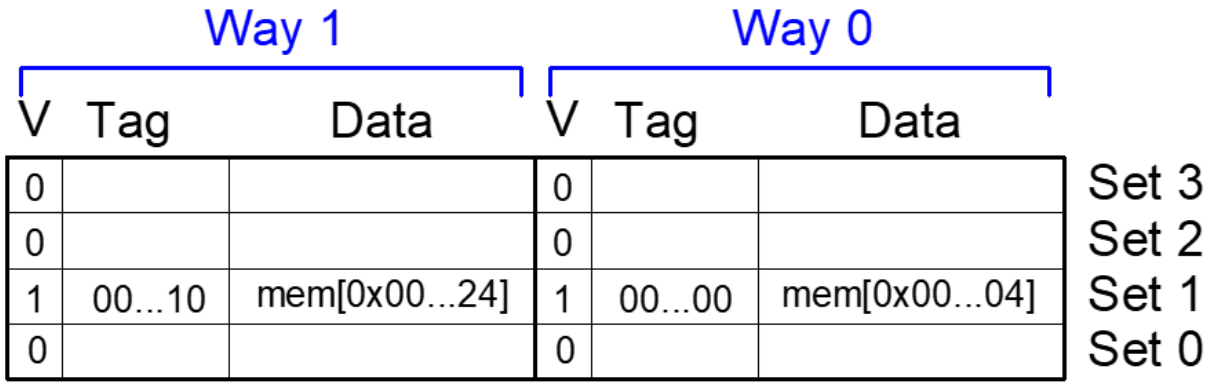
ways = index length / offset length

N-way Set Associative Performance

```
# RISC-V assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =



N-way Set Associative Performance

```
# RISC-V assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate = 2/10

= 20%

Associativity reduces conflict misses

	Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00…10	mem[0x00…24]	1	00…00	mem[0x00…04]	Set 1
0			0			Set 0

Cache Replacement Policy

- If we have the choice, where should we write an incoming block?
  - If there are any locations with valid bit off (empty), then usually write the new block into the first one
  - If all possible locations already have a valid block, we must pick a **replacement policy**:
    - Rule by which we determine which block gets “cached out” on a miss

Cache Replacement Policy

- On cache miss, which block in set to replace (kick out)?
  - If there are any locations with valid bit off (empty), then usually write the new block into the first one
  - If all possible locations already have a valid block, we must pick a **replacement policy**:
    - Rule by which we determine which block gets “cached out” on a miss

Cache Replacement Policy

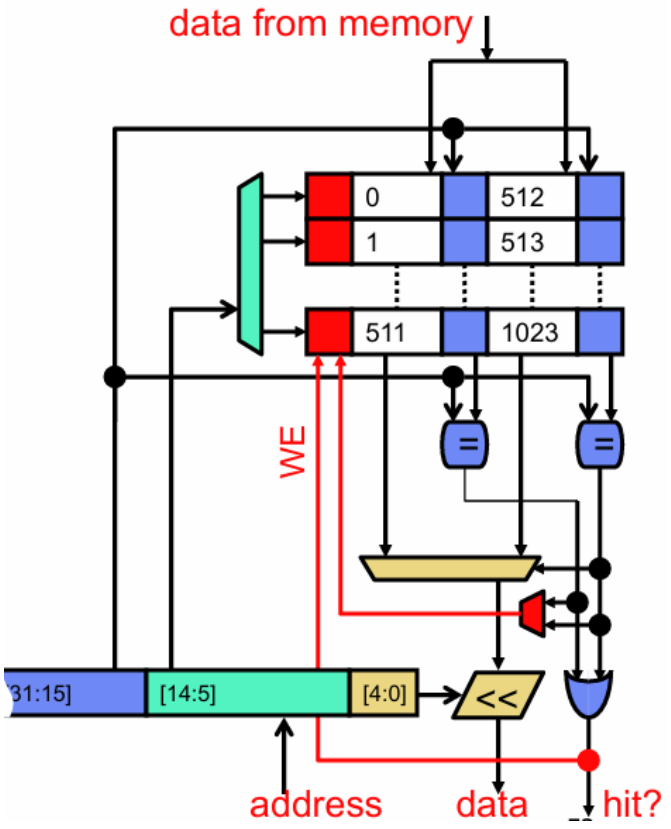
- **Block replacement options**
  - Random
  - FIFO (first-in first-out)
  - LRU (least recently used)
    - Fit with temporal locality, LRU = least likely to be used in future
  - NMRU (not most recently used)
    - Track which block in set is MRU
    - On replacement, pick a non-MRU block
    - One MRU pointer per set (vs. N LRU counters)

Cache Replacement Policy: LRU

- **LRU (Least Recently Used)**
  - **Idea:**
    - cache out block which has been accessed (read or write) least recently
  - **Pro:**
    - temporal locality => recent past use implies likely future use
  - **Con:**
    - with 2-way set assoc, easy to keep track (on LRU bit)
    - With 4-way or greater, requires complicated hardware and much time to keep track of this

Cache Replacement Policy: LRU

- Add LRU field to each set
  - LRU data is encoded “way”
  - Hit? Update MRU
  - LRU bits updated on each access



Cache Replacement Policy: LRU

- We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word access (ignore bytes for this problem)
  - 0, 2, 0, 1, 4, 0, 2, 3, 5, 4
- How many hits an how many misses will there be for the LRU cache replacement policy?

Cache Replacement Policy: LRU

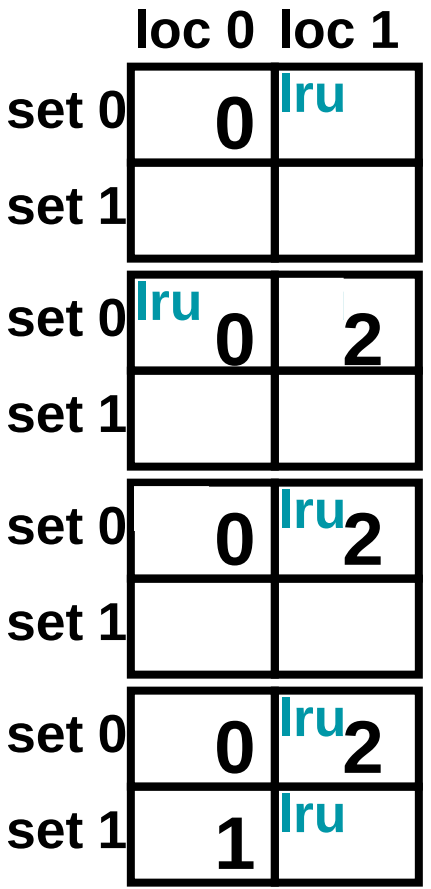
0: miss, bring into set 0 (loc 0)

2: miss, bring into set 0 (loc 1)

0: hit

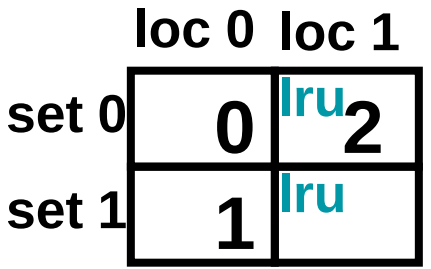
1: miss, bring into set 1 (loc 0)

Addresses 0, 2, 0, 1, 4, 0, ...

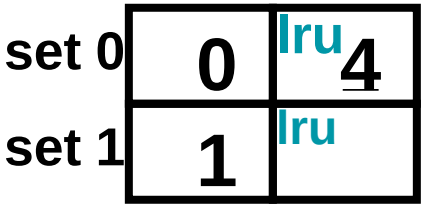


Cache Replacement Policy: LRU

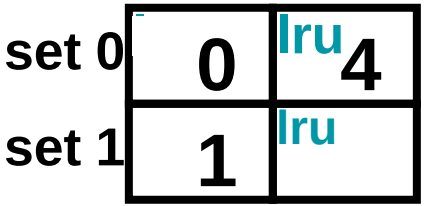
1: miss, bring into set 1 (loc 0)



4: miss, bring into set 0 (loc 1, replace 2)



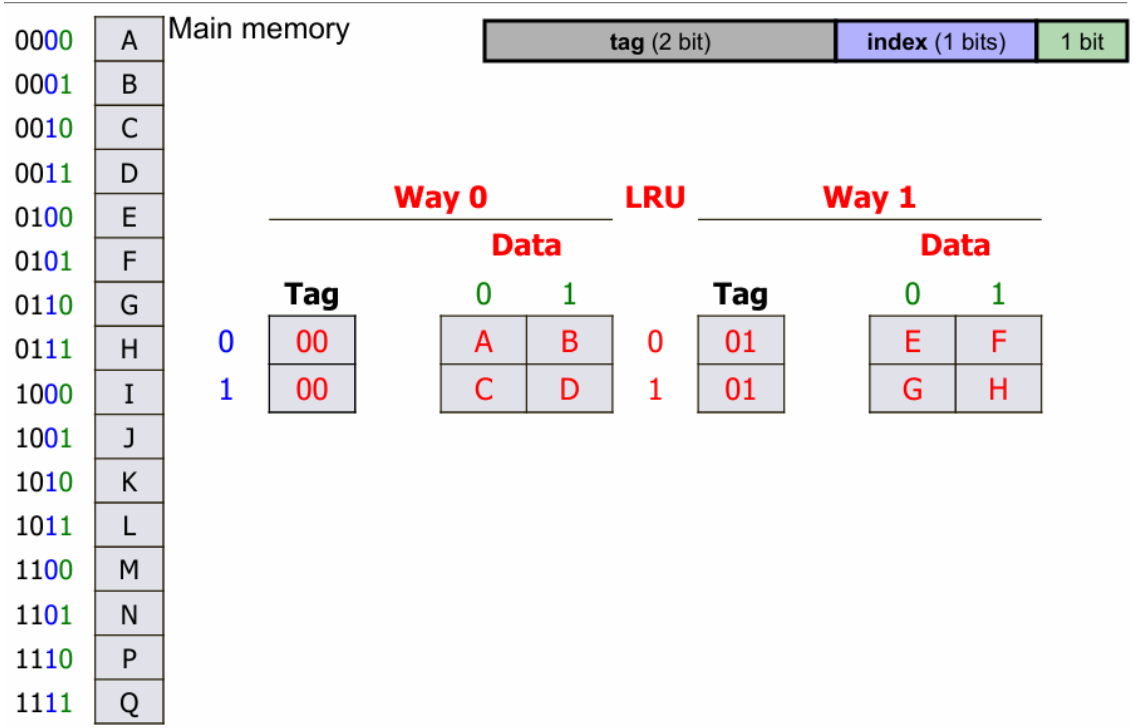
0: hit



Addresses 0, 2, 0, 1, 4, 0, ...

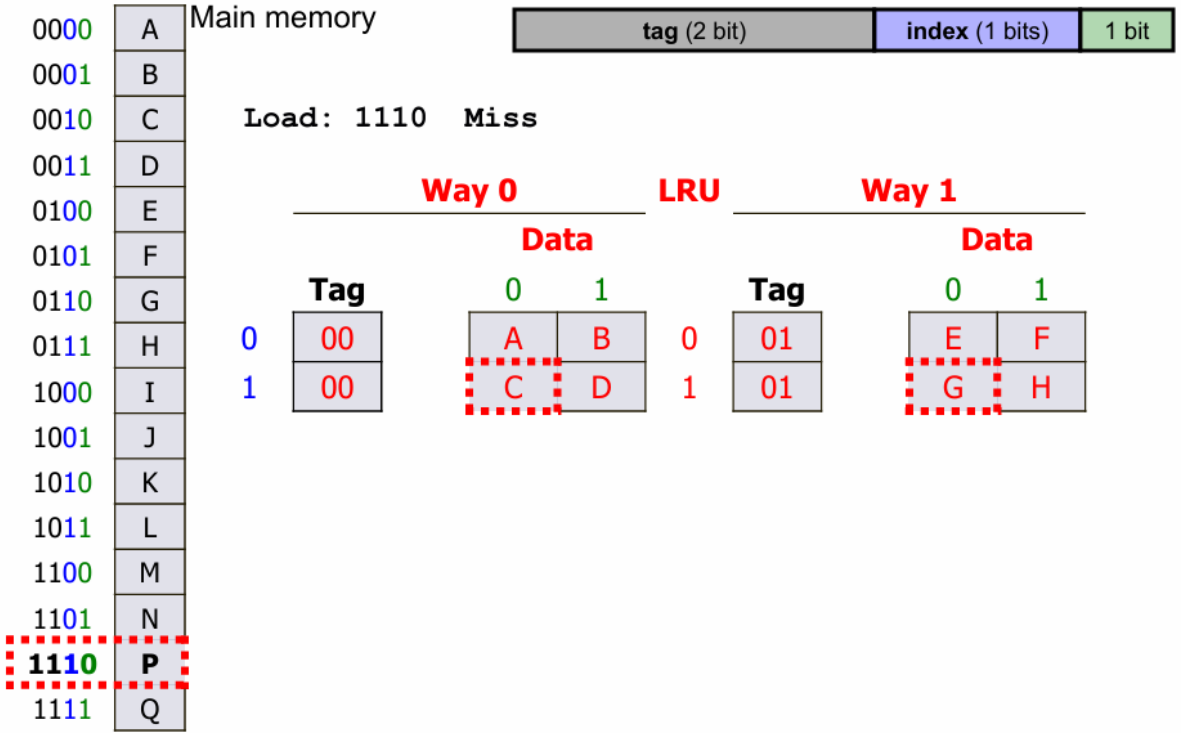
Cache Replacement Policy: LRU

- 4-bit address, 8B Cache, 2B Blocks, 2-way



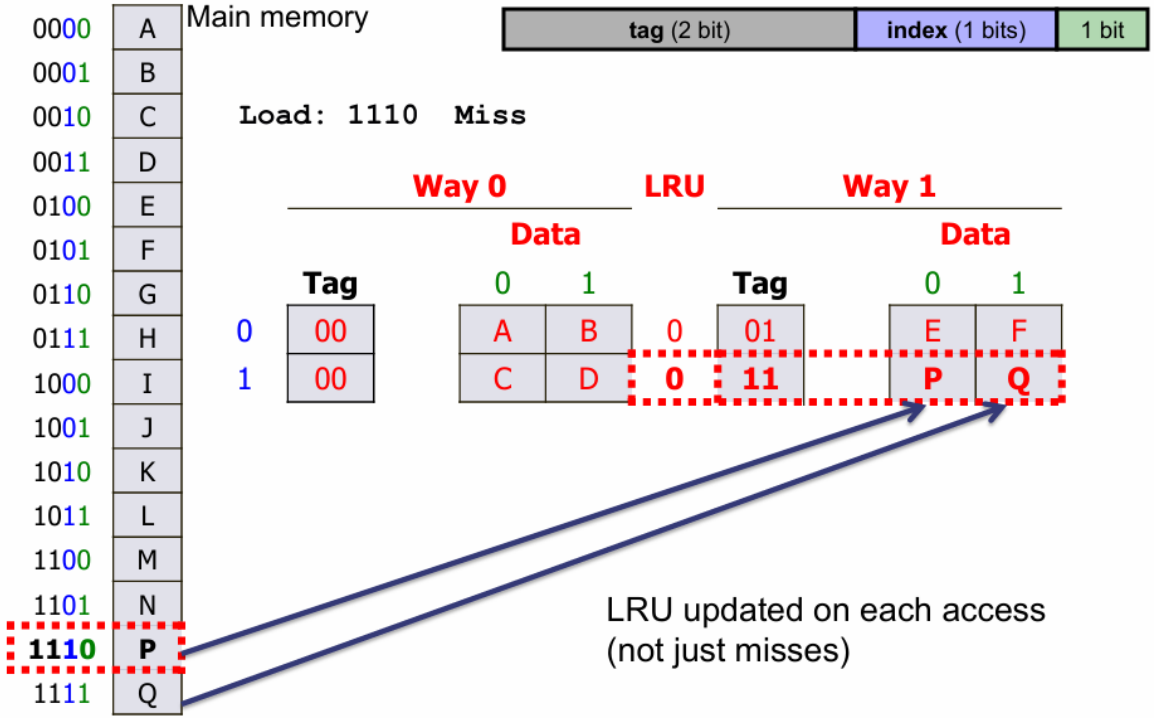
Cache Replacement Policy: LRU

- 4-bit address, 8B Cache, 2B Blocks, 2-way



Cache Replacement Policy: LRU

- 4-bit address, 8B Cache, 2B Blocks, 2-way



Cache Write Issue

- So far we have looked at reading from cache
  - Instruction fetches, loads
- What about writing into cache
  - Stores, not an issue for instruction caches
- Several new issues
  - Tag/data access
  - Write-through vs. write-back
  - Write-allocate vs. write-not-allocate
  - Hiding write miss latency

Cache Write Issue

- Tag/Data access
  - Reads: read tag and data in parallel
    - Tag mis-match -> data is wrong (OK, just stall until good data arrives)
  - Writes: read tag, write data in parallel? No. Why?
    - Tag mis-match -> clobbered data (oops!)
    - For associative caches, which way was written into?

Cache Write Issue

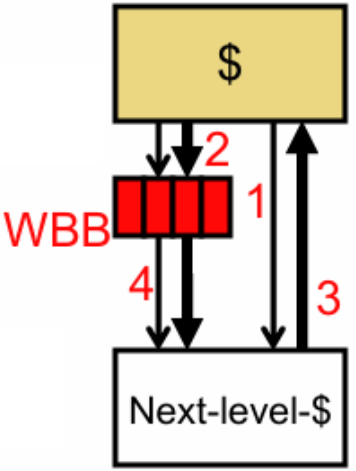
- Tag/Data access
  - Writes are a pipelined two step (multi-cycle) process
    - Step 1: match tag
    - Step 2: write to matching way
    - Bypass (with address check) to avoid load stalls
    - May introduce structural hazards

Cache Write Issue

- Write propagation: when to propagate new value to (lower level) memory?
  - **Option #1: Write-through**: immediately
    - On hit, update cache
    - Immediately send the write to the next level
  - **Option #2: Write-back:** when block is replaced
    - Requires additional "**dirty**" bit per block
    - Replace **clean** block: **no extra traffic**
    - Replace **dirty** block: **extra "writeback" of block**

Cache Write Issue

- **Option #2: Write-back:** when block is replaced
  - Writeback-buffer (WBB)
    - Hide latency of writeback (keep off critical path)
    - Step#1: Send “fill” request to next-level
    - Step#2: While waiting, write dirty block to buffer
    - Step#3: When new blocks arrives, put it into cache
    - Step#4: Write buffer contents to next-level



Cache Write Issue

- **Write-through**
  - - Requires additional bus bandwidth
    - Consider repeated write hits
  - - Next level must handle small writes (1, 2, 4, 8-bytes)
  - + No need for dirty bits in cache
  - + No need to handle “writeback” operations
    - Simplifies miss handling
- **Write-back**
  - + Key advantages: uses less bandwidth
  - Used in most CPU designs

Cache Write Issue

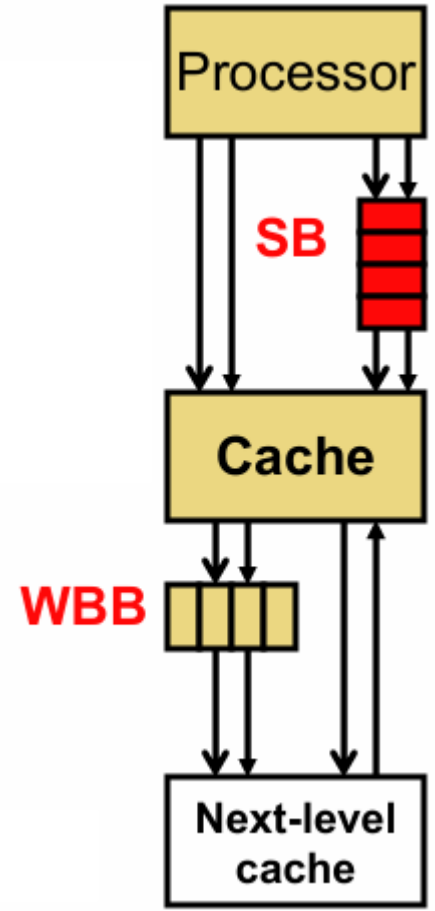
- **Write Miss Handling**
  - Write-allocate: fill block from next level, then write it
    - + Decreases read misses (next read to block will hit)
    - - Requires additional bandwidth
    - Commonly used (especially with write-back caches)
  - Write-non-allocate: just write to next level, no allocate
    - - Potentially more read misses
    - + Uses less bandwidth
    - Use with write-through

Cache Write Issue

- **Write Miss and Store Buffers**
  - Read miss?
    - Load can’t go on without the data
    - It must stall
  - Write miss?
    - No instruction is waiting for data
    - Why stall?

Cache Write Issue

- **Write Miss and Store Buffers**
  - Stores put address/value to store buffer, keep going
  - Store buffer writes stores to D$ in the background
  - Loads must search store buffer
  - + Eliminates stalls on write misses (mostly)
  - Store buffer vs. write-back buffer
    - **Store buffer**: in front of D$, for hiding store misses
    - **Write-buffer**: behind D$, for hiding writebacks



Classifying Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**
  - **(Coherence)**: miss due to external invalidations
    - Only in shared memory multiprocessors (later)

Miss Rate: ABC

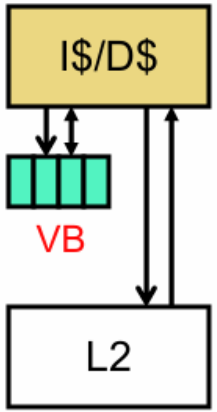
- Why do we care about 3C miss model?
  - So that we know what to do to eliminate misses
  - If you don't have conflict misses, increasing associativity won't help

- **More associativity** (assuming fixed capacity)
  - \+ Decreases conflict misses
  - – Increases $\text{latency}_{\text{hit}}$
- **Larger block size** (assuming fixed capacity)
  - – Increases conflict/capacity misses (fewer frames)
  - \+ Decreases compulsory misses (spatial locality)
  - No significant effect on $\text{latency}_{\text{hit}}$
- **More capacity**
  - \+ Decreases capacity misses
  - – Increases $\text{latency}_{\text{hit}}$

Reducing Conflict Misses: Victim Buffer

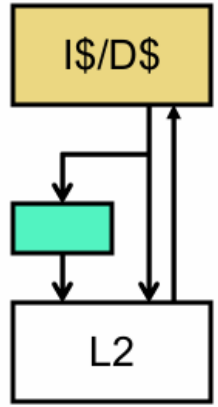
- Conflict misses: not enough associativity
  - High associativity is expensive, but also rarely needed
    - 3 blocks mapping to same 2-way set and accessed (XYZ)+

- **Victim buffer (VB)**: small fully-associative cache
  - Sits on I$/D$ miss path
  - Small (e.g., 8 entries) so very fast
  - Blocks kicked out of I$/D$ placed in VB
  - On miss, check VB: hit? Place block back in I$/D$
  - 8 extra ways, shared among all sets
    + Only a few sets will need it at any given time
  + Very effective in practice



Prefetching

- Bring data into cache proactively/**speculatively**
  - If successful, reduces number of caches misses
- Key: anticipate upcoming miss addresses accurately
  - Can do in software or hardware
- Simple hardware prefetching: **next block prefetching**
  - Miss on address **X** → anticipate miss on **X+block-size**
  - \+ Works for insns: sequential execution
  - \+ Works for data: arrays
- Table-driven hardware prefetching
  - Use **predictor** to detect strides, common patterns
- Effectiveness determined by:
  - **Timeliness**: initiate prefetches sufficiently in advance
  - **Coverage**: prefetch for as many misses as possible
  - **Accuracy**: don't pollute with unnecessary data



Software Prefetching

- Use a special “prefetch” instruction
  - Tells the hardware to bring in data
  - Just a hint
- Inserted by programmer or compiler
- Example

```
int tree_add(tree_t* t) {
  if (t == NULL) return 0;
  __builtin_prefetch(t->left);
  __builtin_prefetch(t->right);
  return t->val + tree_add(t->right) + tree_add(t->left);
}
```

- Multiple prefetches bring multiple blocks in parallel
  - More “Memory-level” parallelism (MLP)

Software Restructuring: Data

- Capacity misses: poor spatial or temporal locality
  - Several code restructuring techniques to improve both
  - – Compiler must know that restructuring preserves semantics

- **Loop interchange**: spatial locality
  - Example: row-major matrix: `X[i][j]` followed by `X[i][j+1]`
  - Poor code: `X[i][j]` followed by `X[i+1][j]`

```
for (j = 0; j<NCOLS; j++)
   for (i = 0; i<NROWS; i++)
      sum += X[i][j];
```

  - Better code

```
for (i = 0; i<NROWS; i++)
   for (j = 0; j<NCOLS; j++)
      sum += X[i][j];
```

Software Restructuring: Data

- **Loop blocking**: temporal locality
  - Poor code

```
for (k=0; k<NUM_ITERATIONS; k++)
    for (i=0; i<NUM_ELEMS; i++)
        X[i] = f(X[i]);     // say
```

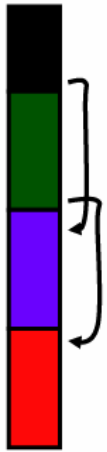
  - Better code
    - Cut array into CACHE_SIZE chunks
    - Run all phases on one chunk, proceed to next chunk

```
for (i=0; i<NUM_ELEMS; i+=CACHE_SIZE)
    for (k=0; k<NUM_ITERATIONS; k++)
        for (j=0; j<CACHE_SIZE; j++)
            X[i+j] = f(X[i+j]);
```

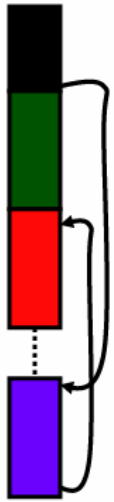
  - Assumes you know **CACHE_SIZE**, do you?

Software Restructuring: Code

- Compiler can lay out code for temporal and spatial locality
  - If (a) { **code1;** } else { **code2;** } **code3;**
  - But, **code2** case never happens (say, error condition)



+ Better locality
+ Fewer taken branches



+ Better locality for code after **code3**
+ Fewer taken branches

Multi-level Caches

- T_{access} vs. $\%_{miss}$ tradeoff
- Upper memory components (I$, D$) emphasize low t_{access}
  - Frequent access -> t_{access} important
  - T_{miss} is not bad -> $\%_{miss}$ less important
  - Lower capacity and lower associativity (to reduce t_{access})
  - Small-medium block-size (to reduce conflicts)
- Moving down (L2, L3) emphasis turns to $\%_{miss}$
  - T_{miss} is bad -> $\%_{miss}$ important
  - High capacity, associativity, and block size (to reduce $\%_{miss}$)

Multi-level Caches

- Memory hierarchy parameters

Parameter	I$/D$	L2	L3	Main Memory
t_{access}	2ns	10ns	30ns	100ns
t_{miss}	**10ns**	**30ns**	**100ns**	**10ms (10M ns)**
Capacity	8KB–64KB	256KB–8MB	2–16MB	1-4GBs
Block size	16B–64B	32B–128B	32B-256B	NA
Associativity	2-8	4–16	4-16	NA

Multi-level Caches

- **Split vs. unified caches**
- Split I$/D$: instruction and data in different caches
  - To minimize structural hazards and t_{access}
  - Larger unified I$/D$ would be slow, 2nd port even slower
  - Optimize I$ and D$ separately
    - Not write for I$, smaller reads for D$

Multi-level Caches

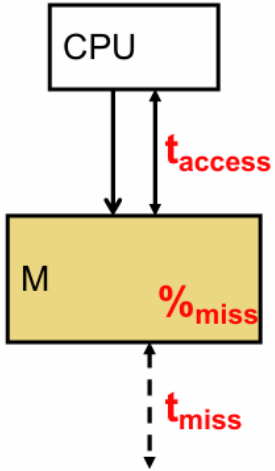
- **Split vs. unified caches**
- Unified L2, L3: instruction and data together
  - To minimize $\%_{miss}$
  - + Fewer capacity misses: unused instruction capacity can be used for data
  - - More conflict misses: instruction/data conflict
    - A much smaller effect in large caches
  - Instruction/data structural hazards are rare: simultaneous I$/D$ miss

Multi-level Caches

- **Inclusion**
  - Bring block from memory into L2 then L1
    - A block in the L1 is always in the L2
  - If block evicted from L2, must also evict it from L1
    - Why? more on this when we talk about multicore
- **Exclusion**
  - Bring block from memory into L1 but not L2
    - Move block to L2 on L1 eviction
      - L2 becomes a large victim cache
    - Block is either in L1 or L2 (never both)
  - Good if L2 is small relative to L1
    - Example: AMD's Duron 64KB L1s, 64KB L2

Multi-level Caches

- **Memory performance equation**



- For memory component M
  - **Access**: read or write to M
  - **Hit**: desired data found in M
  - **Miss**: desired data not found in M
    - Must get from another (slower) component
  - **Fill**: action of placing data in M
  - $\%_{miss}$ (miss-rate): #misses / #accesses
  - t_{access}: time to read data from (write data to) M
  - t_{miss}: time to read data into M
- Performance metric
  - t_{avg}: average access time

$$t_{avg} = t_{access} + (\%_{miss} * t_{miss})$$

Takeaway Question

- Parameters
  - Baseline pipeline CPI = 1
  - 30% of instructions are memory operations
  - L1: t_{access} = 1 cycle (included in CPI of 1), $\%_{miss}$ = 5% of accesses
  - L2: t_{access} = 10 cycle, $\%_{miss}$ = 20% of L2 accesses
  - DRAM: t_{access} = 50 cycle
  - What is the new CPI?

Takeaway Question

- Parameters
  - 30% of instructions are memory operations
  - L1: t_{access} = 1 cycle (included in CPI of 1), $\%_{miss}$ = 5% of accesses
  - L2: t_{access} = 10 cycle, $\%_{miss}$ = 20% of L2 accesses
  - DRAM: t_{access} = 50 cycle
  - What is the new CPI?
    - CPI = 1 + 30% * 5% * $t_{missD\$}$
    - $t_{missD\$} = t_{avgL2} = t_{accL2} + (\%_{missL2} * t_{accMem})$ = 10 + (20%*50) = 20 cycles
    - Thus, CPI = 1 + 30% *5% * 20 = 1.3 CPI

Conclusion

- **Memory hierarchy**
  - Cache (SRAM) -> Memory (DRAM) -> swap (Disk)
  - Smaller, faster, more expensive
- **Cache ABCs (capacity, associativity, block size)**
  - 3C miss model: compulsory, capacity, conflict
- **Write issues**
  - Write-back vs. write-through/write-allocate vs. write-no-allocate