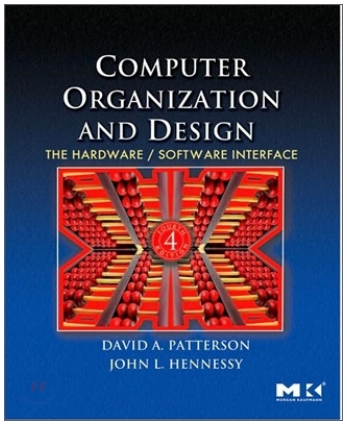
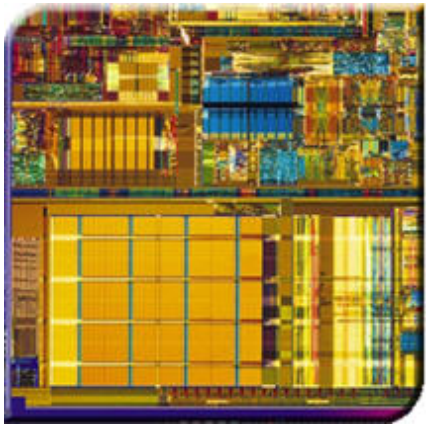
Computer Architecture

Lecture 8 Memory Subsystem



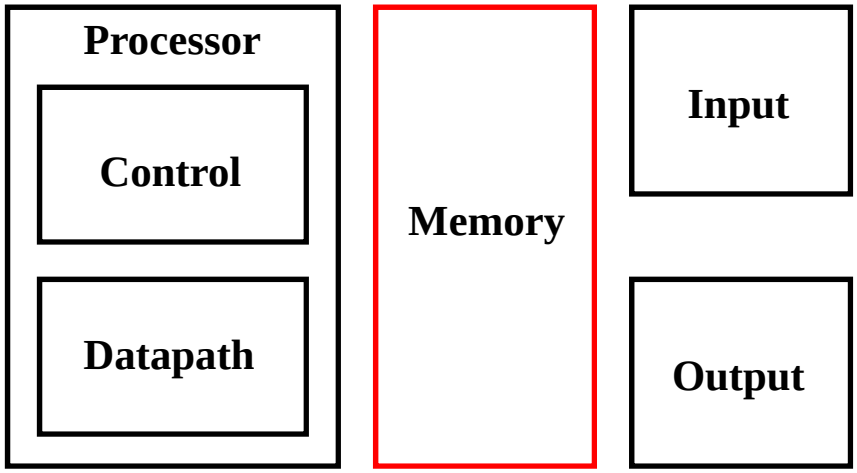


Prof. Jongmyon Kim

울산대학교
UNIVERSITY OF ULSAN

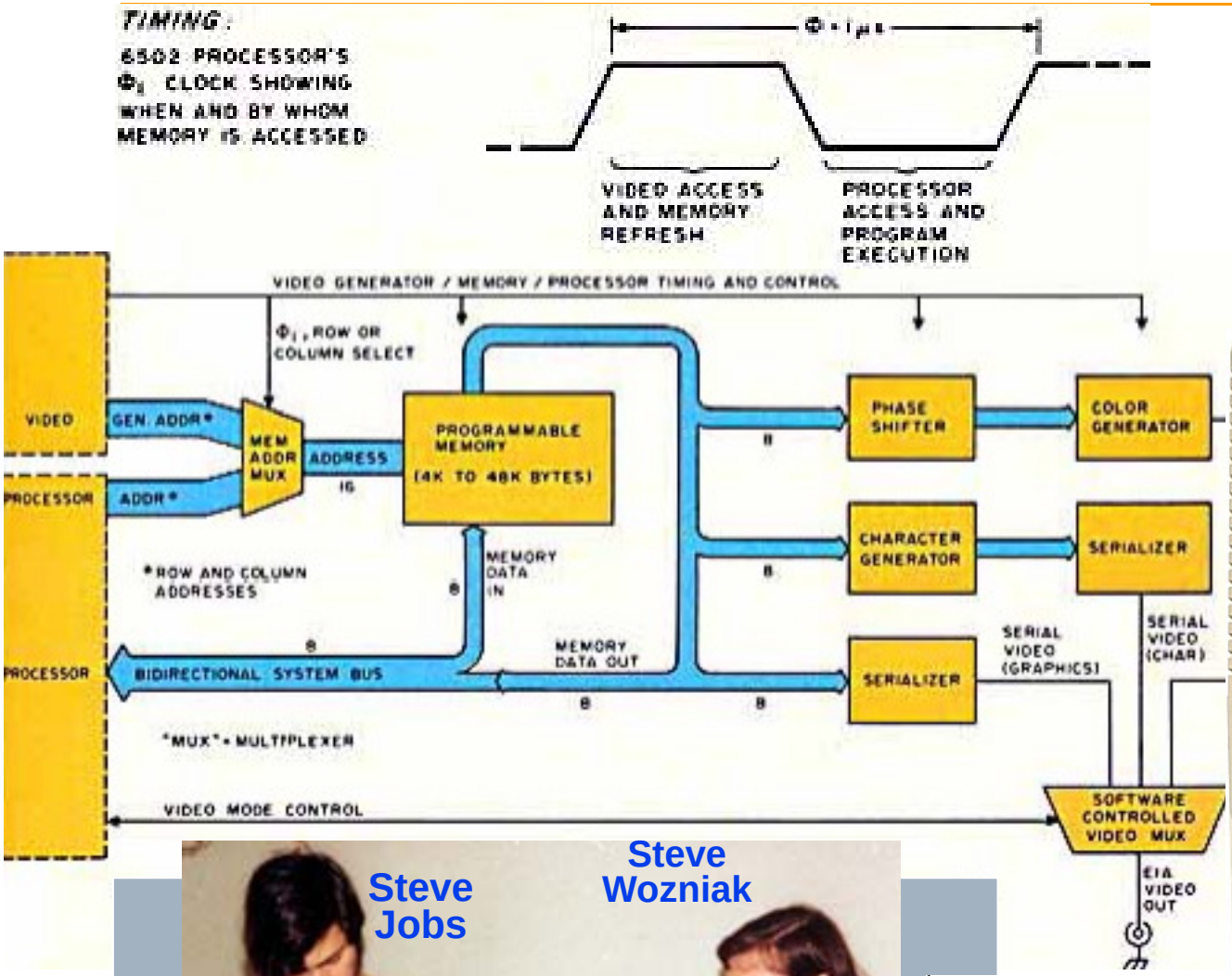
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer



- This lecture (and next few): Memory System

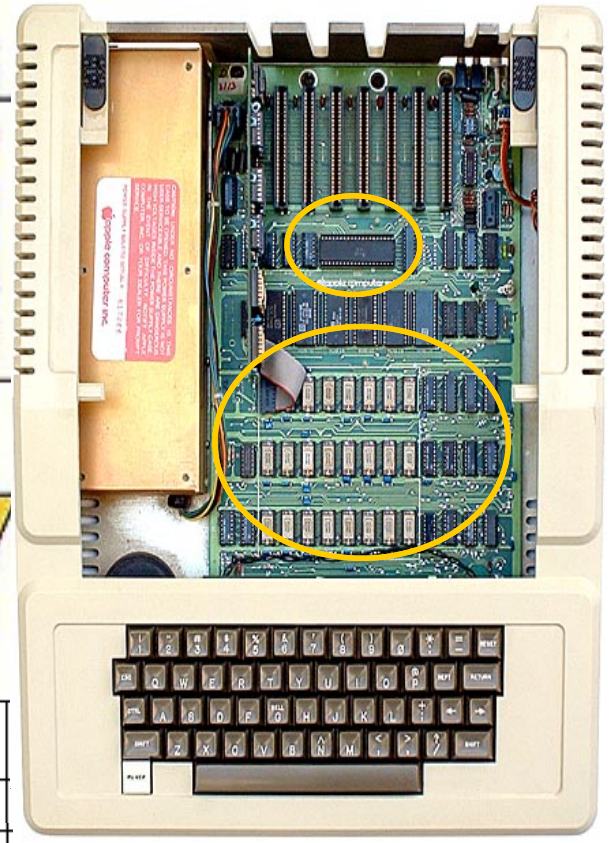
1977: DRAM faster than microprocessors

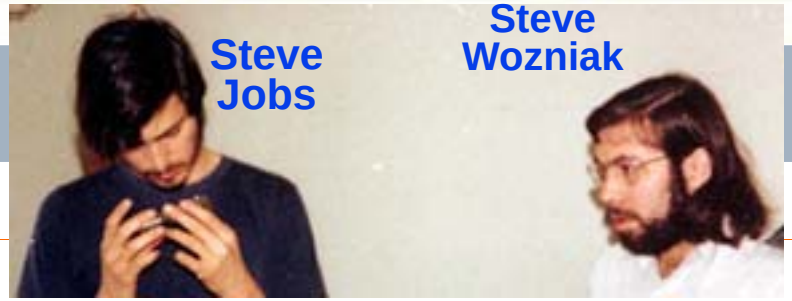


Apple][(1977)

CPU: 1000 ns

DRAM: 400 ns



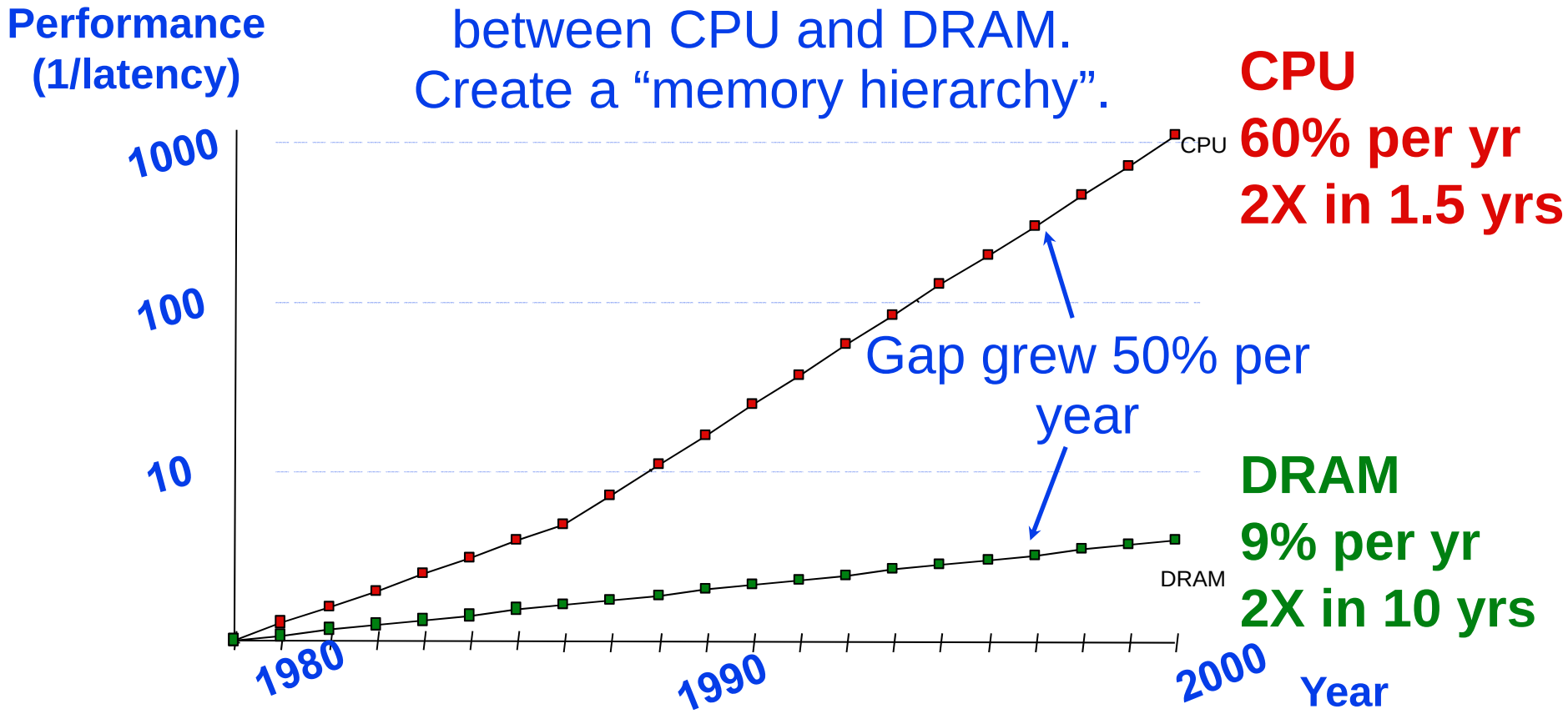


RAM Complement	Apple II System
4K	$ 1,298.00
48K	2,638.00

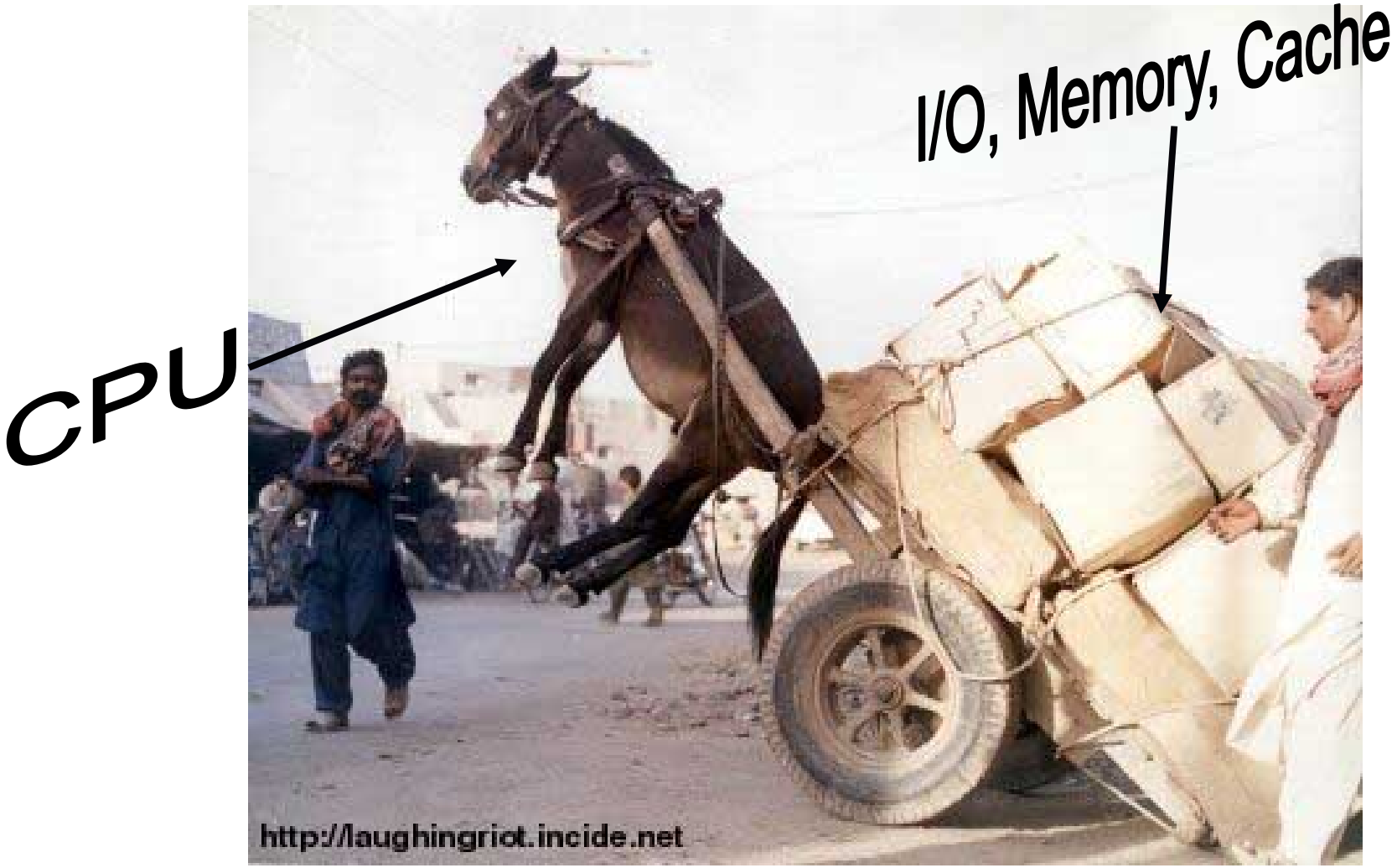
Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?

A. Put smaller, faster “cache” memories between CPU and DRAM. Create a “memory hierarchy”.



An Unbalanced System



Source: Bob Colwell keynote ISCA'29 2002

Performance matters

- Consider basic 5 stage pipeline design
  - CPU runs at 1ns cycle time (1GHz)
  - Main memory runs at 100ns access time
- How is performance?
  - Fetch A (100 cycles)
  - Decode A (1 cycle) + Fetch B (100 cycles)
  - Ex A (1 cy) + Dec B (1 cy) + Fetch C (100 cycles)
    - Effective 1 instr per 100 cycles --> 10MHz CPU
  - Latency killing system
- Problem only getting worse
  - CPU speeds grow much faster than DRAM speeds

Memory Issues

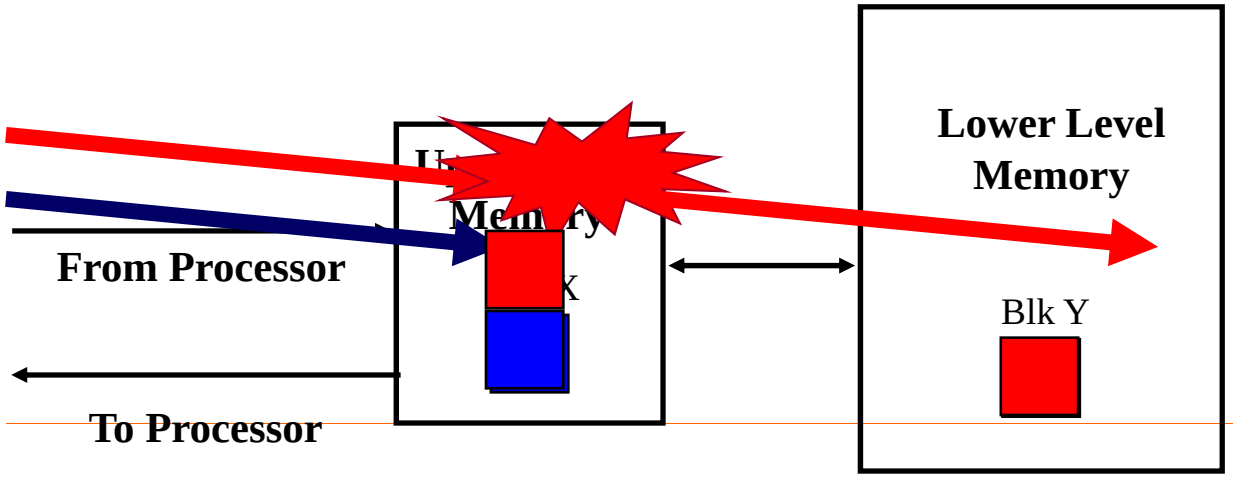
- Latency
  - Time to move through the longest circuit path (from the start of request to the response)
- Bandwidth
  - Number of bits transported at one time
- Capacity
  - Size of memory
- Energy
  - Cost of accessing memory (to read and write)

Typical Solution

- With the large gap in CPU vs DRAM speeds, must have solution
  - Cache (“slave memories”)
  - Ultra-fast, small local memory
  - Cache runs at 1ns access time...
    - Fetch A (1 cycle)
    - Decode A (1 cycle) + Fetch B (1 cycle)
    - Ex A (1 cy) + Dec B (1 cy) + Fetch C (1 cy)
      - Effective 1 instr per 1 cyc ----> 1GHz CPU
- Special terms
  - hit
  - miss
  - rates (hit/miss)

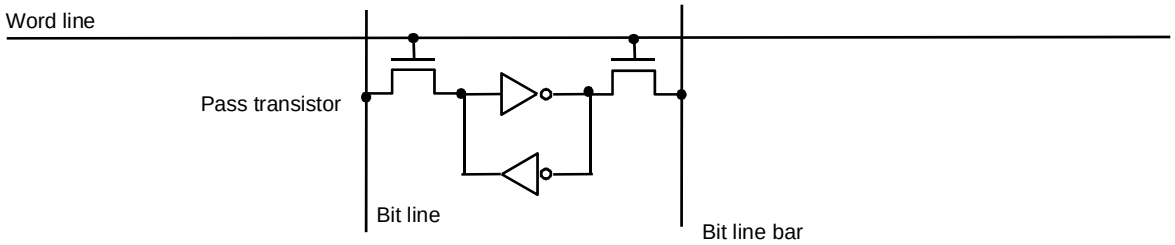
Cache Terminology

- Hit: data appears in some block
  - Hit Rate: the fraction of memory accesses found in the level
  - Hit Time: Time to access the level (consists of RAM access time + Time to determine hit)
- Miss: data needs to be retrieved from a block in the lower level (e.g., Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +
    Time to deliver the block to the processor
- Hit Time << Miss Penalty

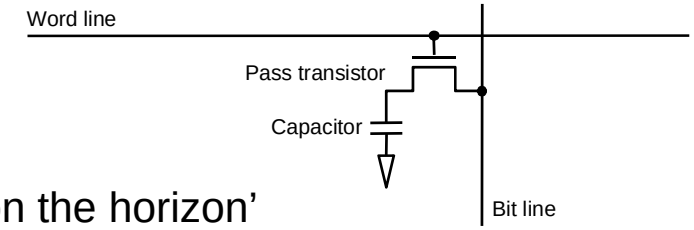


Memories: Two basic types*

- SRAM:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space than DRAM (4 to 6 transistors)

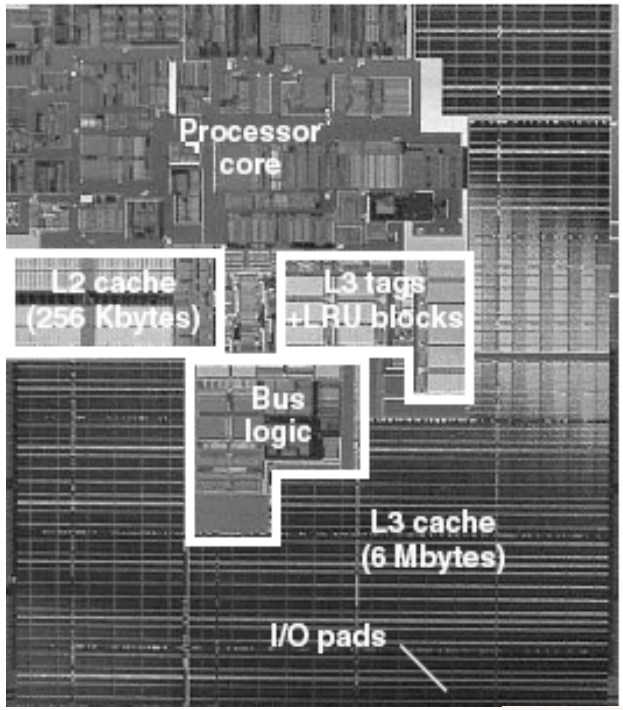


- DRAM:
  - value is stored as a charge on capacitor (must be refreshed)
  - very small but slower than SRAM (factor of 5 to 10)

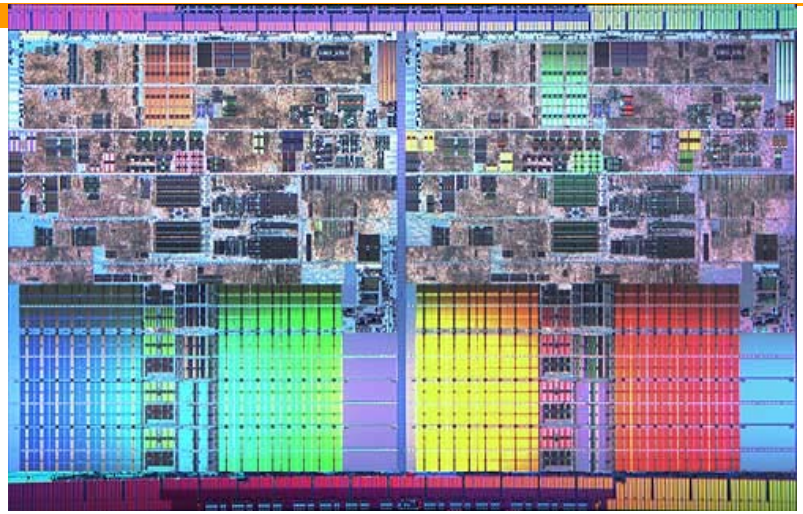


- * ignoring new technologies ‘on the horizon’

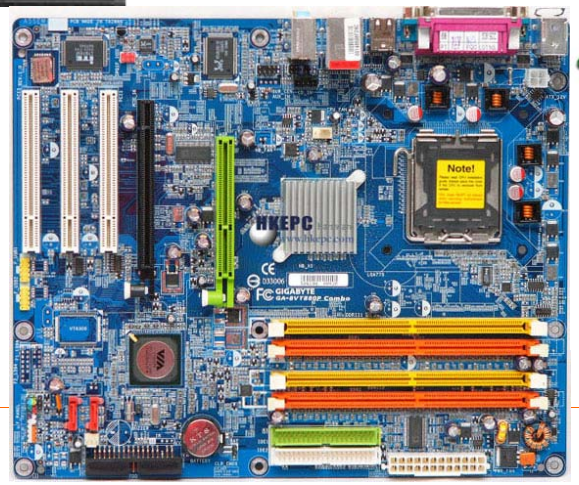
Memory Photos

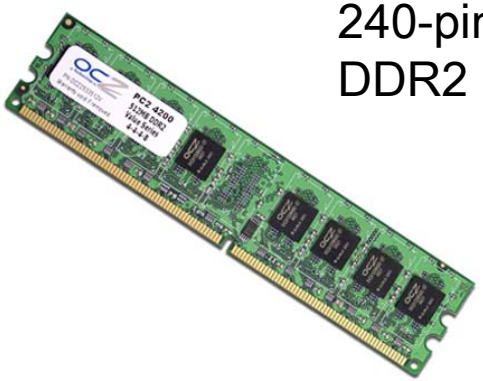


Intel Itanium2 .13µm
24-way 6MB L3



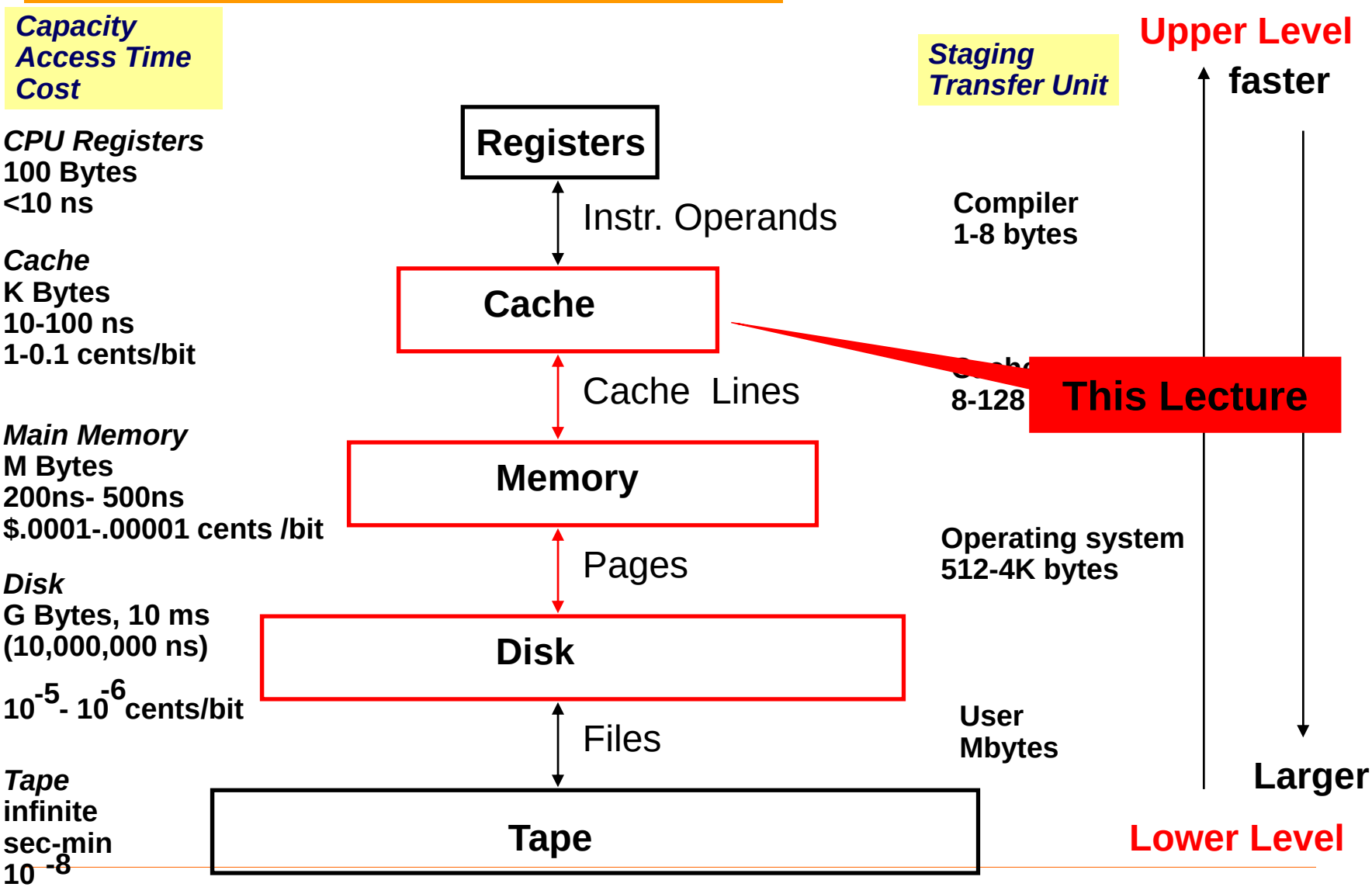
Intel Paxville (dual Core) 90nm
8-way 2MB L2 for each core



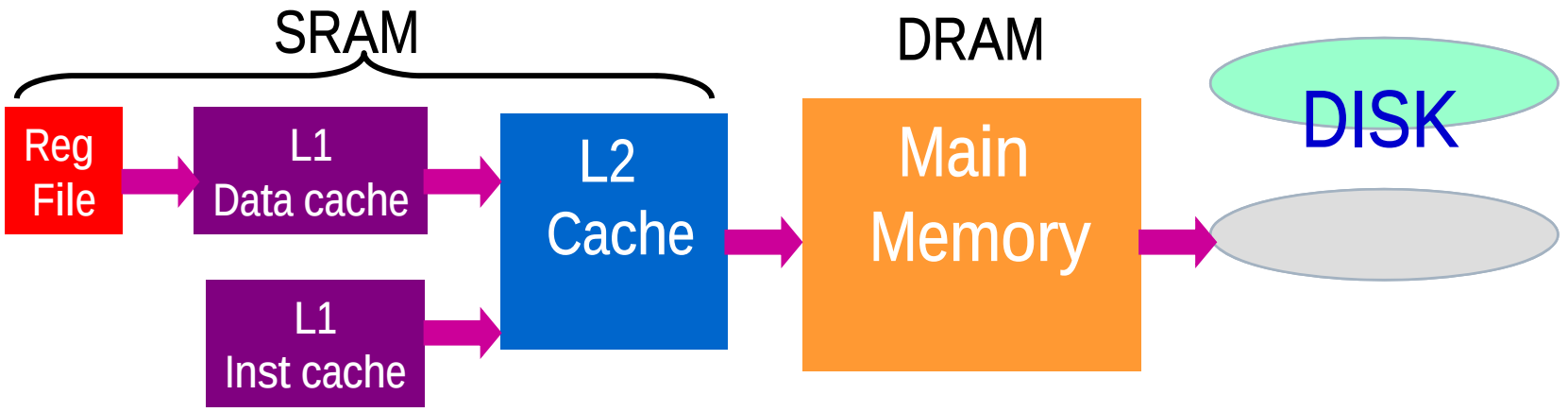


240-pin
DDR2 DRAM

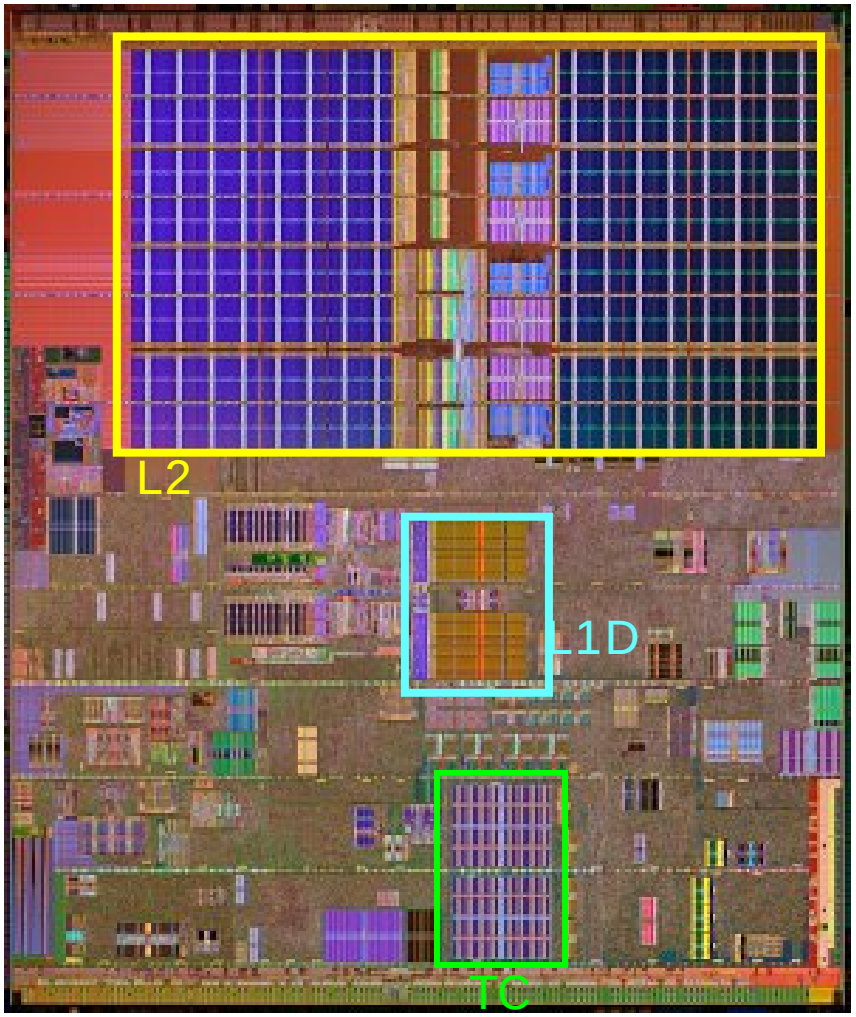
Levels of the Memory Hierarchy



Model of Memory Hierarchy



P4: Prescott w/ 2MB L2 (90nm)



- Prescott runs *very* fast (3.4+ GHz)
- 2MB L2 Unified Cache
- 12K* trace cache (think “I$”)
- 16KB data cache

- Check this out:
  - www.chip-architect.com

Locality

- A principle that makes having a memory hierarchy a good idea
- If an item is referenced,

  Temporal locality: it will tend to be referenced again soon

  Spatial locality: nearby items will tend to be referenced soon.

Why does code have locality? What about data locality?

- Our initial focus: two levels (upper, lower)
  - block: minimum unit of data
  - hit: data requested is in the upper level
  - miss: data requested is not in the upper level

Average Memory Access Time

- Average memory-access time
  = Hit time + Miss rate x Miss penalty
- *Miss penalty*: time to fetch a block from lower memory level
  - *access time*: function of latency
  - *transfer time*: function of bandwidth
    - **Transfer one "cache line/block" at a time**
    - **Transfer at the size of the memory-bus width**

Cache

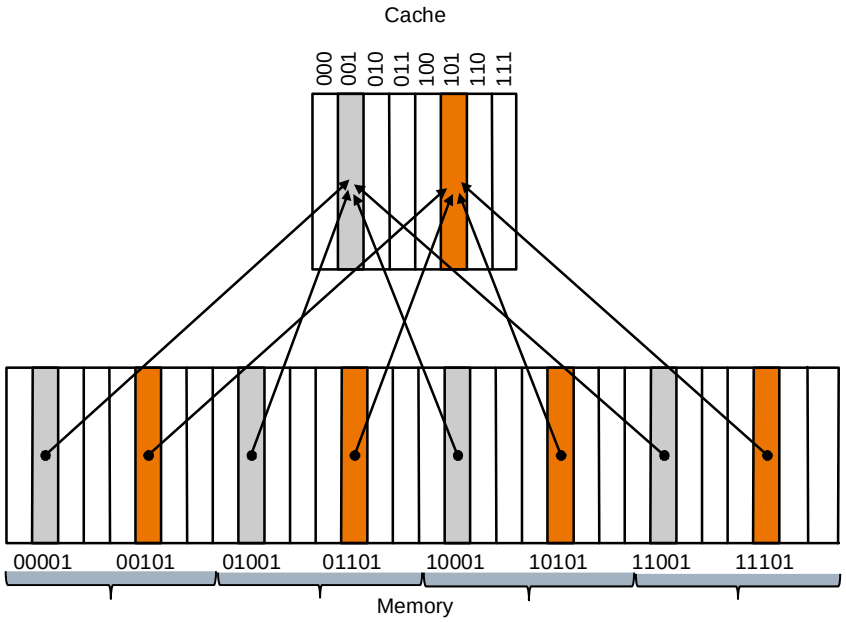
- Two issues:
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
- Our first example:
  - block size is one word of data
  - "direct mapped"

For each item of data at the lower level,
there is exactly one location in the cache where it might be.

e.g., lots of items at the lower level share locations in the upper level

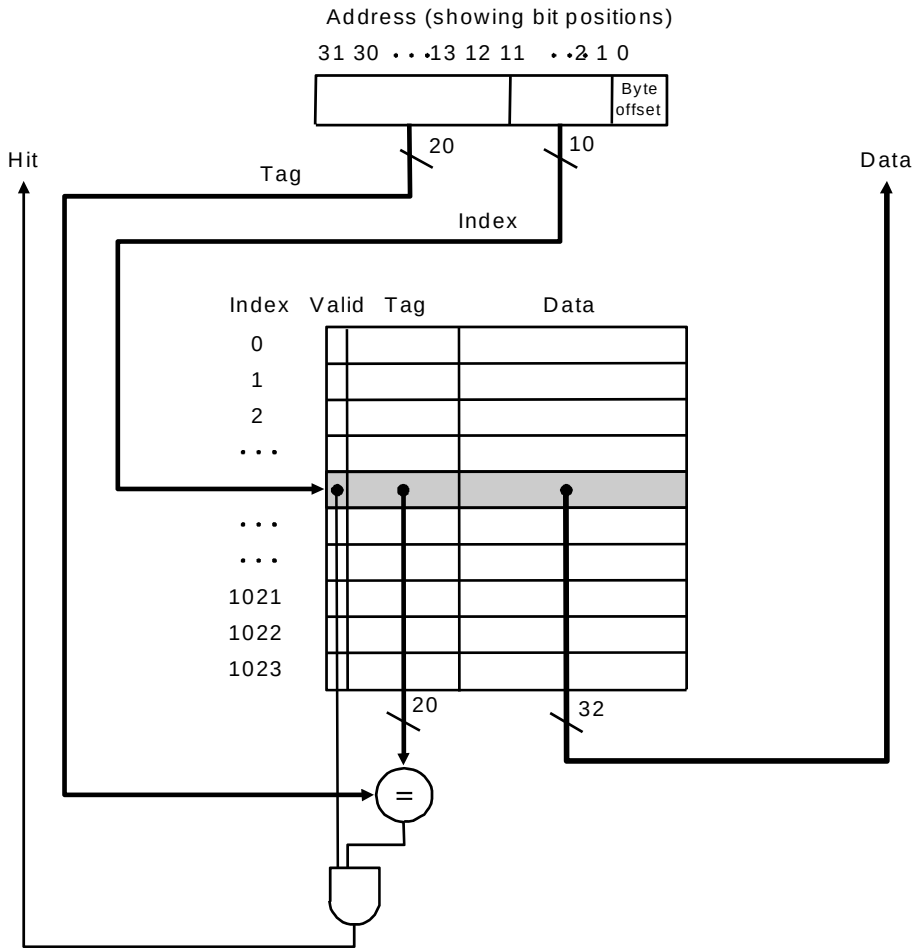
Direct Mapped Cache

- Mapping: address is modulo the number of blocks in the cache



Direct Mapped Cache

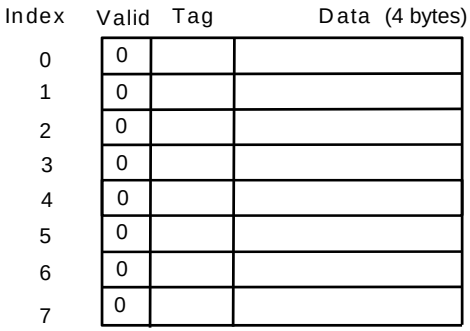
- For MIPS:

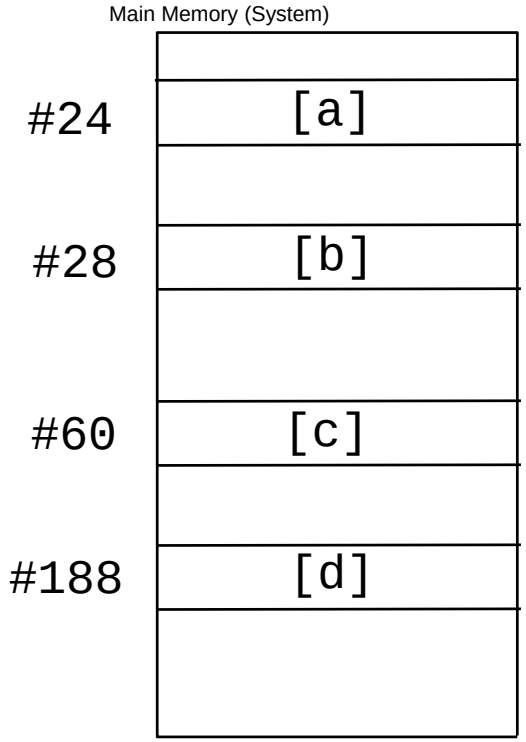


What kind of locality are we taking advantage of?

Example: DM$, 8-Entry, 4B

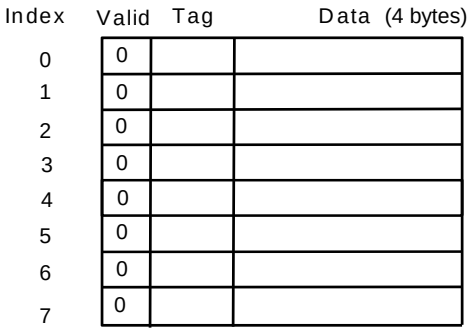
```
lw $0, #24   [a]
lw $1, #28   [b]
sw $2, #60   [c]
sw $3, #188  [d]
```

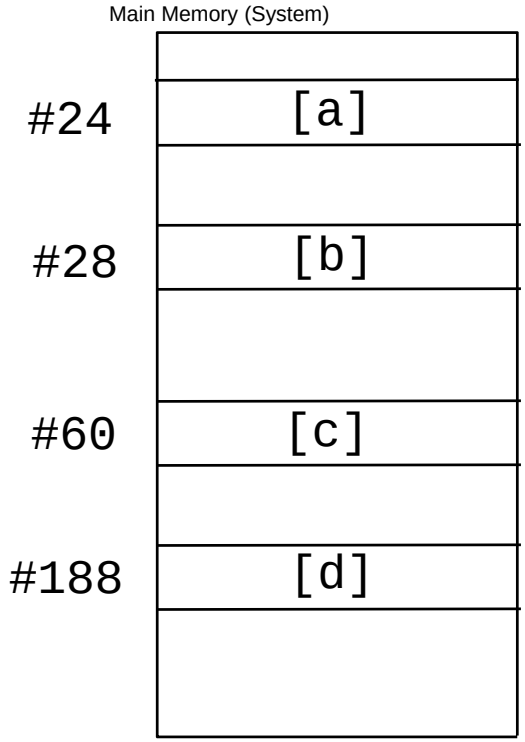




Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



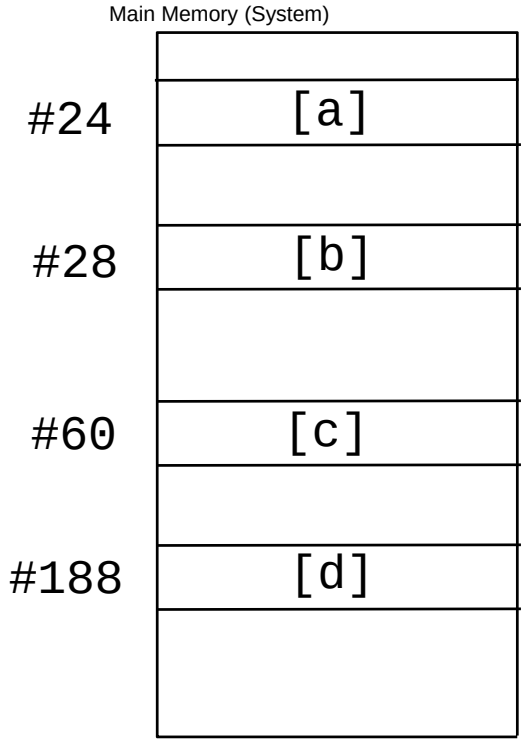


Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#24 is 0001 1000

Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

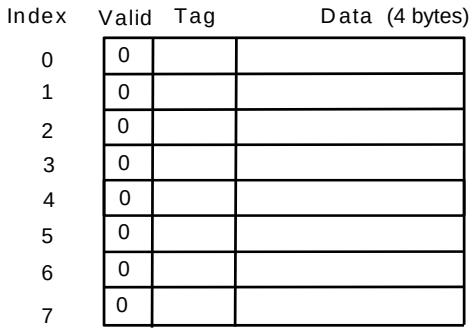


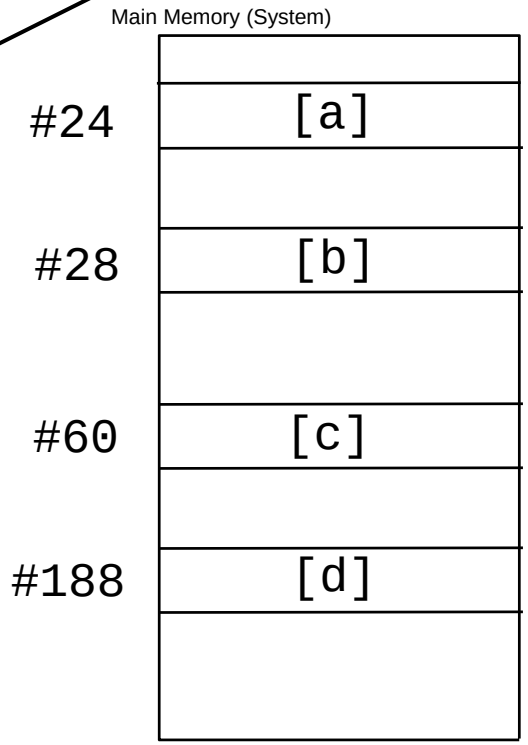
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

4-byte block, drop low 2 bits for byte offset! Only matters for byte-addressable systems

#24 is 0001 1000





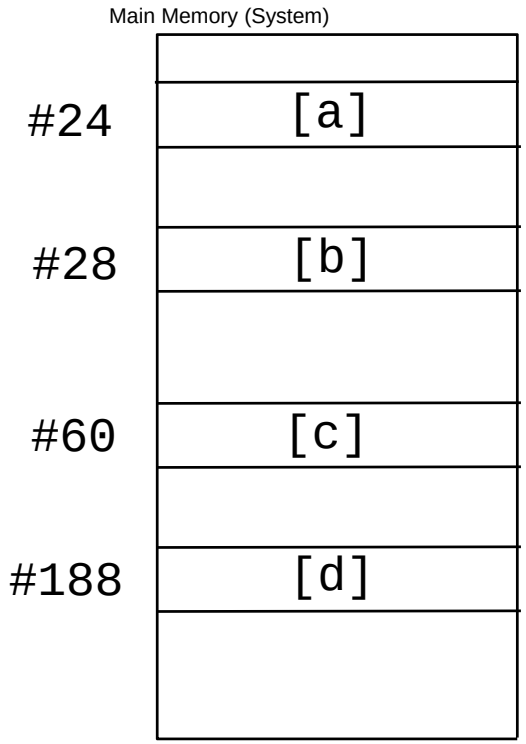
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#24 is 0001 1000

Next $\log_2(8)$ bits mod 8 = Index

Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		



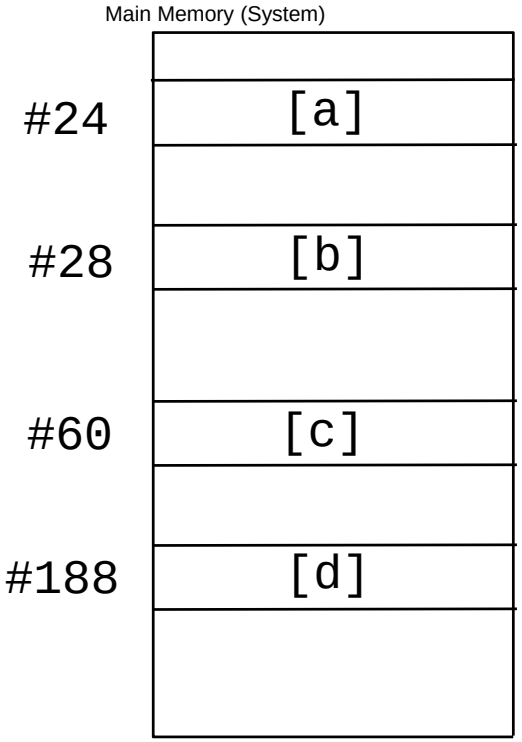
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#24 is 0001 1000

Next $\log_2(8)$ bits mod 8 = Index

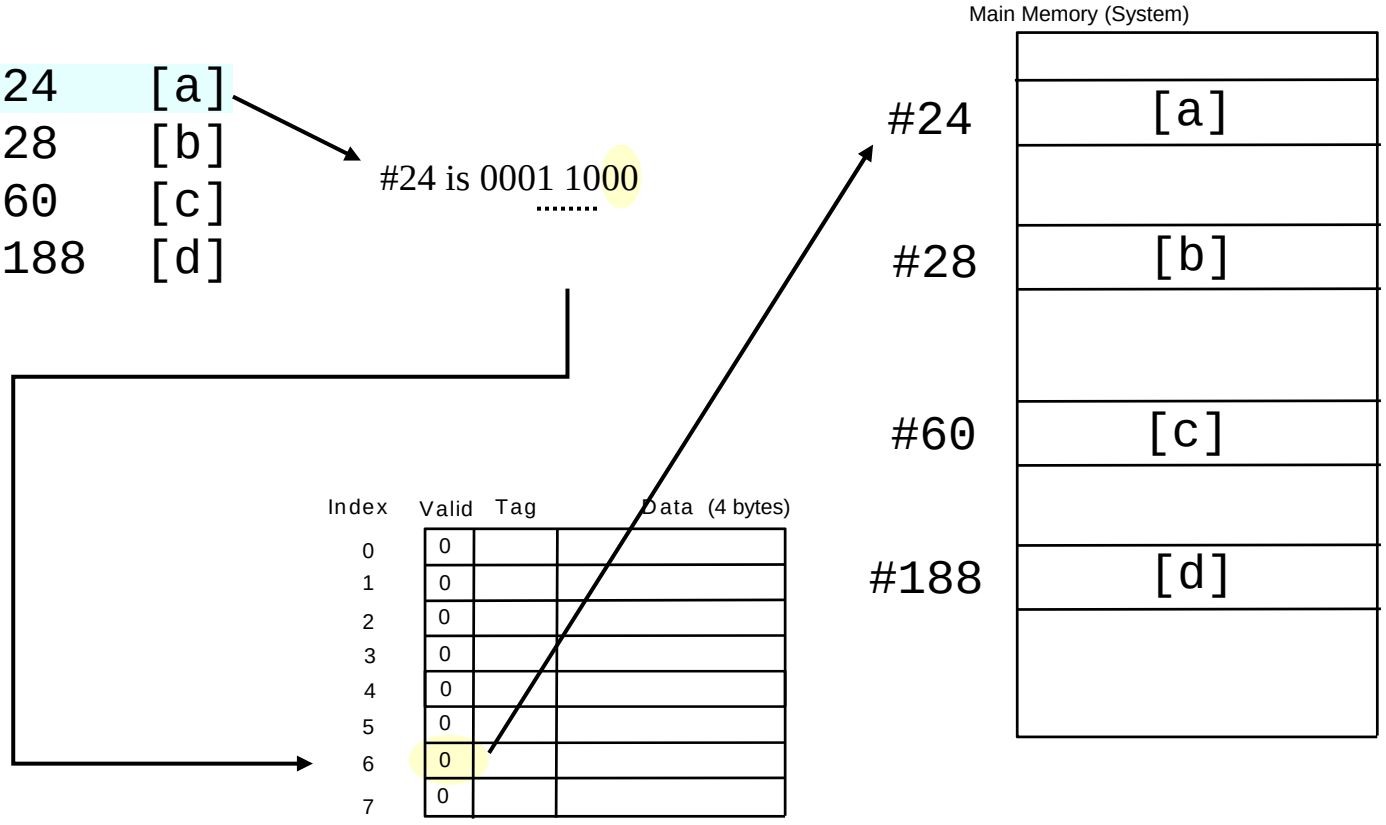
Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		



Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

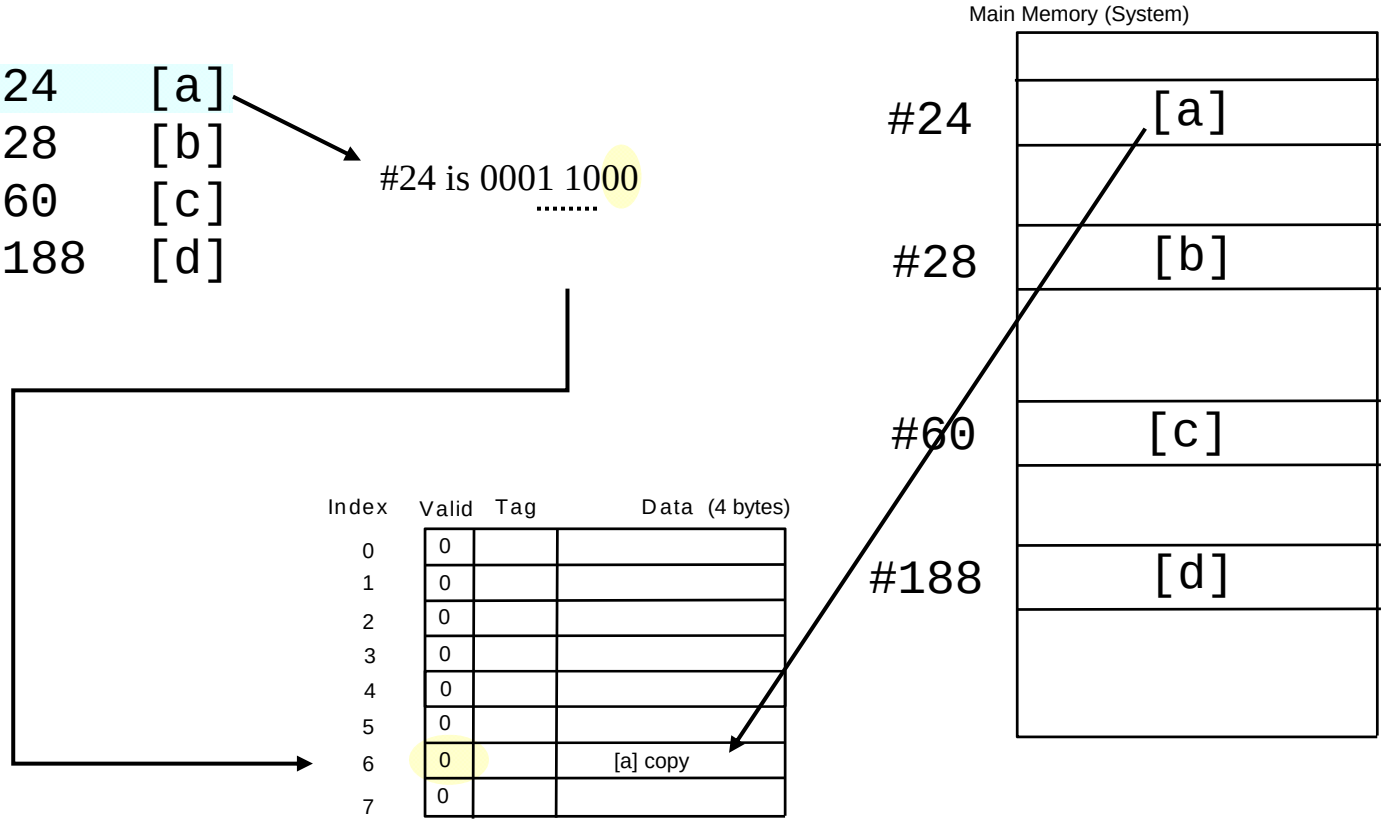
#24 is 0001 1000



Example: DM$, 8-Entry, 4B

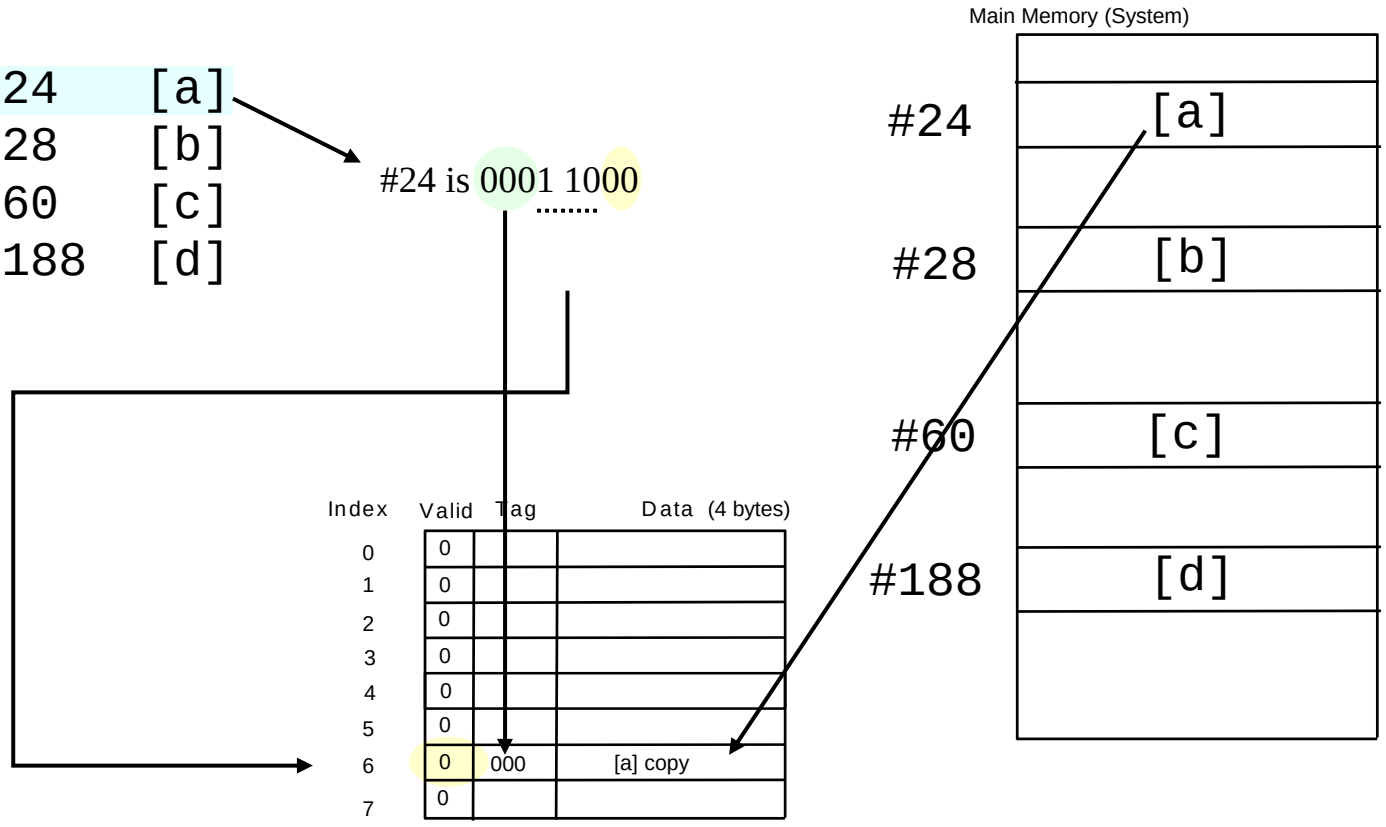
```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#24 is 0001 1000



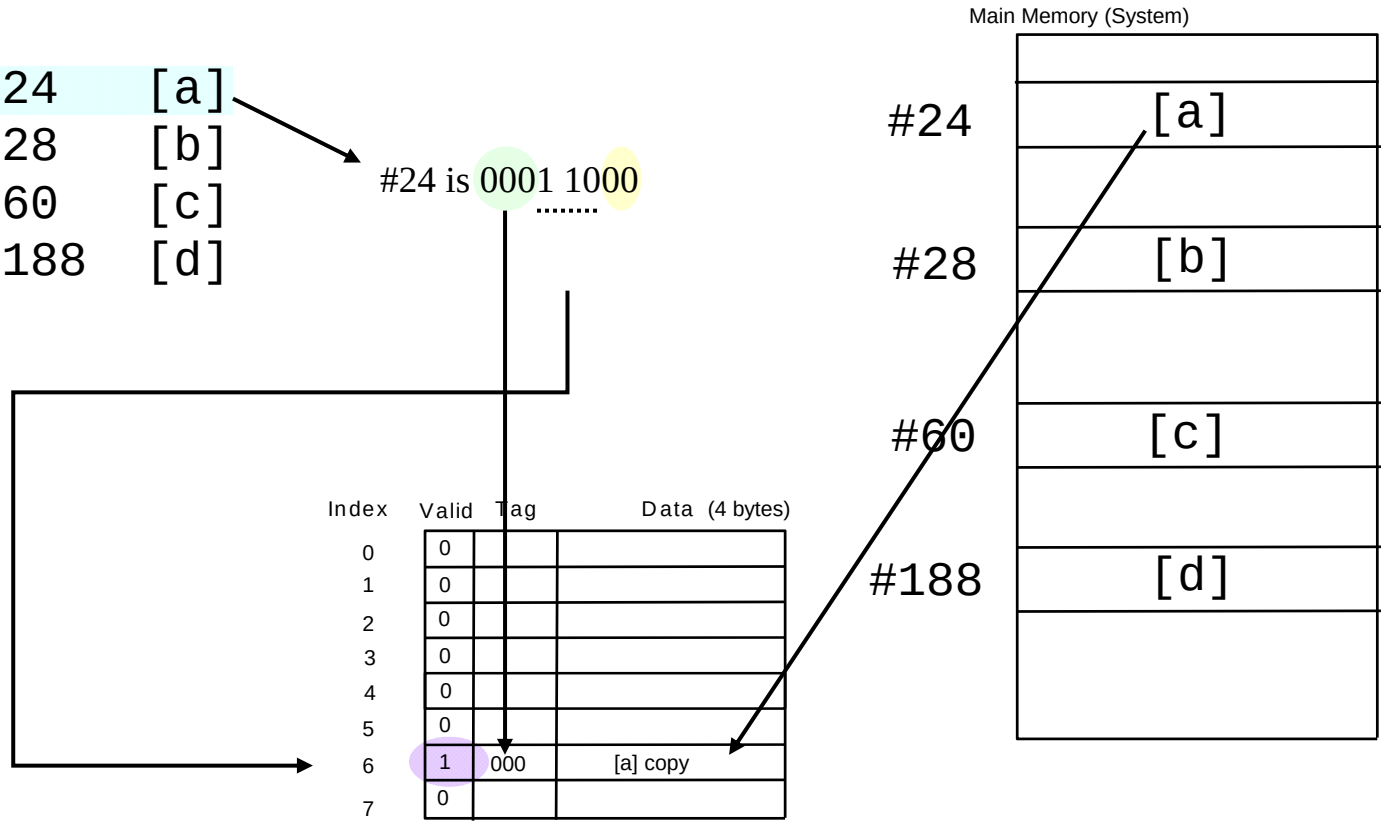
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



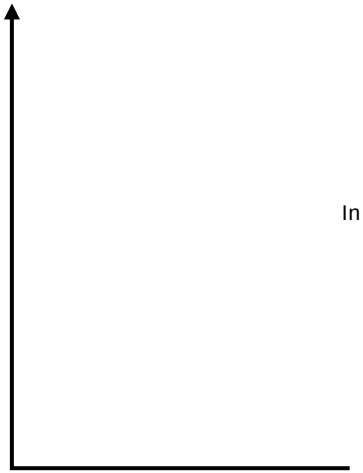
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

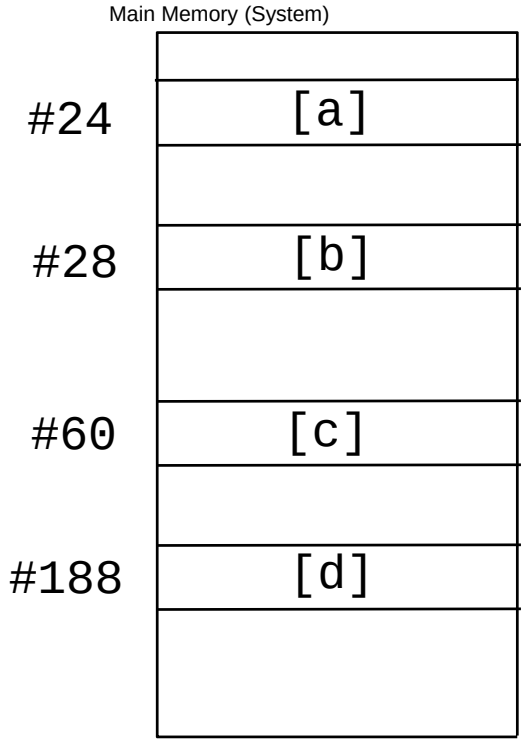


Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

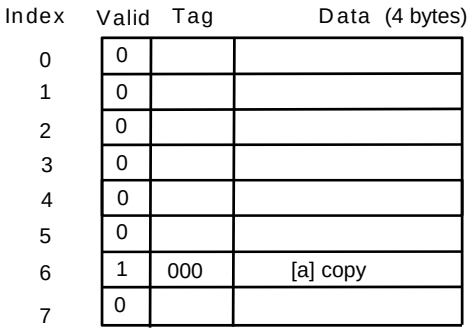


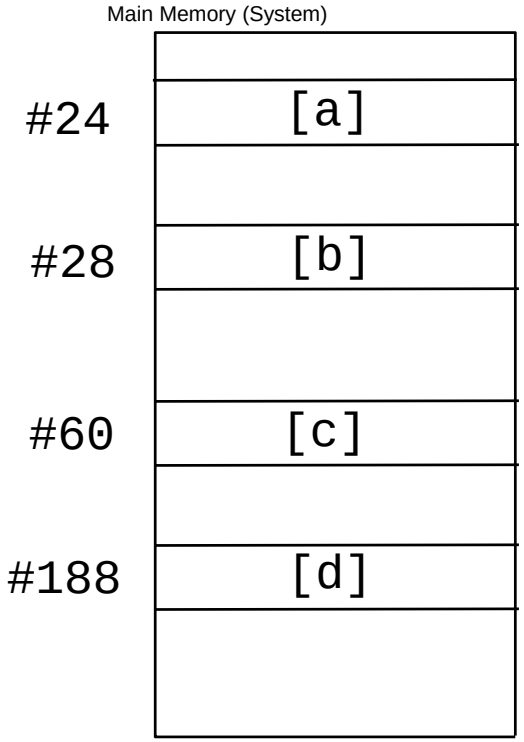
Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	0		



Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



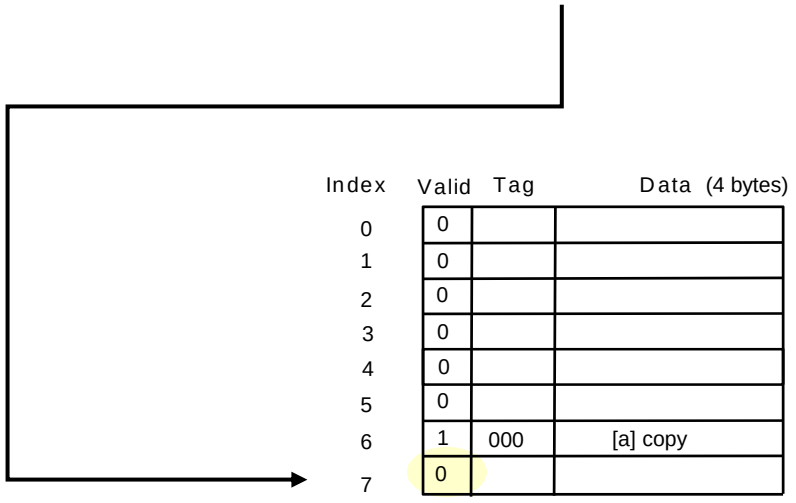


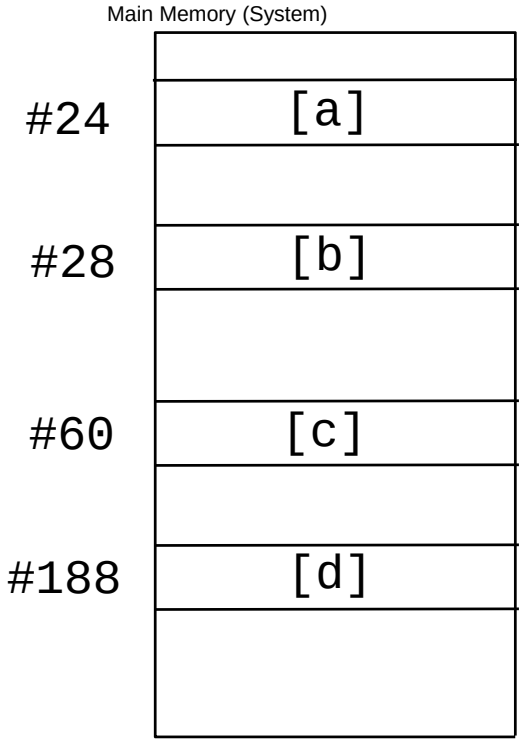
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#28 is 0001 1100

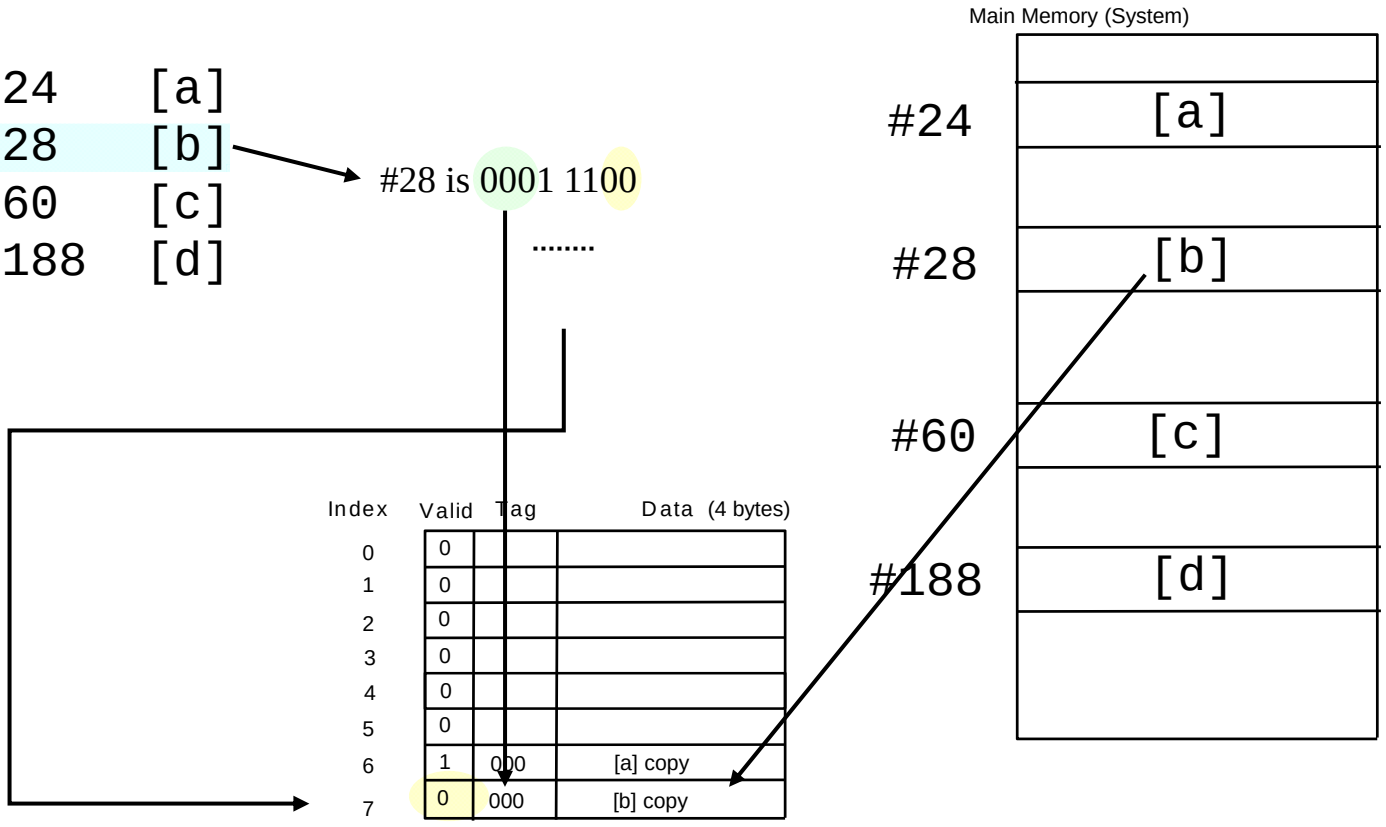
.....





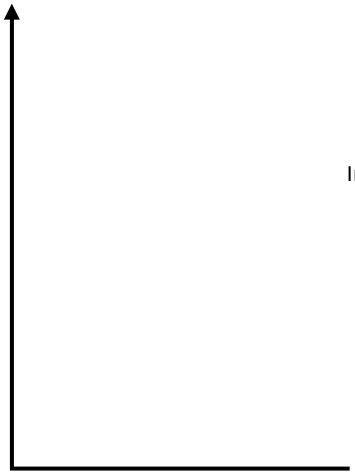
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

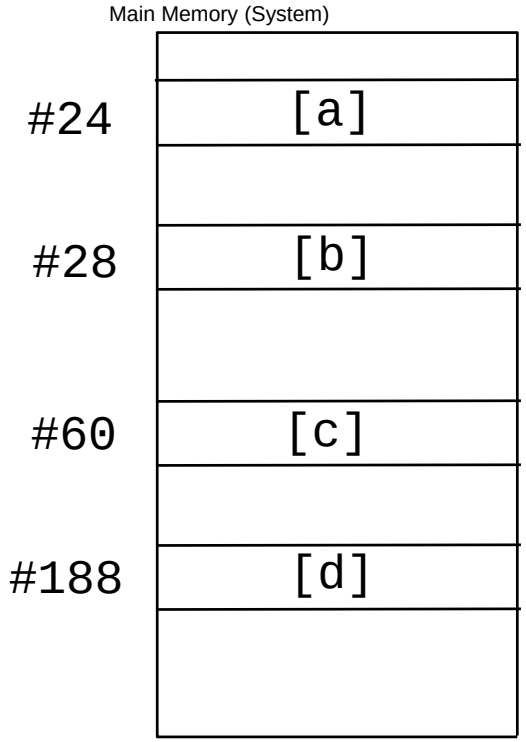


Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

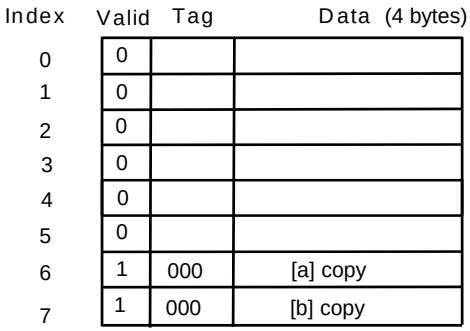


Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	000	[b] copy

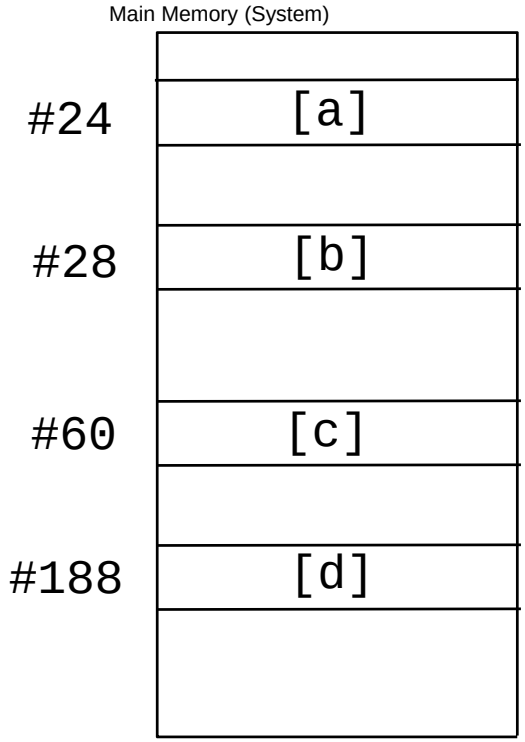


Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	000	[b] copy

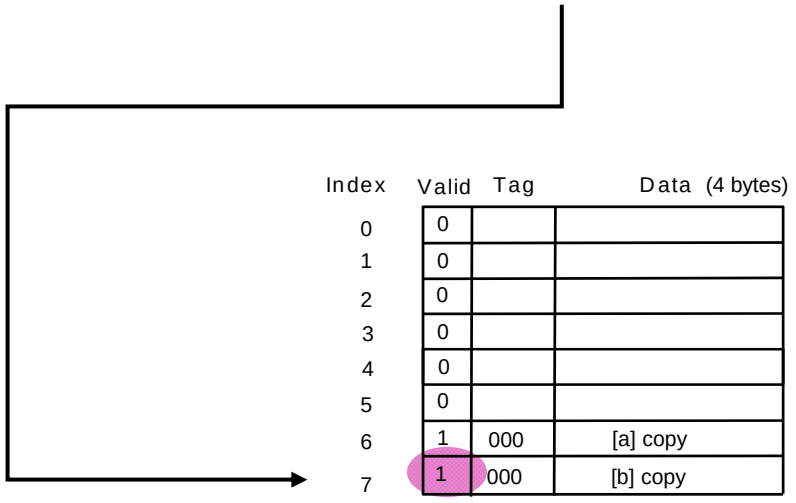


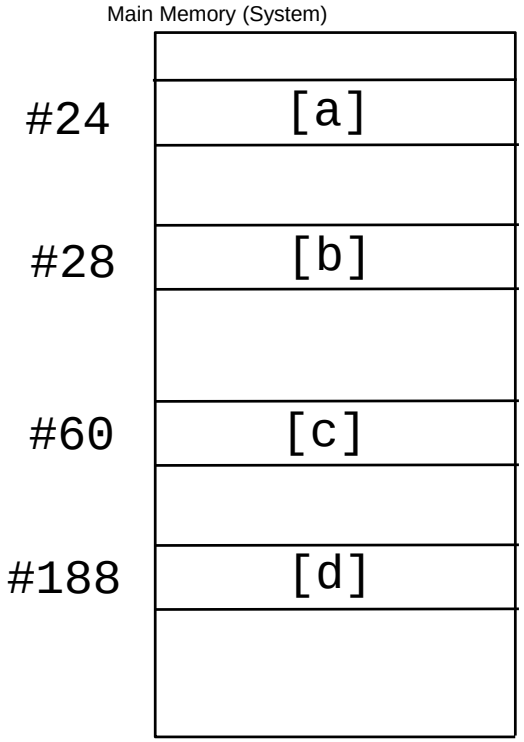
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#60 is 0011 1100

.....

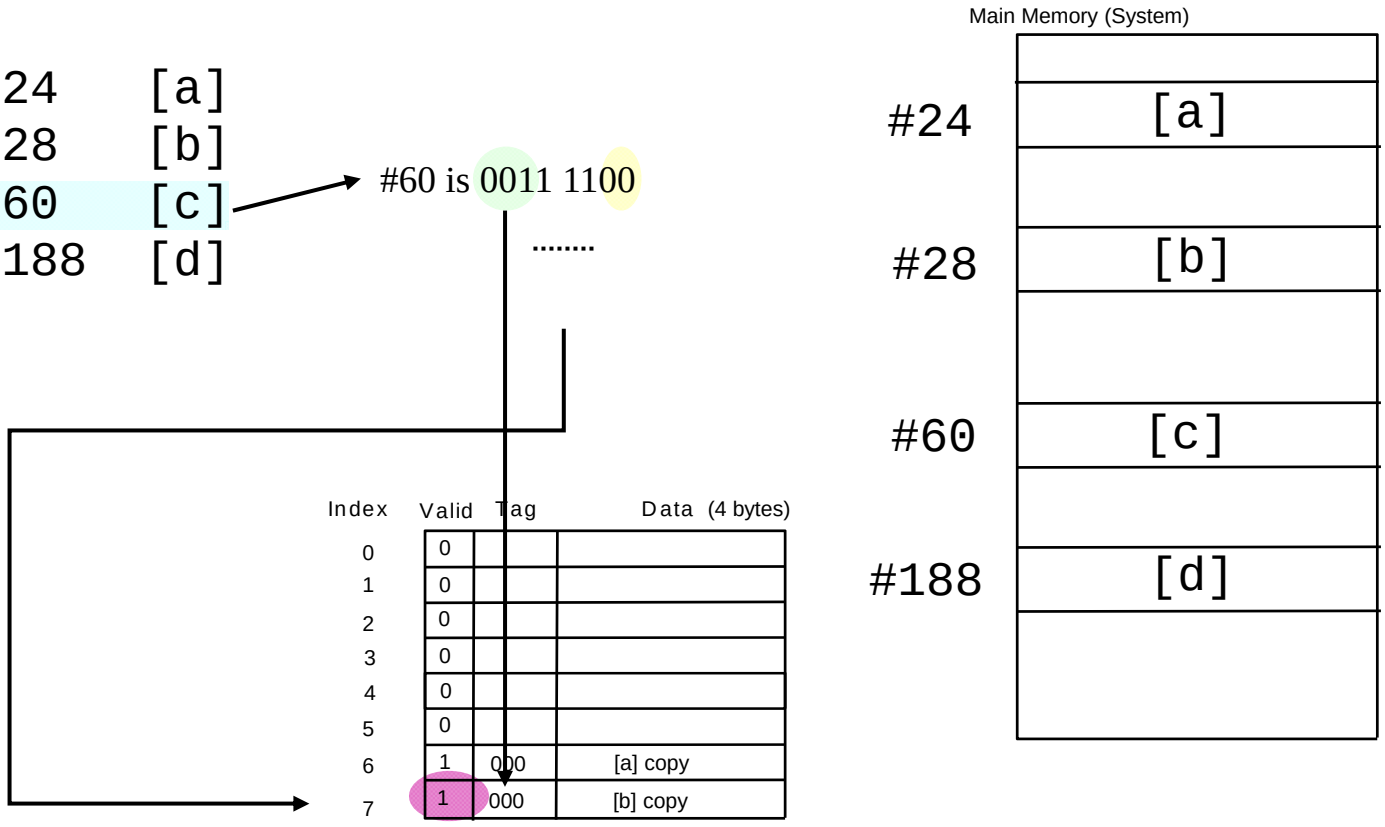




It's valid! How to tell it's the wrong address?

Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



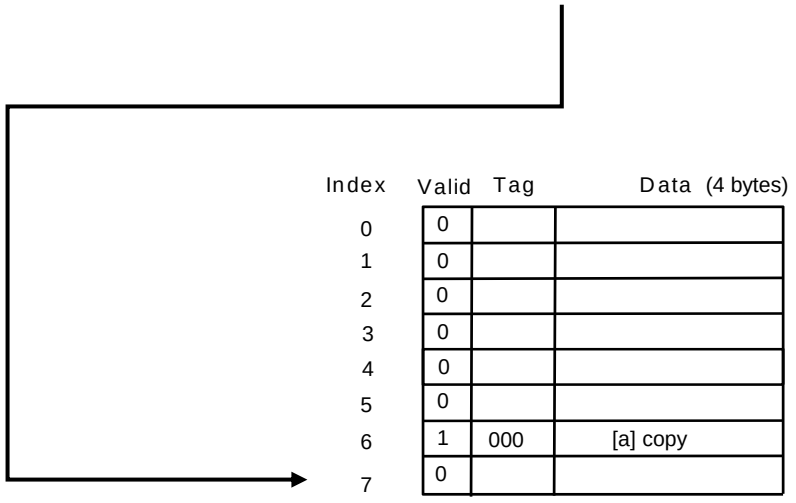
The tags don't match! It's not what we want to access!

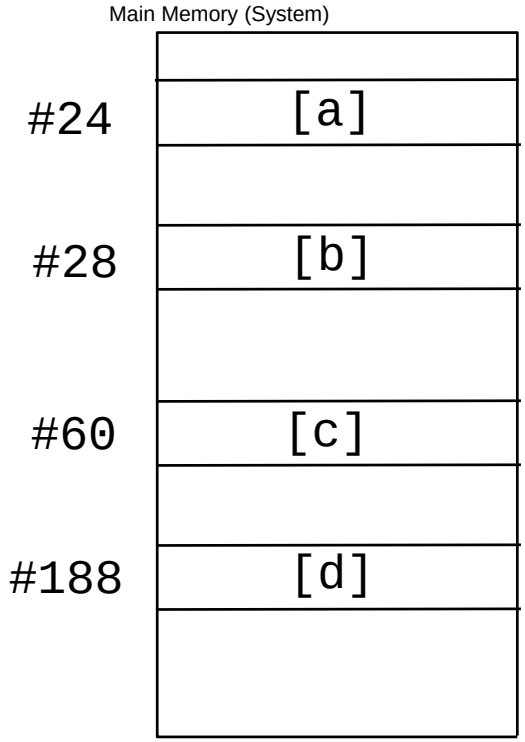
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

#60 is 0011 1100

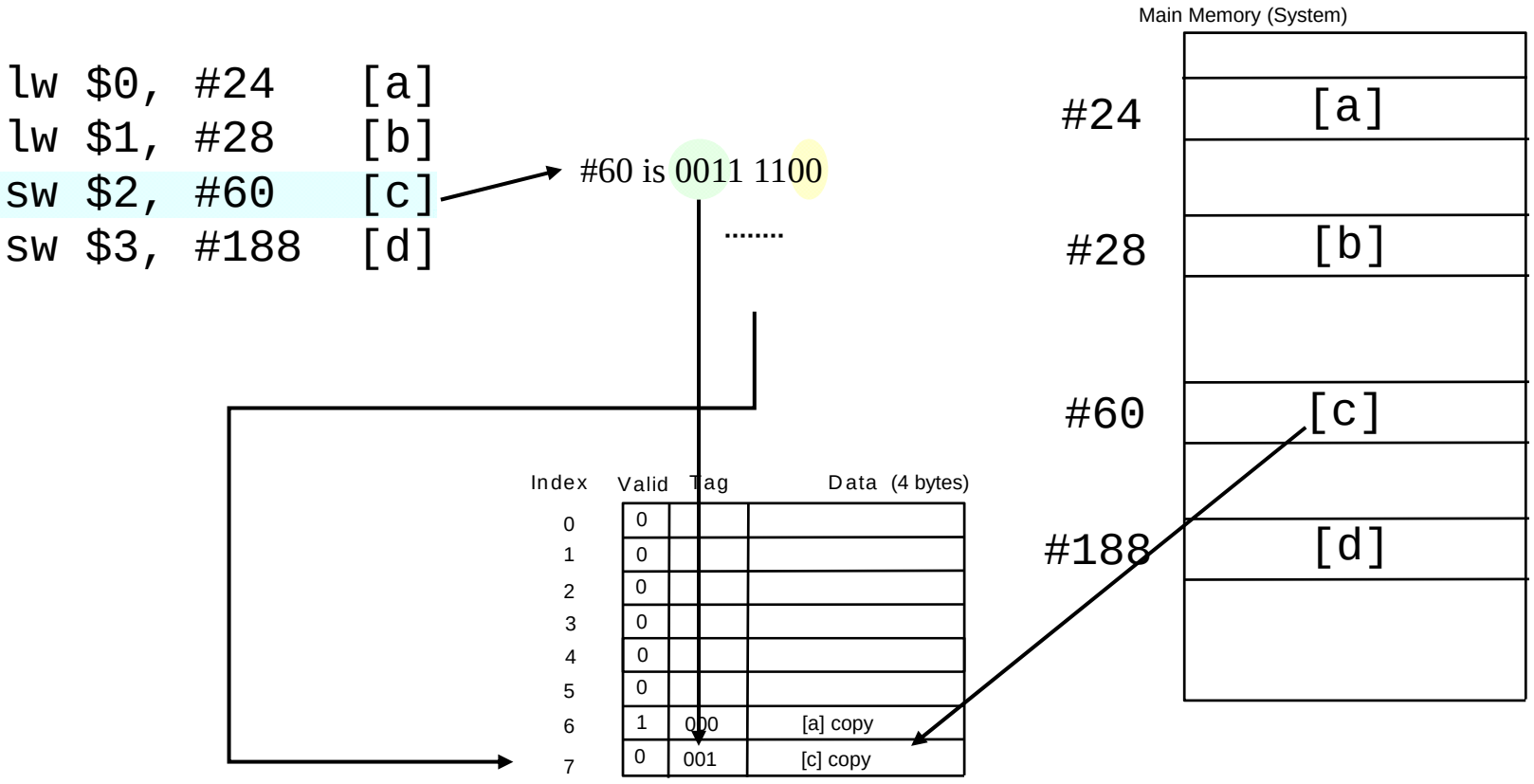
.....





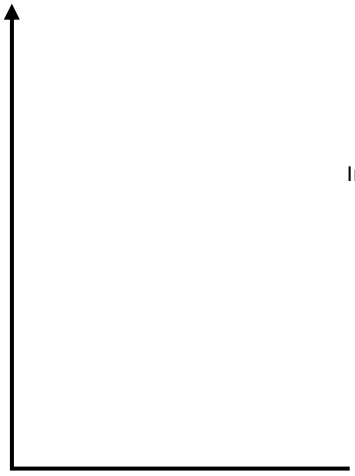
Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

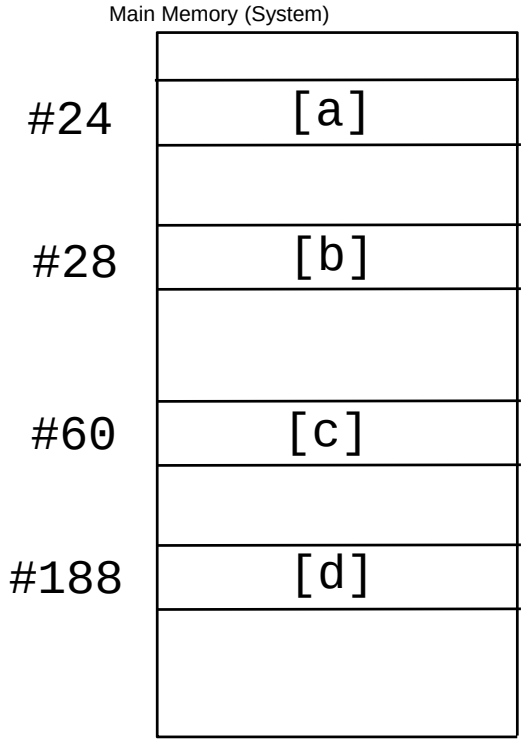


Example: DM$, 8-Entry, 4B

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



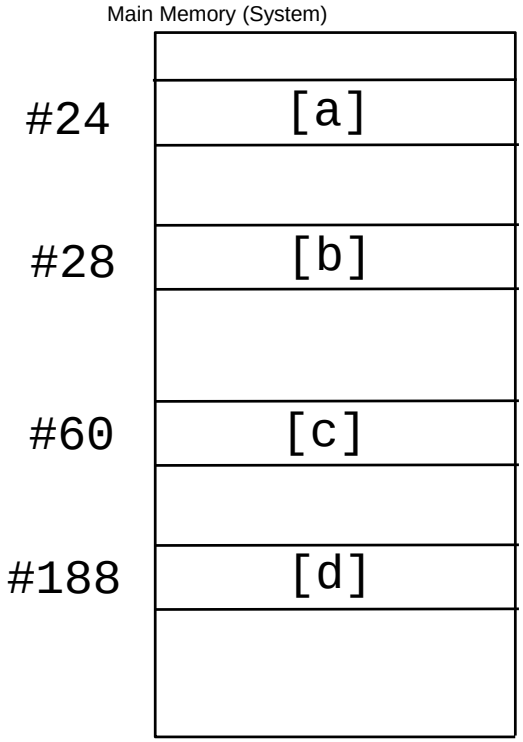
Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	001	[c] copy



Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24   [a]
lw $1, #28   [b]
sw $2, #60   [c]
sw $3, #188  [d]
```

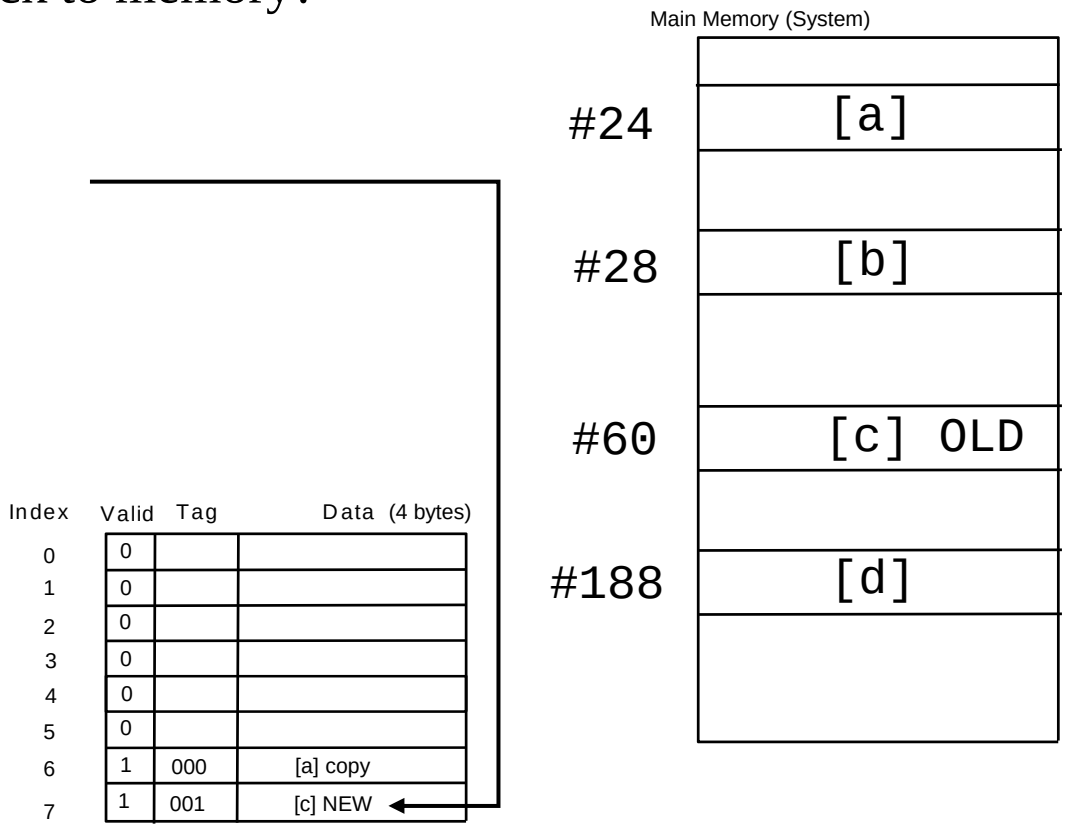


Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	001	[c] copy

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

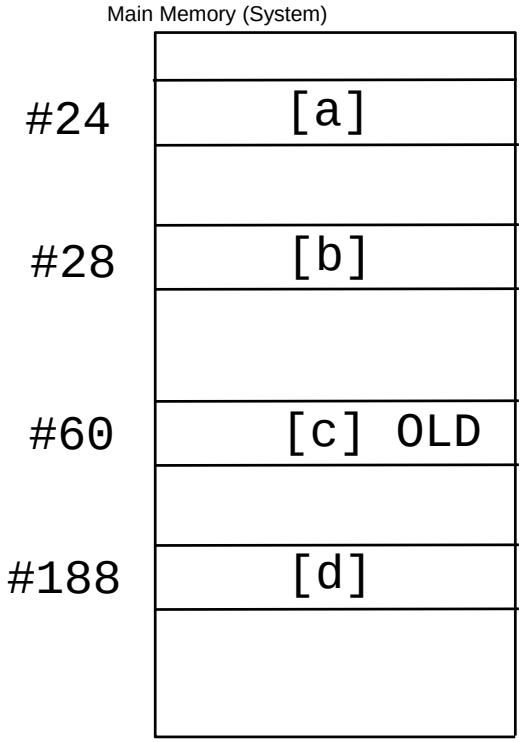


Do we update memory now? Or later?

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



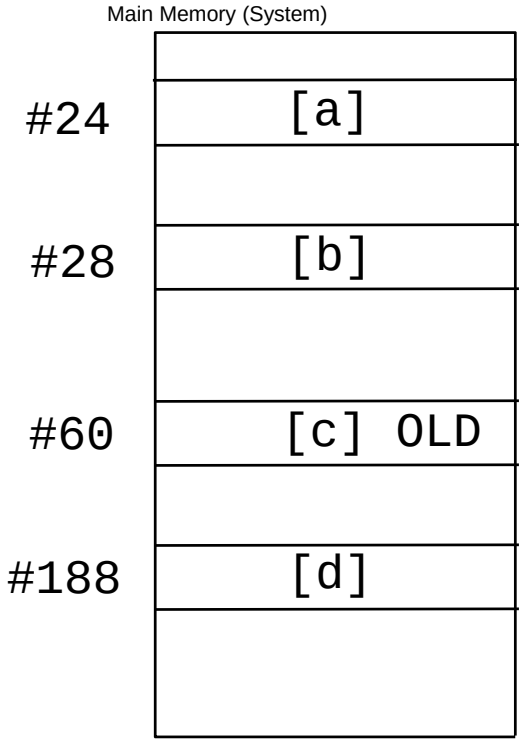
Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	001	[c] NEW

Assume later...

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

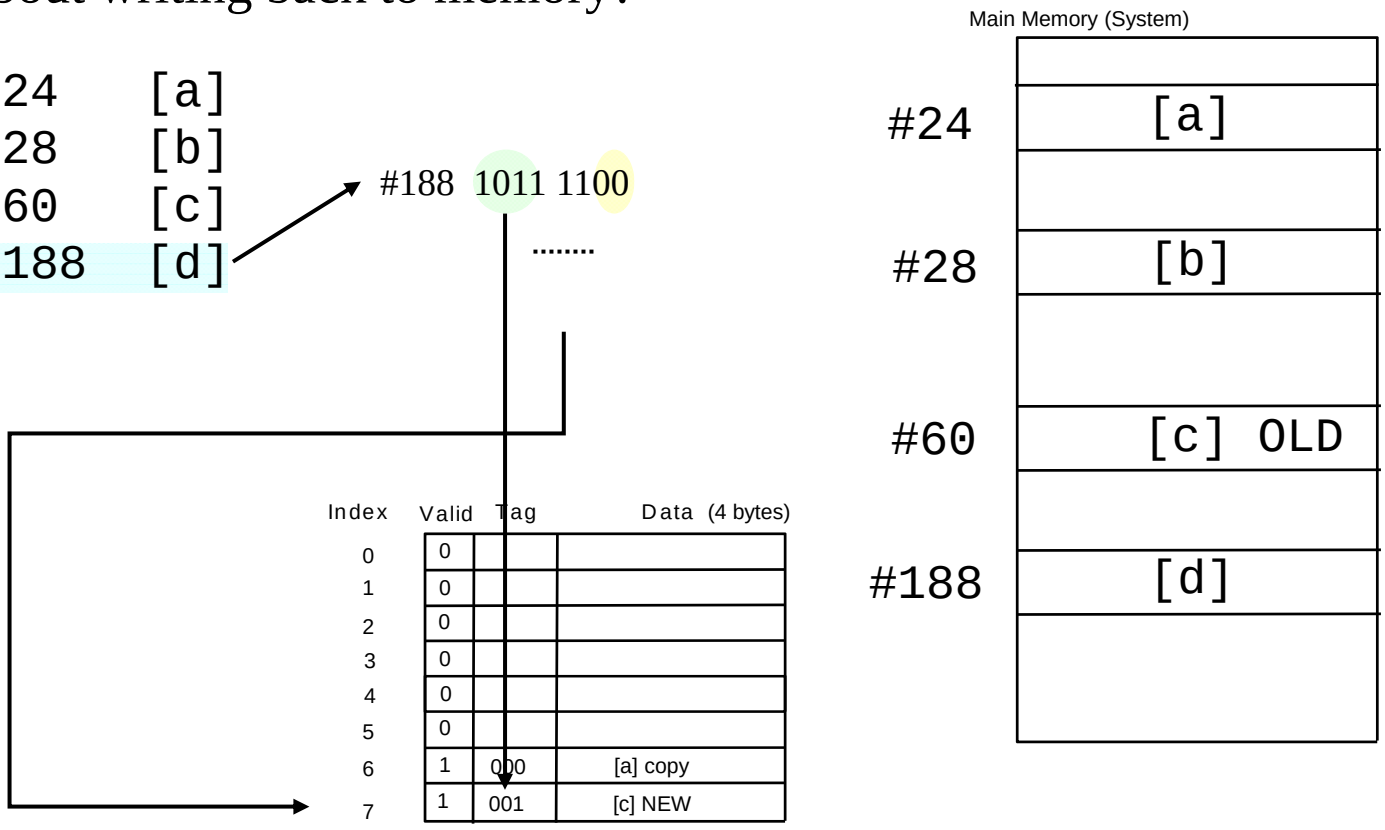


Index	Valid	Tag	Data (4 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	1	000	[a] copy
7	1	001	[c] NEW

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

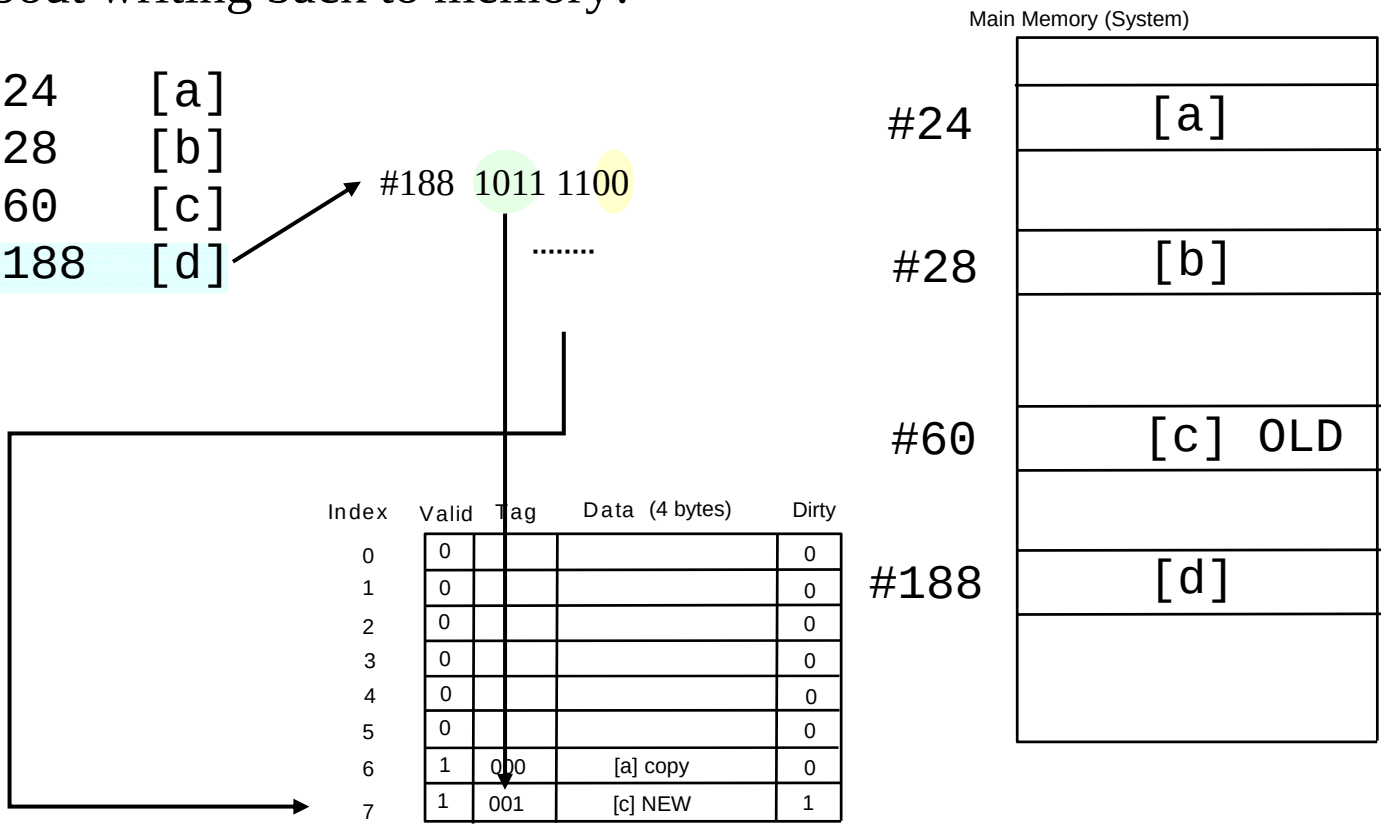


Now What? How do we know to write back?

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```

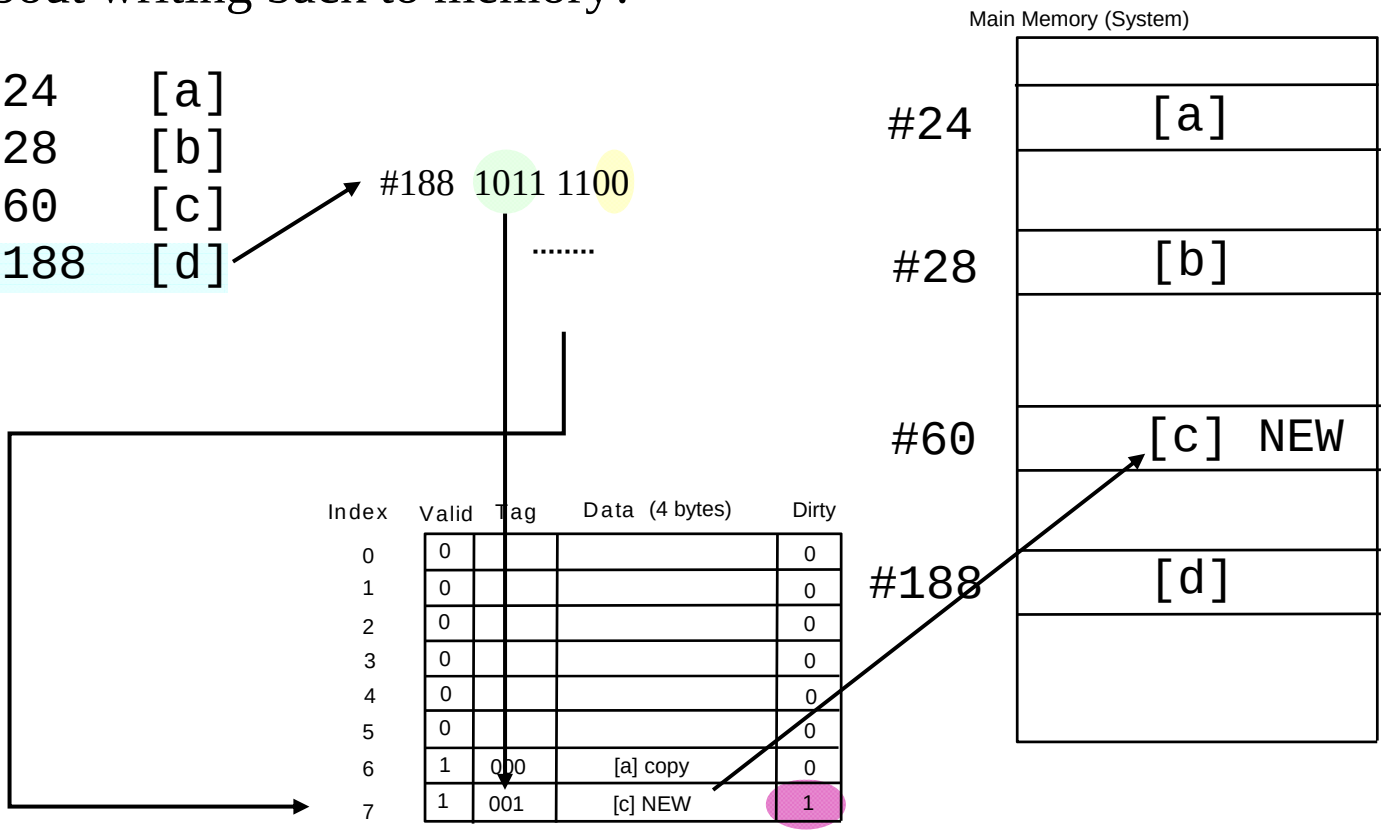


Need extra state! The “dirty” bit!

Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

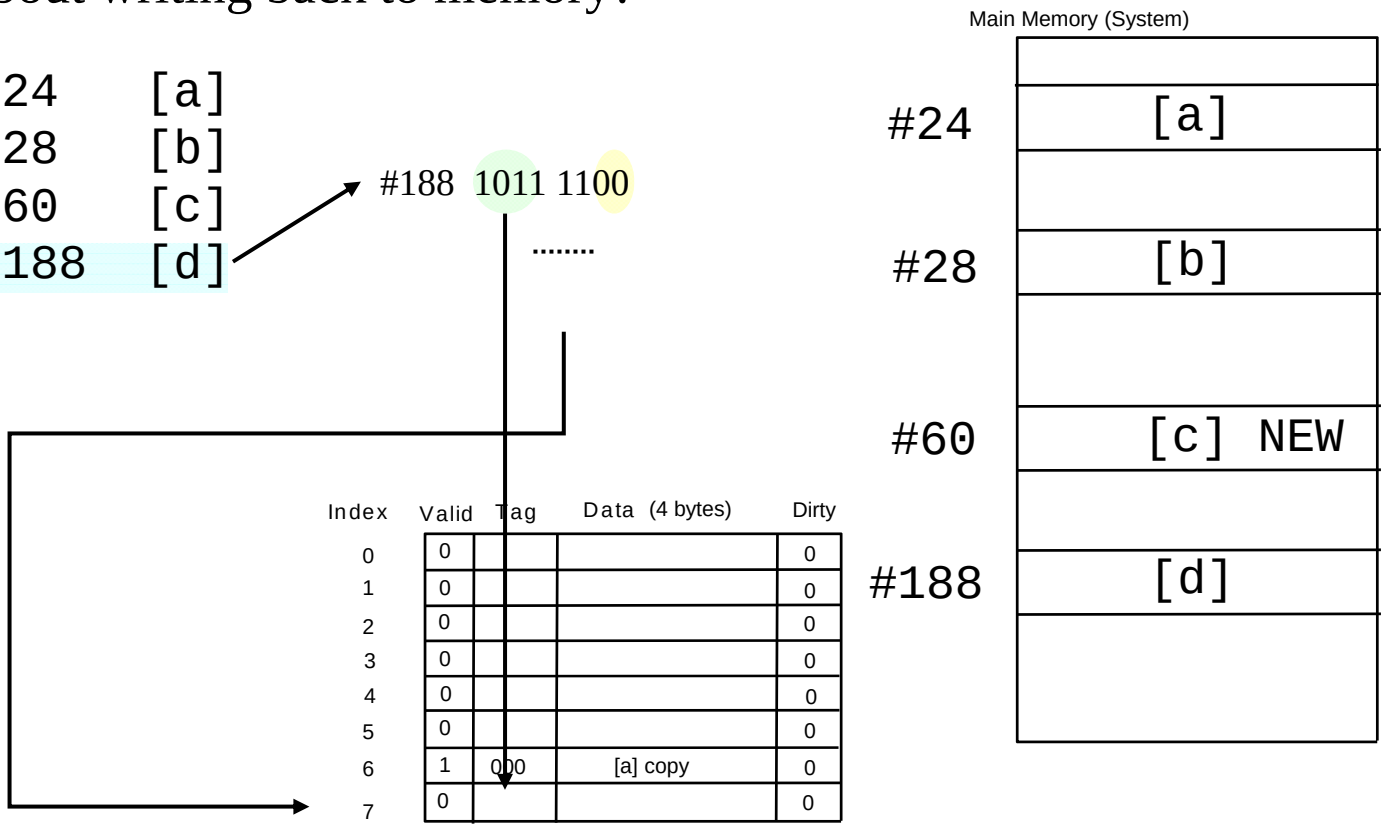
```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

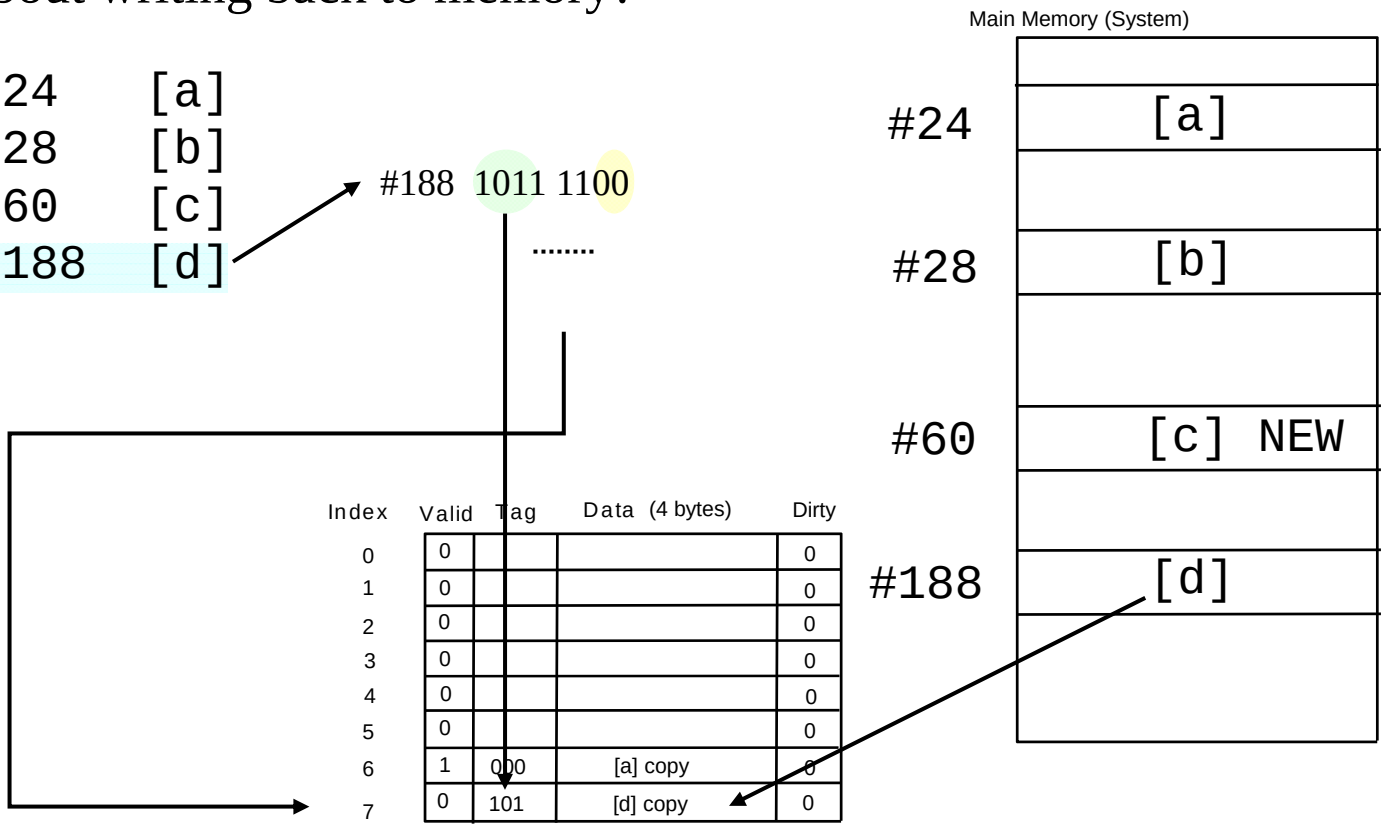
```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

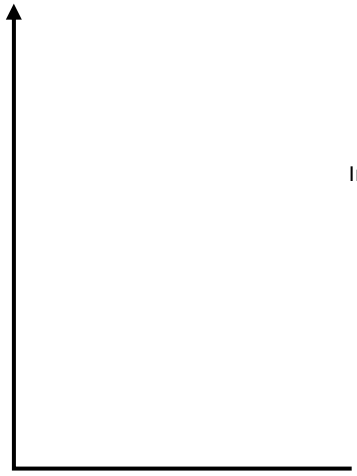
```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



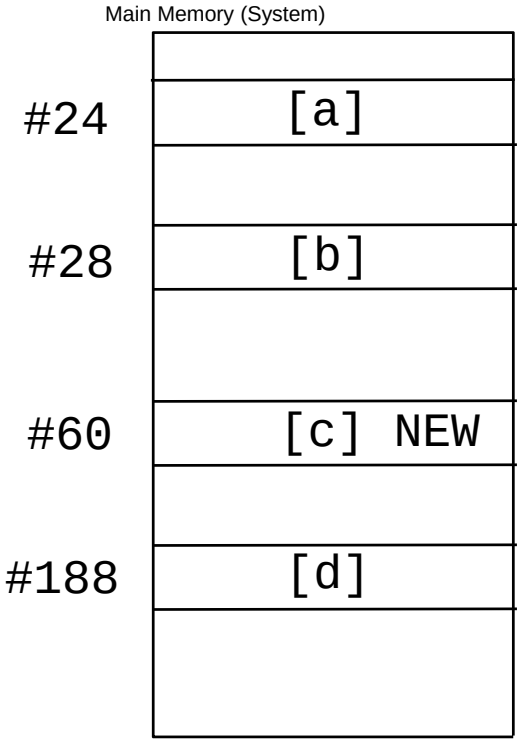
Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



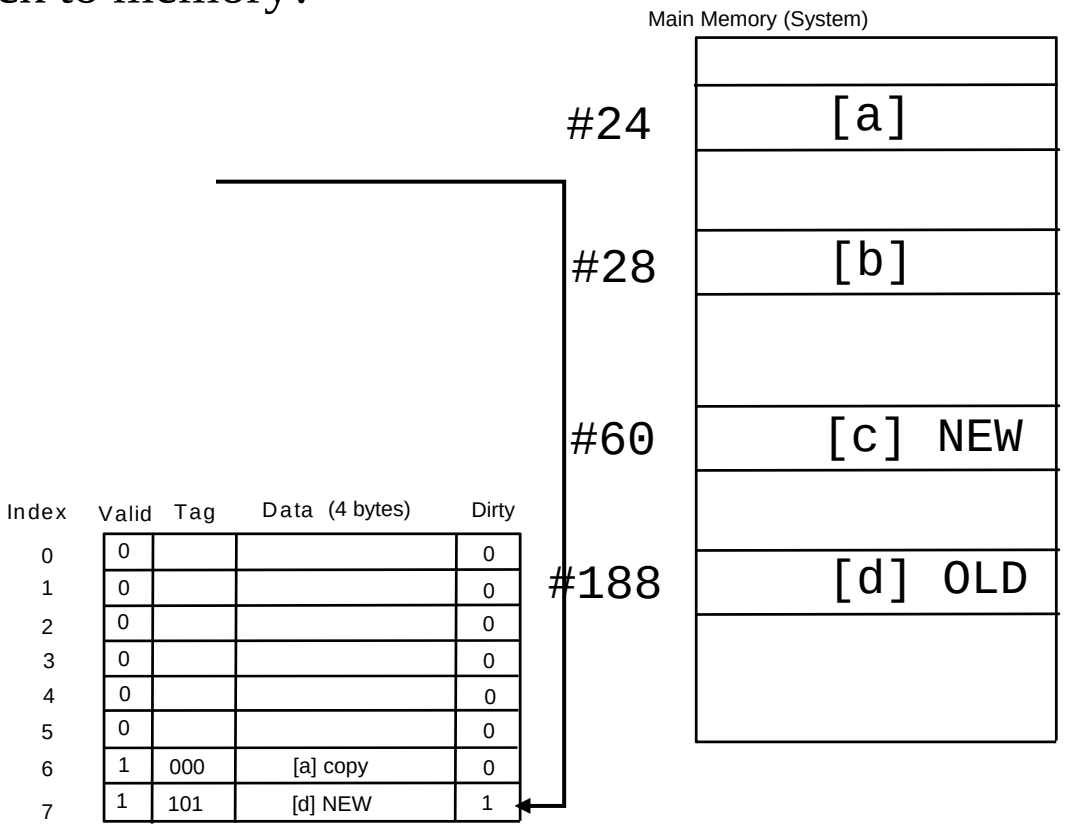
Index	Valid	Tag	Data (4 bytes)	Dirty
0	0			0
1	0			0
2	0			0
3	0			0
4	0			0
5	0			0
6	1	000	[a] copy	0
7	1	101	[d] copy	0



Example: DM$, 8-Entry, 4B

Q: What about writing back to memory?

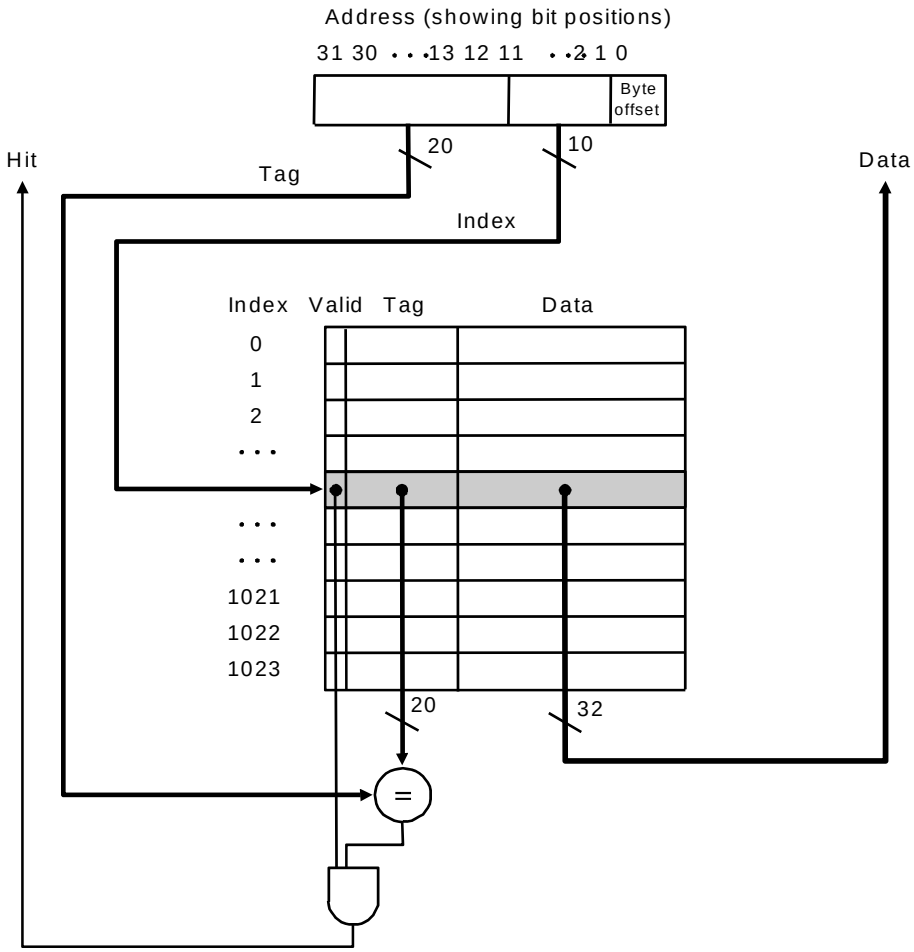
```
lw $0, #24    [a]
lw $1, #28    [b]
sw $2, #60    [c]
sw $3, #188   [d]
```



DM$: Thoughts

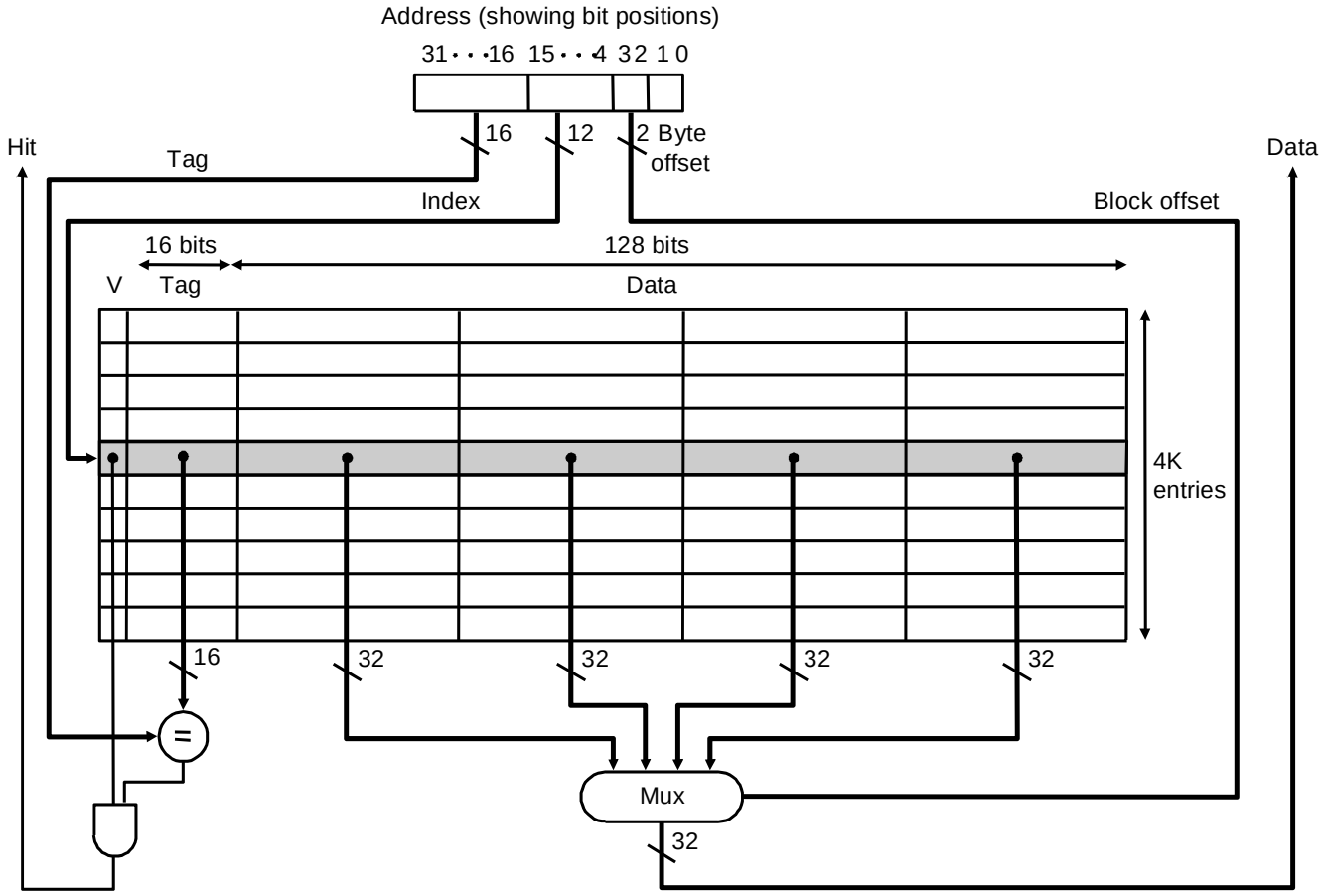
- Trade-Offs:
  - Write-back or Write-Through?
  - Write-Alloc or No-Write-Alloc?
  - How does Tag change with # of Entries?
  - How does minimum machine word size impact tag?

What kind of locality are we taking advantage of?



Direct Mapped Cache

- Taking advantage of spatial locality:

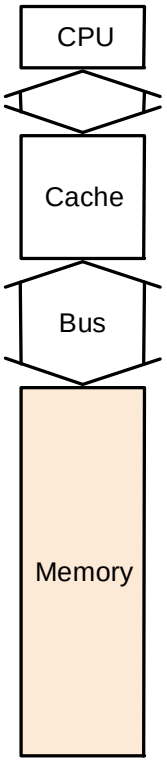


Hits vs. Misses

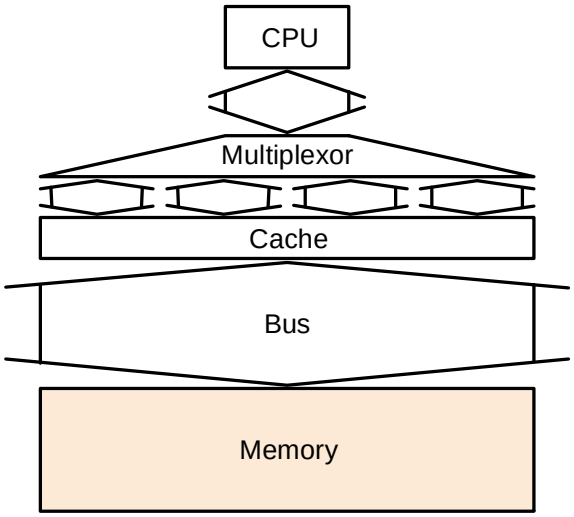
- Read hits
  - this is what we want!
- Read misses
  - stall the CPU, fetch block from memory, deliver to cache, restart
- Write hits:
  - can replace data in cache and memory (write-through)
  - write the data only into the cache (write-back the cache later)
- Write misses:
  - read the entire block into the cache, then write the word… ?

Hardware Issues

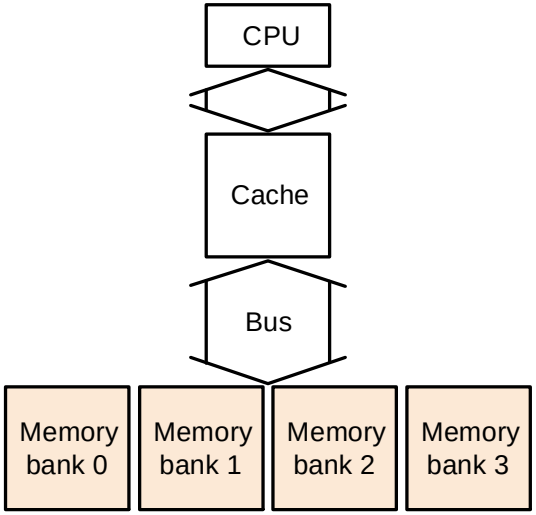
- Make reading multiple words easier by using banks of memory



a. One-word-wide memory organization



b. Wide memory organization



c. Interleaved memory organization

Example:

- 1 memory bus clock cycle to send address
- 15 memory bus clock cycles for each DRAM access initiated
- 1 memory bus clock cycle to send a word of data

If we have a cache block of four words,

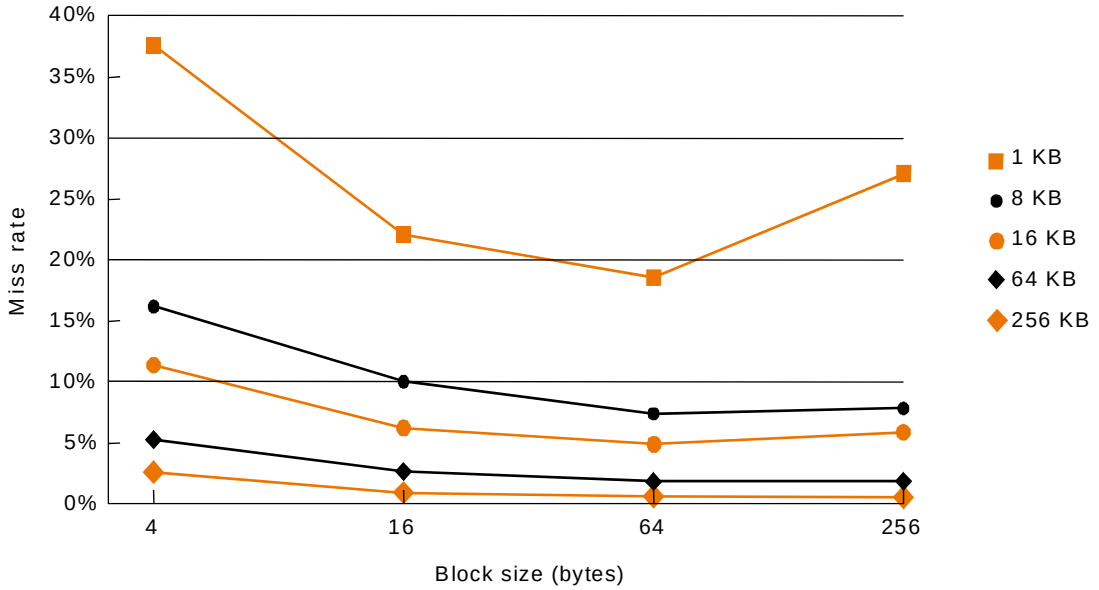
a. 1+4*15+4*1 = 65 memory bus clock cycles

b. 1+2*15+2*1 = 33 memory bus clock cycles

c. 1+1*15+4*1 = 20 memory bus clock cycles

Performance

- Increasing the block size tends to decrease miss rate:



Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

- Use split caches because there is more spatial locality in code:

Performance

- Simplified model:

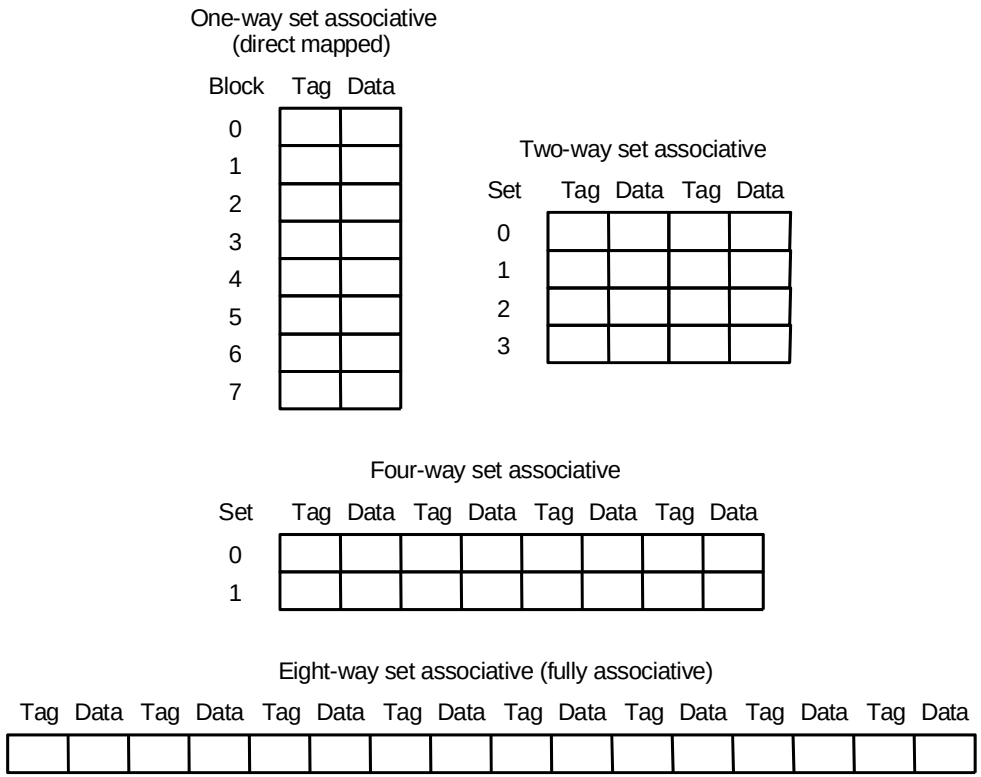
  execution time = (execution cycles + stall cycles) * cycle time

  stall cycles = # of instructions * miss ratio * miss penalty

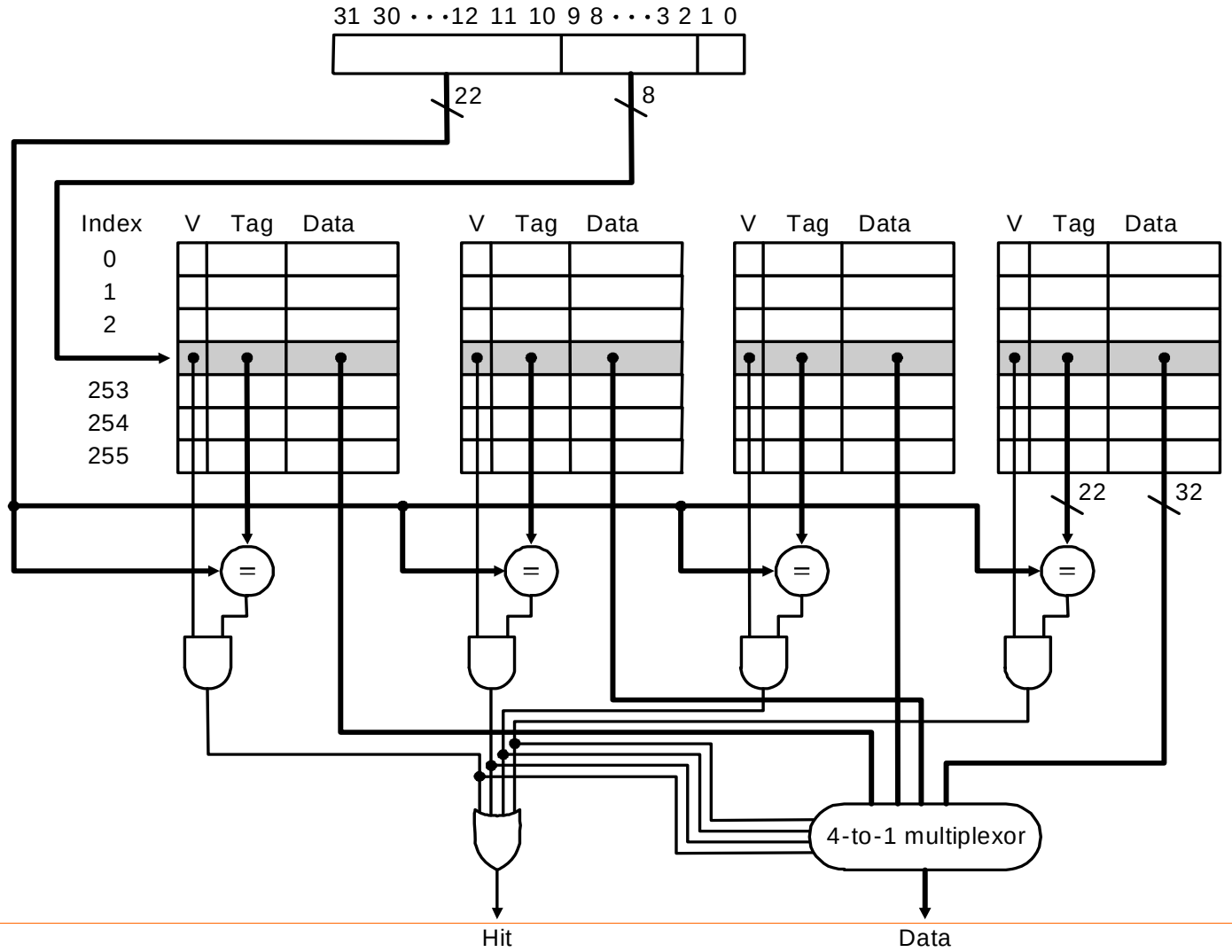
- Two ways of improving performance:
  - decreasing the miss ratio
  - decreasing the miss penalty

What happens if we increase block size?

Decreasing miss ratio with associativity



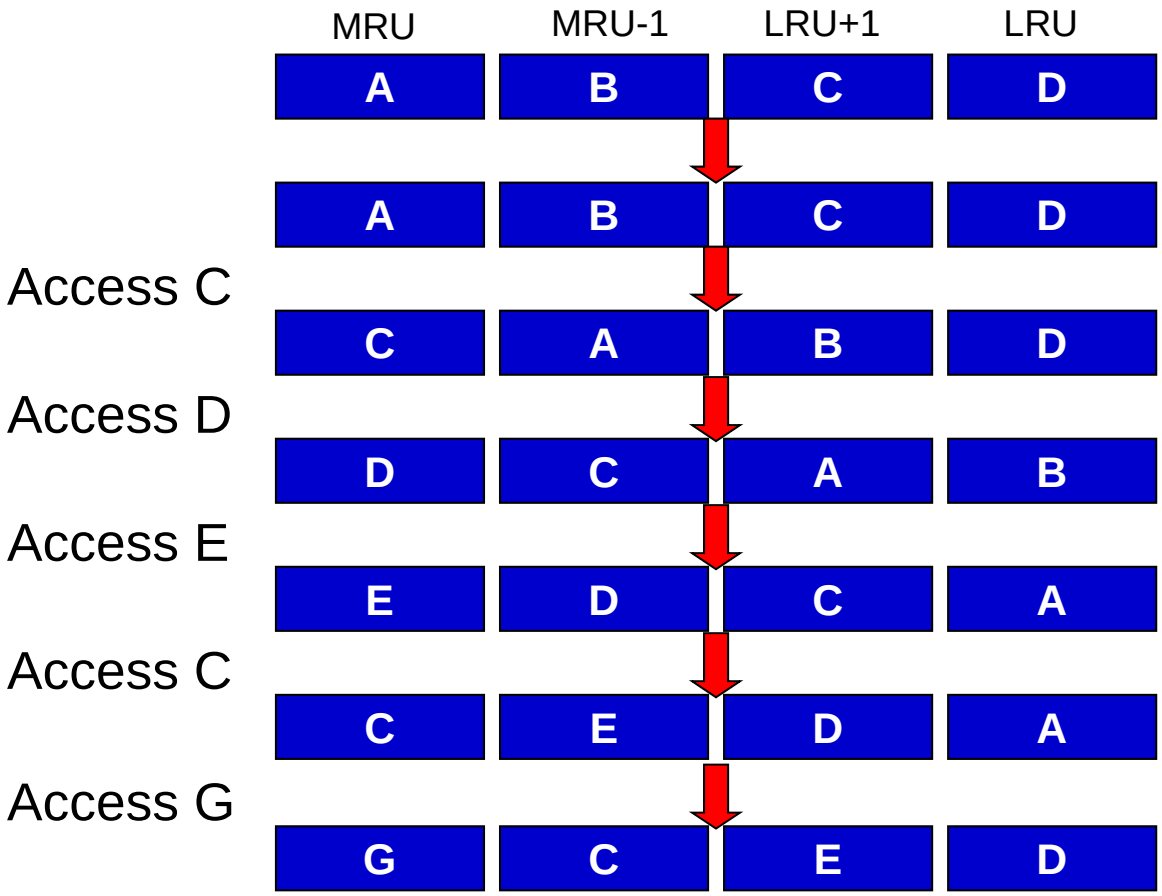
An implementation



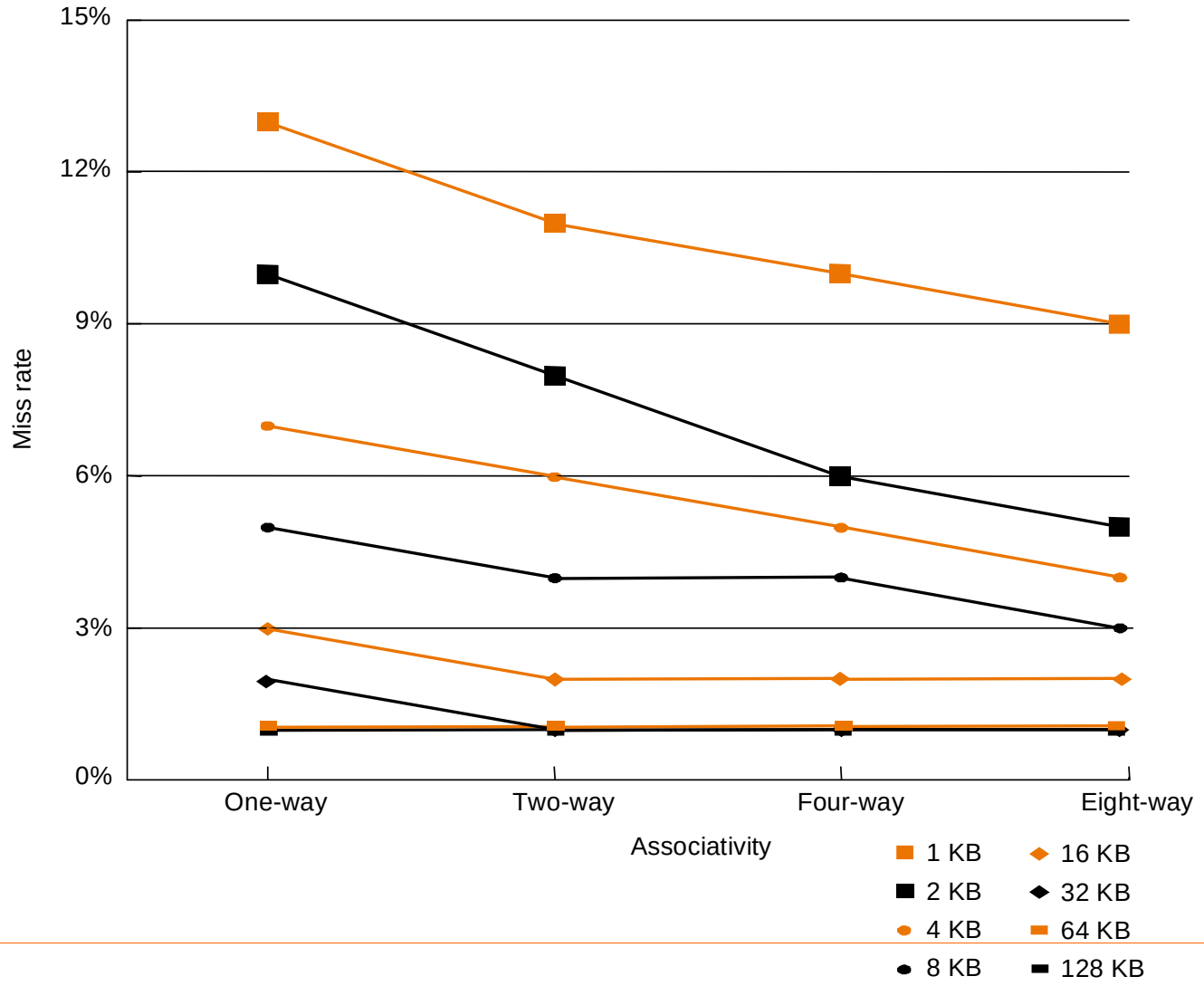
Set-Associative Cache

- Multiple cache blocks (lines) can be allocated into the same set
- When full, needs to evict some block out of the cache
- Need to consider the locality
- Replacement policy
  - Last-In First-Out (LIFO), like a stack
  - Random
  - First-In First-Out (FIFO), like a queue
  - Least Recently Used (LRU)

Least Recently Used (LRU)



Performance



Decreasing miss penalty with multilevel caches

- Add a second level cache:
  - often primary cache is on the same chip as the processor
  - use SRAMs to add another cache above primary memory (DRAM)
  - miss penalty goes down if data is in 2nd level cache
- Refer to previous example
- Using multilevel caches:
  - try and optimize the hit time on the 1st level cache
  - try and optimize the miss rate on the 2nd level cache

Some Issues

- Processor speeds continue to increase very fast
  — much faster than either DRAM or disk access times

- Design challenge: dealing with this growing disparity

- Trends:
  - synchronous SRAMs (provide a burst of data)
  - redesign DRAM chips to provide higher bandwidth or processing
  - restructure code to increase locality
  - use prefetching (make cache visible to ISA)

Direct-Mapped Cache Design Example

Given a 1KB direct-mapped writeback cache. The cache linesize is 128 bytes. Also assume your program addressable memory space is 4GB. All the entries inside the cache were initialized to 0. Given the following 5 memory references. Note that you only need to fill in those lines that are occupied by each instruction and leave blanks for those not accessed. Omit the data array since it is irrelevant and only show the valid (V) bit, dirty (D) bit and tag arrays. Use **Hexadecimal** number for tags.

read 0xBBCCD288

read 0xBBCCDF88

write 0xBBCCD300

write 0xBBCCDFFC

Read 0xBBCCD680

Set Associativity Cache Design Example

Given a 2KB 2-way writeback data cache. The cache line size is 256 bytes. Also assume your program addressable memory space is 4GB. All the entries inside the cache were initialized to 0. Given the following 5 memory references. Note that you only need to fill in those lines that are occupied by each instruction and leave blanks for those not accessed. When both ways are empty, Tag 0 has higher priority for being filled. Omit the data array since it is irrelevant and only show the valid (V) bit, dirty (D) bit and tag arrays. Use ***Hexadecimal*** *number for tags.*

read 0xBBCCD288

read 0xBBCCDF88

write 0xBBCCD300

write 0xBBCCDFFC

Read 0xBBCCD680