Caches (Writing)

Hakim Weatherspoon

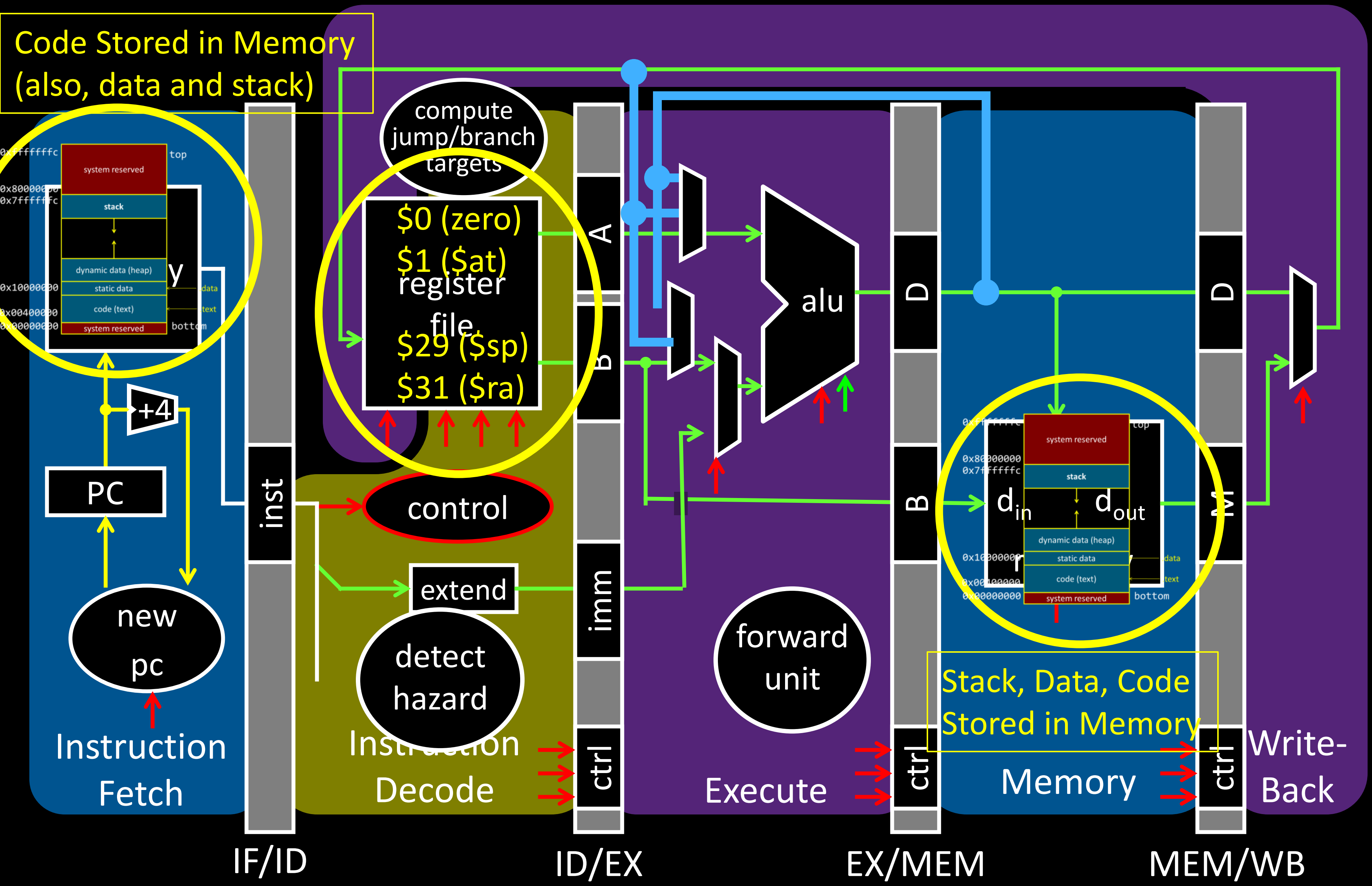
CS 3410, Spring 2013

Computer Science

Cornell University

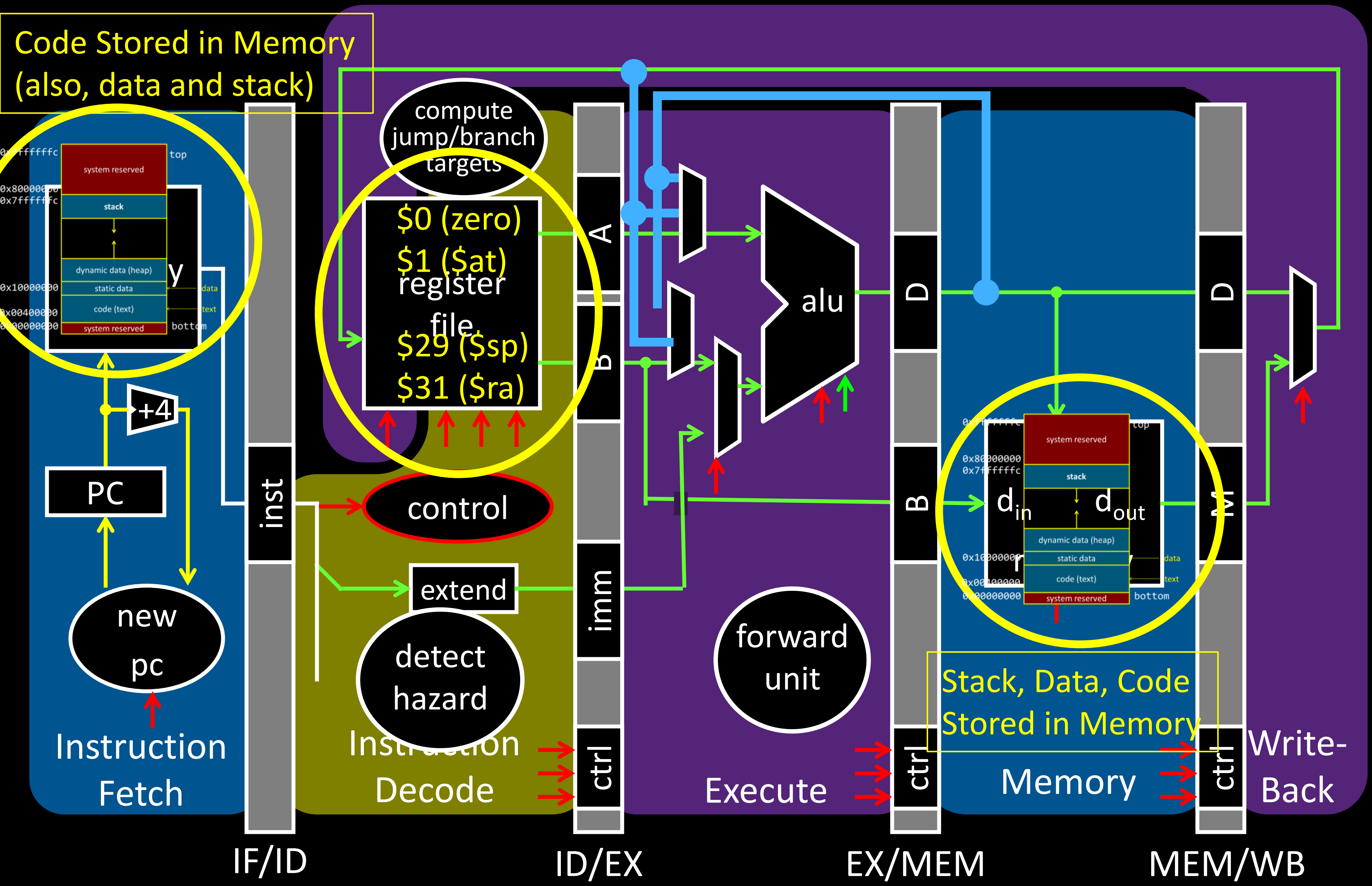
P & H Chapter 5.2-3, 5.5

Big Picture: Memory



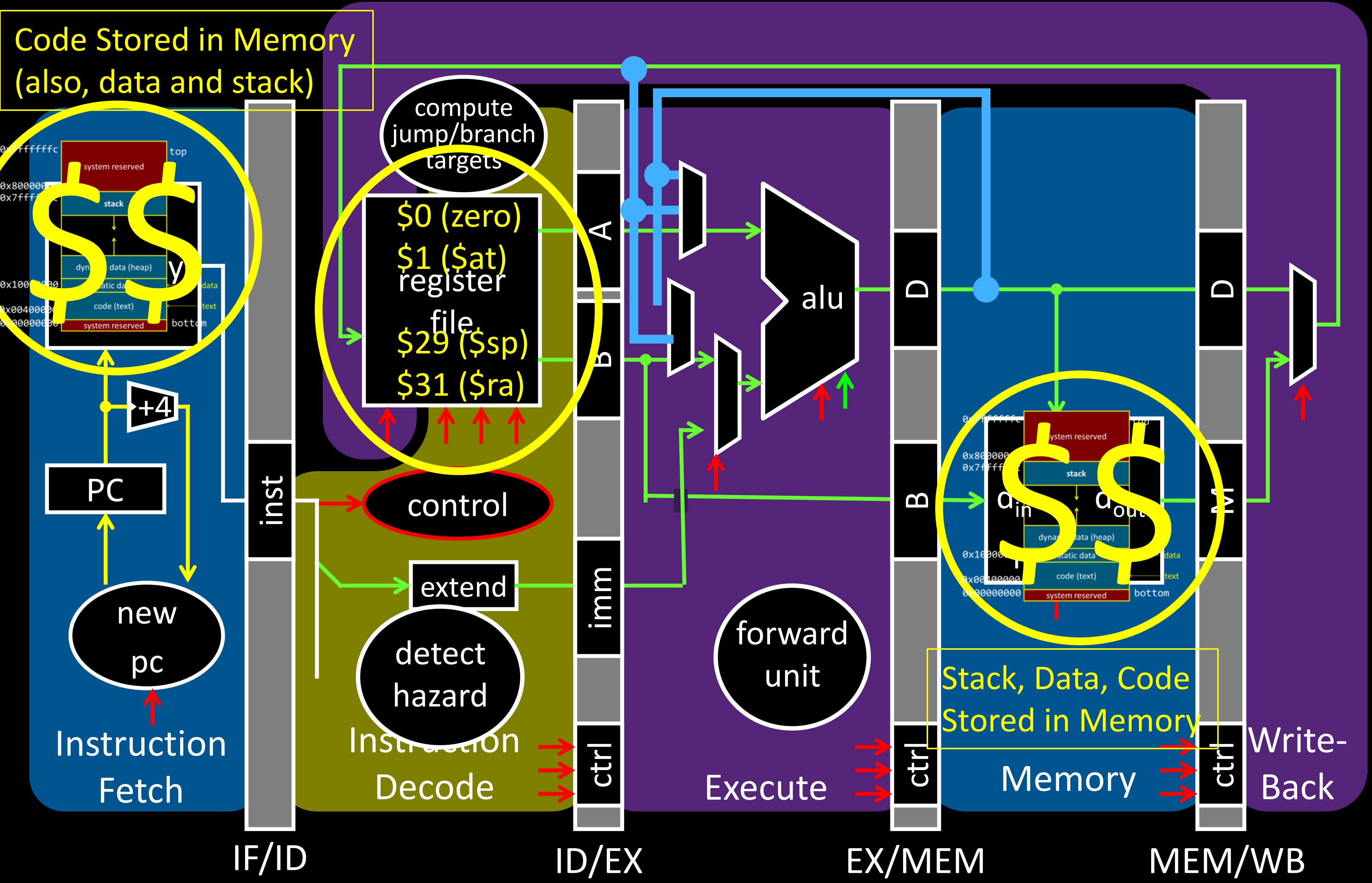
Big Picture: Memory

Memory: big & slow vs Caches: small & fast



Big Picture: Memory

Memory: big & slow vs Caches: small & fast



Big Picture

How do we make the processor fast,

Given that memory is VEEERRRYYYY SLLOOOWWW!!

Big Picture

How do we make the processor fast,
Given that memory is VEEERRRYYYY SLLOOOWWW!!

But, insight for Caches

If Mem[x] was accessed *recently*...
... then Mem[x] is likely to be accessed *soon*

- Exploit temporal locality:
  - Put recently accessed Mem[x] higher in memory hierarchy since it will likely be accessed again soon

... then Mem[$x \pm \varepsilon$] is likely to be accessed *soon*

- Exploit spatial locality:
  - Put entire block containing Mem[x] and surrounding addresses higher in memory hierarchy since nearby address will likely be accessed

Goals for Today: caches

Comparison of cache architectures:

- Direct Mapped
- Fully Associative
- N-way set associative

Writing to the Cache

- Write-through vs Write-back

Caching Questions

- How does a cache work?
- How effective is the cache (hit rate/miss rate)?
- How large is the cache?
- How fast is the cache (AMAT=average memory access time)

Next Goal

How do the different cache architectures compare?

- Cache Architecture Tradeoffs?
- Cache Size?
- Cache Hit rate/Performance?

Cache Tradeoffs

A given data block can be placed…

- … in any cache line → Fully Associative
- … in exactly one cache line → Direct Mapped
- … in a small set of cache lines → Set Associative

Cache Tradeoffs

Direct Mapped		Fully Associative
+ Smaller	Tag Size	Larger –
+ Less	SRAM Overhead	More –
+ Less	Controller Logic	More –
+ Faster	Speed	Slower –
+ Less	Price	More –
+ Very	Scalability	Not Very –
– Lots	# of conflict misses	Zero +
– Low	Hit rate	High +
– Common	Pathological Cases?	?

Cache Tradeoffs

Compromise: Set-associative cache

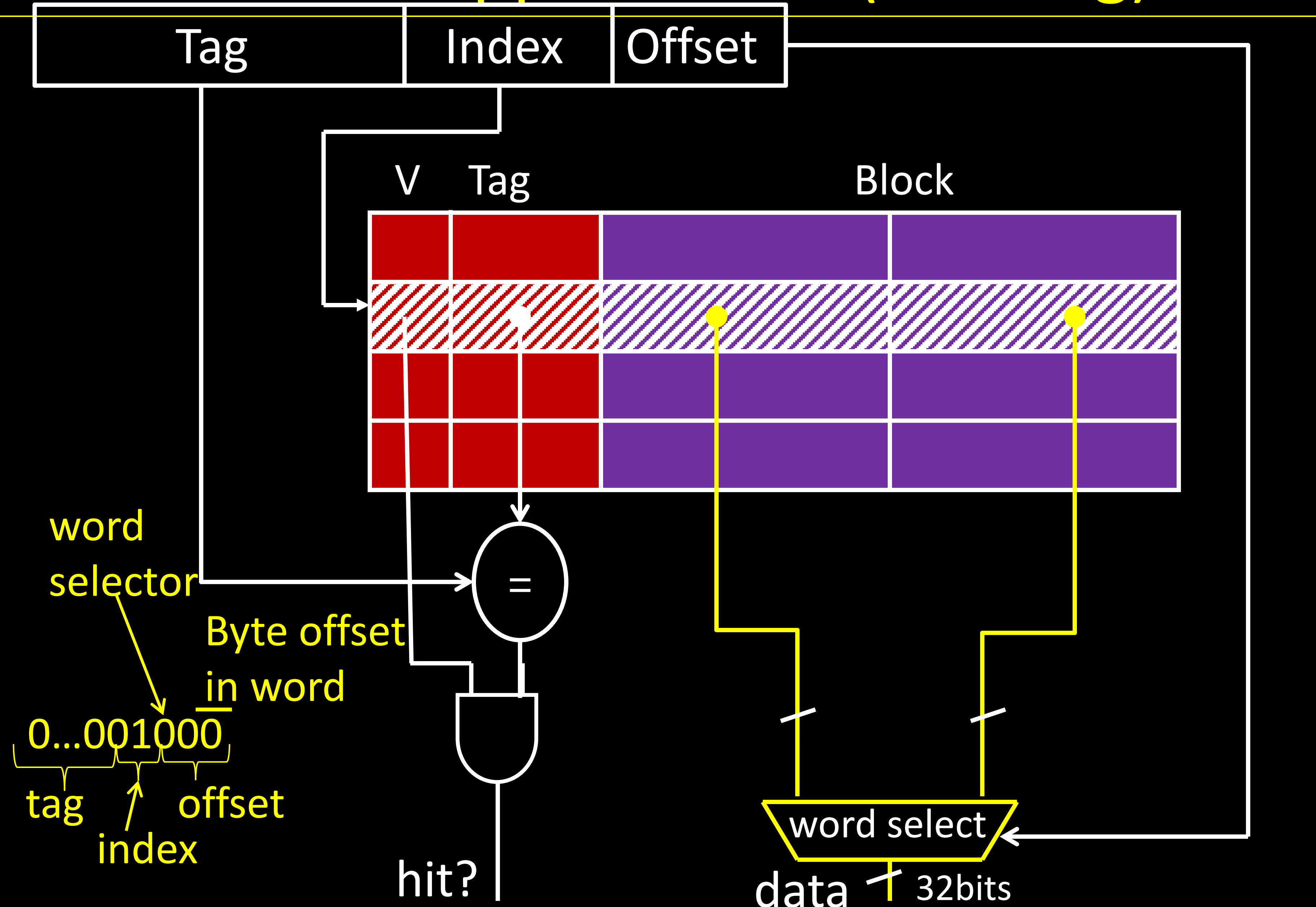
Like a direct-mapped cache

- Index into a location
- Fast

Like a fully-associative cache

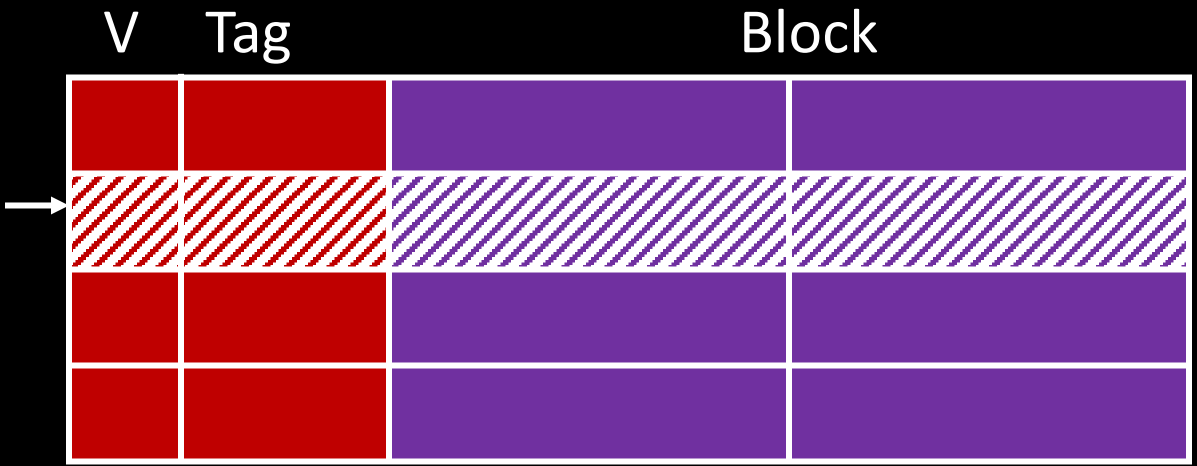
- Can store multiple entries
  - decreases thrashing in cache
- Search in each element

Direct Mapped Cache (Reading)



Direct Mapped Cache (Reading)

Tag	Index	Offset

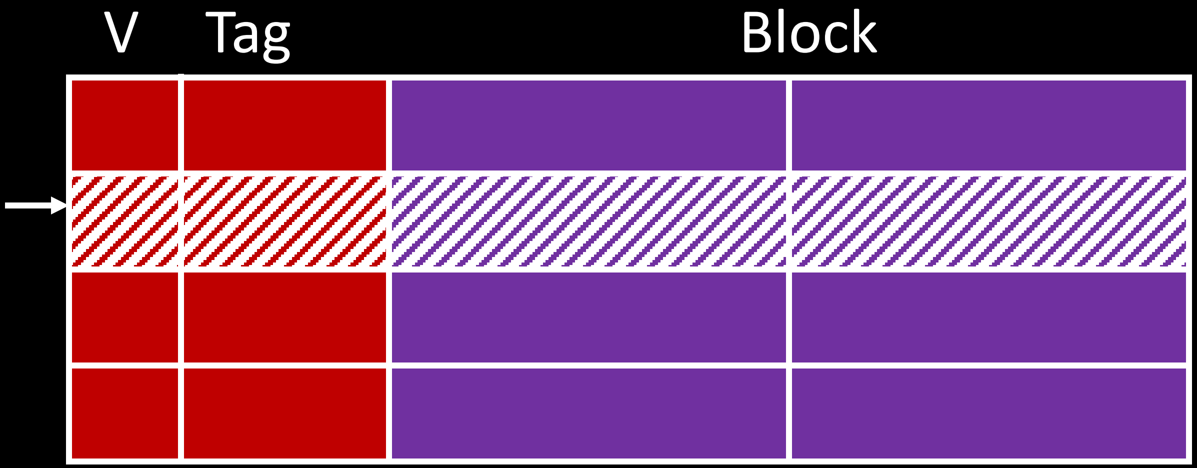


n bit index, *m* bit offset

Q: How big is cache (***data only***)?

Direct Mapped Cache (Reading)

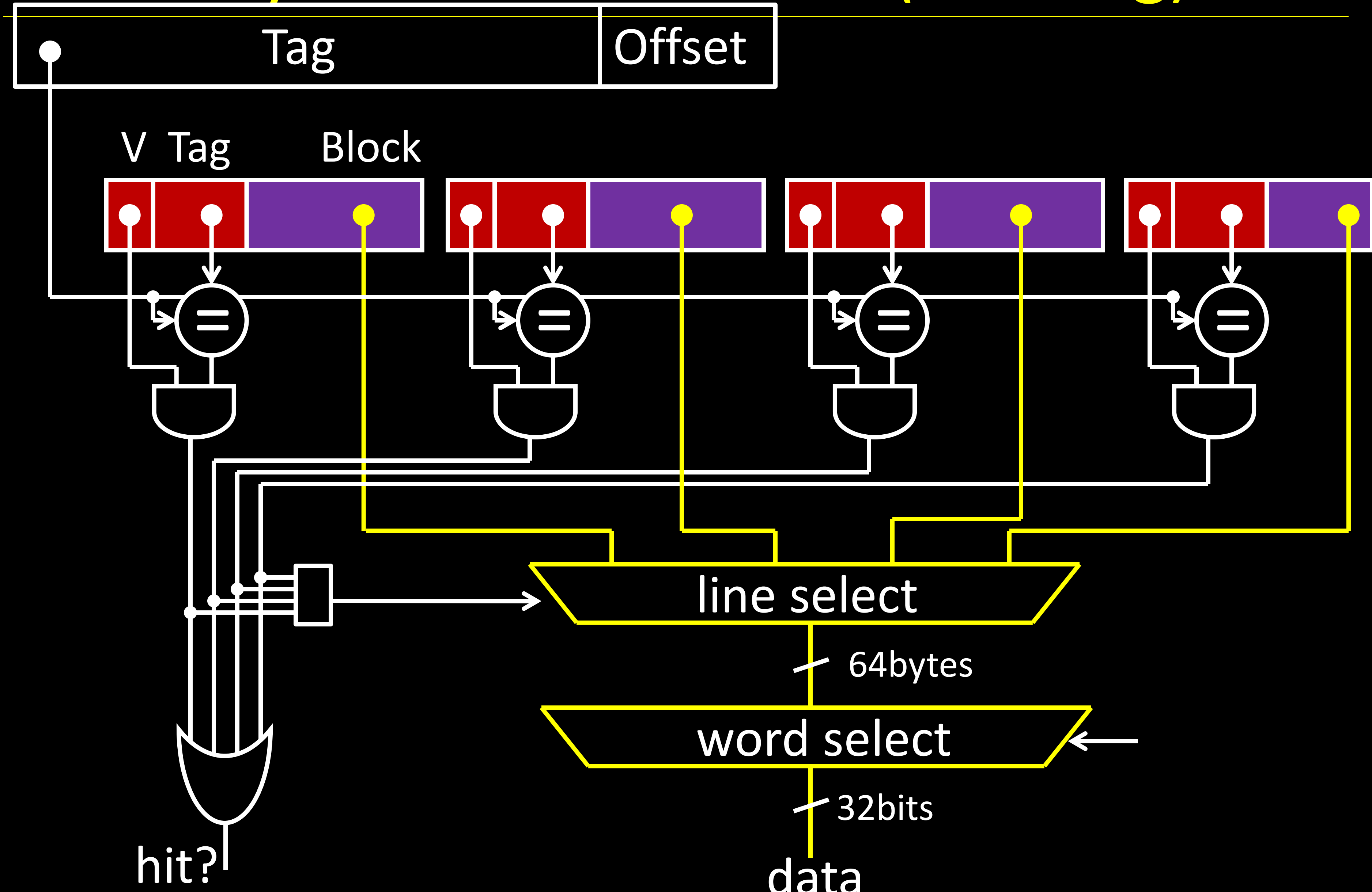
Tag	Index	Offset



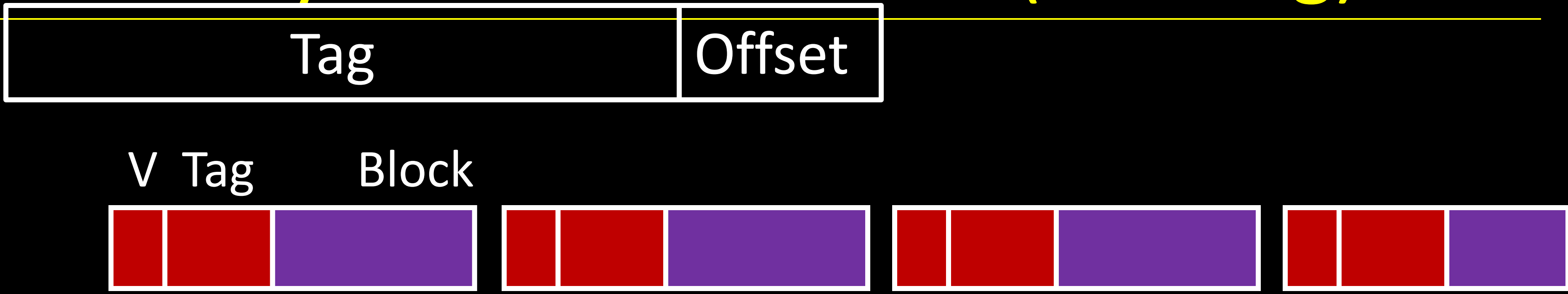
n bit index, *m* bit offset

Q: How much SRAM is needed (***data + overhead***)?

Fully Associative Cache (Reading)



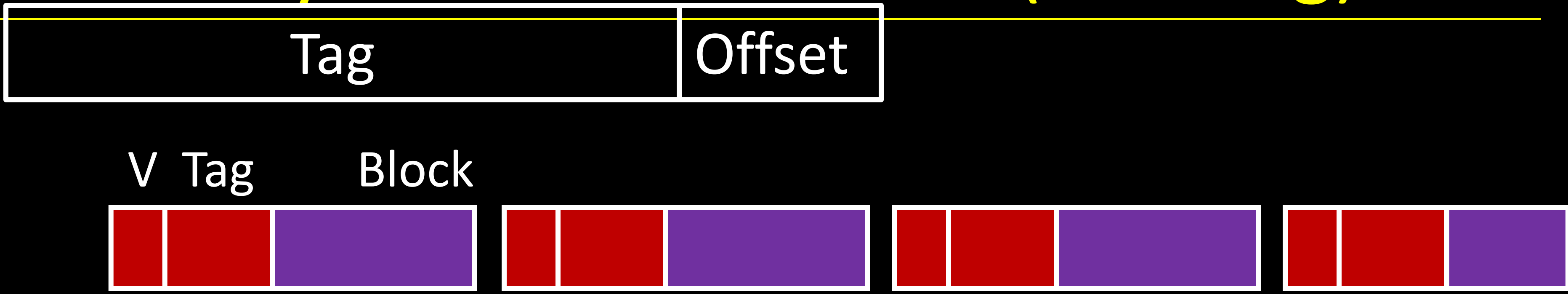
Fully Associative Cache (Reading)



m bit offset , 2^n blocks (cache lines)

Q: How big is cache (**data only**)?

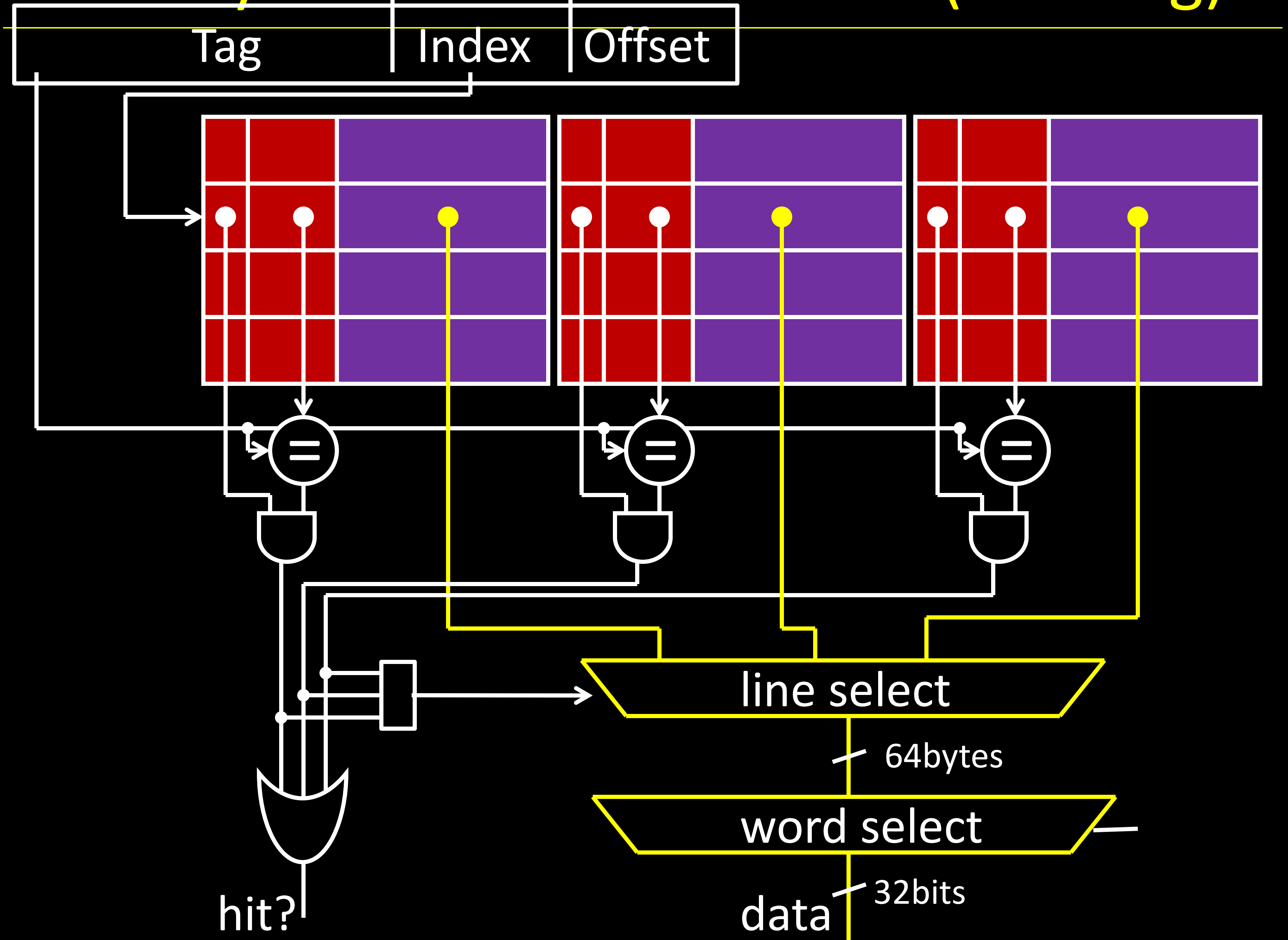
Fully Associative Cache (Reading)



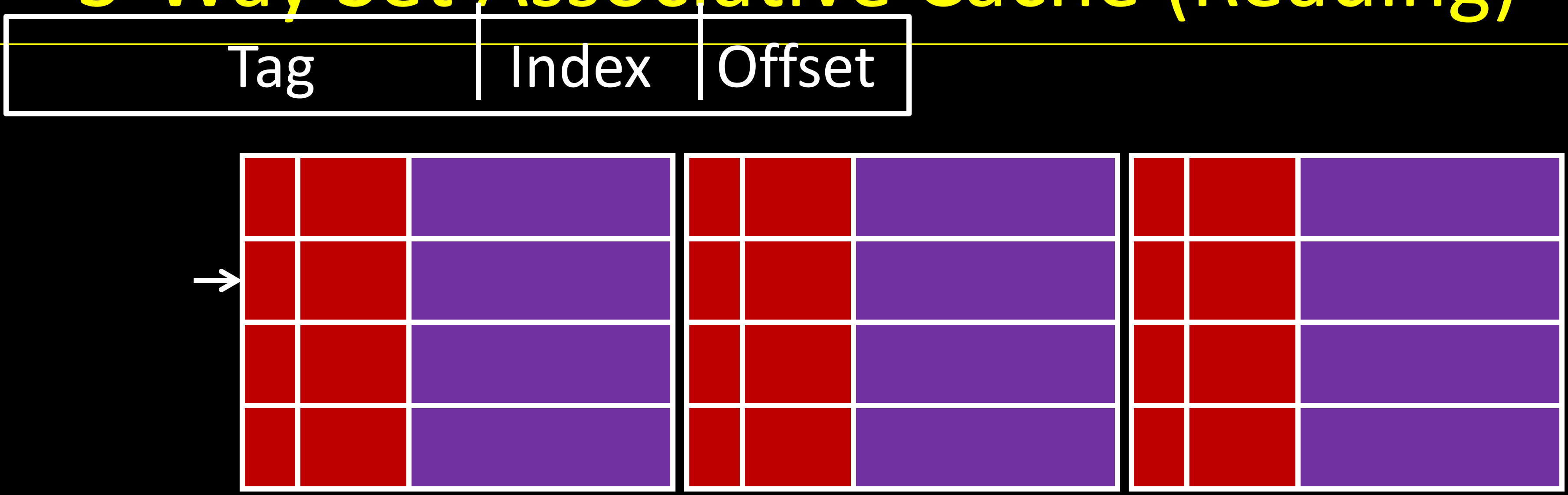
m bit offset , 2^n blocks (cache lines)

Q: How much SRAM needed (data + overhead)?

3-Way Set Associative Cache (Reading)



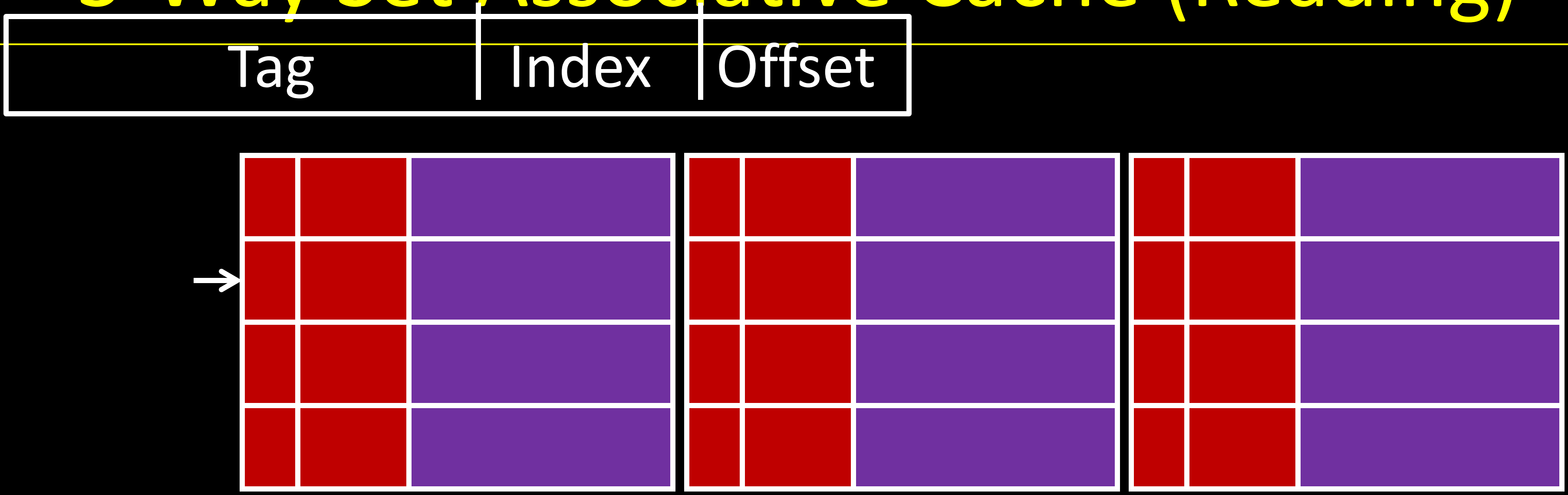
3-Way Set Associative Cache (Reading)



n bit index, *m* bit offset, **N-way Set Associative**

Q: How big is cache (***data only***)?

3-Way Set Associative Cache (Reading)

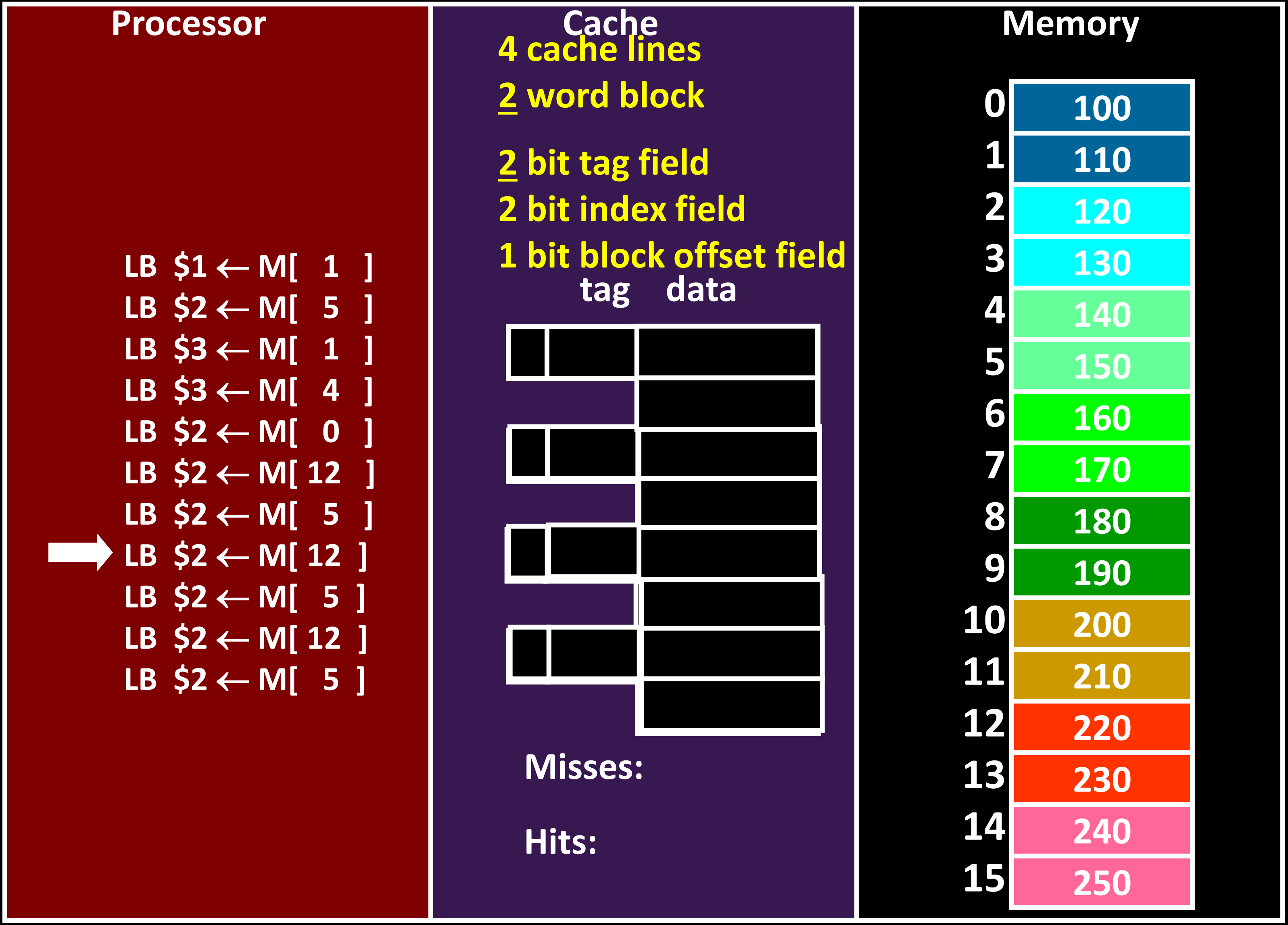


n bit index, *m* bit offset, **N-way Set Associative**

Q: How much SRAM is needed (***data + overhead***)?

Comparison: Direct Mapped

Using **byte addresses** in this example! Addr Bus = 5 bits



4 cache lines
2 word block

2 bit tag field
2 bit index field
1 bit block offset field

```
LB $1 ← M[ 1 ]
LB $2 ← M[ 5 ]
LB $3 ← M[ 1 ]
LB $3 ← M[ 4 ]
LB $2 ← M[ 0 ]
LB $2 ← M[ 12 ]
LB $2 ← M[ 5 ]
LB $2 ← M[ 12 ]
LB $2 ← M[ 5 ]
LB $2 ← M[ 12 ]
LB $2 ← M[ 5 ]
```

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Comparison: Fully Associative

Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

```
   LB  $1 ← M[  1  ]
   LB  $2 ← M[  5  ]
   LB  $3 ← M[  1  ]
   LB  $3 ← M[  4  ]
   LB  $2 ← M[  0  ]
   LB  $2 ← M[ 12  ]
   LB  $2 ← M[  5  ]
⇨  LB  $2 ← M[ 12 ]
   LB  $2 ← M[  5 ]
   LB  $2 ← M[ 12 ]
   LB  $2 ← M[  5 ]
```

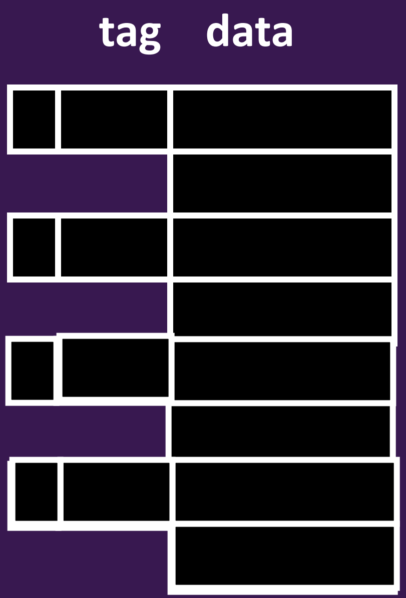
Cache

4 cache lines

2 word block

4 bit tag field

1 bit block offset field



Misses:

Hits:

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Comparison: 2 Way Set Assoc

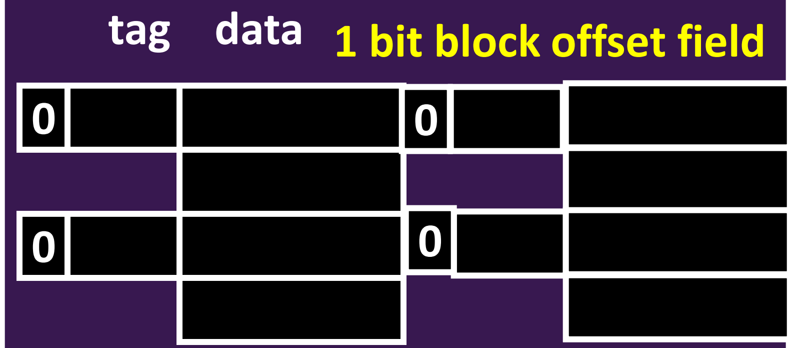
Using **byte addresses** in this example! Addr Bus = 5 bits

Processor

```
LB  $1 ← M[  1  ]
LB  $2 ← M[  5  ]
LB  $3 ← M[  1  ]
LB  $3 ← M[  4  ]
LB  $2 ← M[  0  ]
LB  $2 ← M[ 12  ]
LB  $2 ← M[  5  ]
LB  $2 ← M[  12 ]
LB  $2 ← M[  5 ]
LB  $2 ← M[ 12  ]
LB  $2 ← M[  5  ]
```

Cache

2 sets
2 word block
3 bit tag field
1 bit set index field
1 bit block offset field



Misses:

Hits:

Memory

0	100
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Misses

Cache misses: classification

The line is being referenced for the first time

- Cold (aka Compulsory) Miss

The line was in the cache, but has been evicted…

… because some other access with the same index

- Conflict Miss

… because the cache is too small

- i.e. the *working set* of program is larger than the cache
- Capacity Miss

Misses

Cache misses: classification

Cold (aka Compulsory)

- The line is being referenced for the first time

Capacity

- The line was evicted because the cache was too small
- i.e. the *working set* of program is larger than the cache

Conflict

- The line was evicted because of another access whose index conflicted

Cache Performance

Average Memory Access Time (AMAT)

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped

  Data cost: 3 cycle per word access

  Lookup cost: 2 cycle

Mem (DRAM): 4GB

  Data cost: 50 cycle per word, plus 3 cycle per consecutive word

Performance depends on:

  Access time for hit, miss penalty, hit rate

Takeway

Direct Mapped → simpler, low hit rate

Fully Associative → higher hit cost, higher hit rate

N-way Set Associative → middleground

Writing with Caches

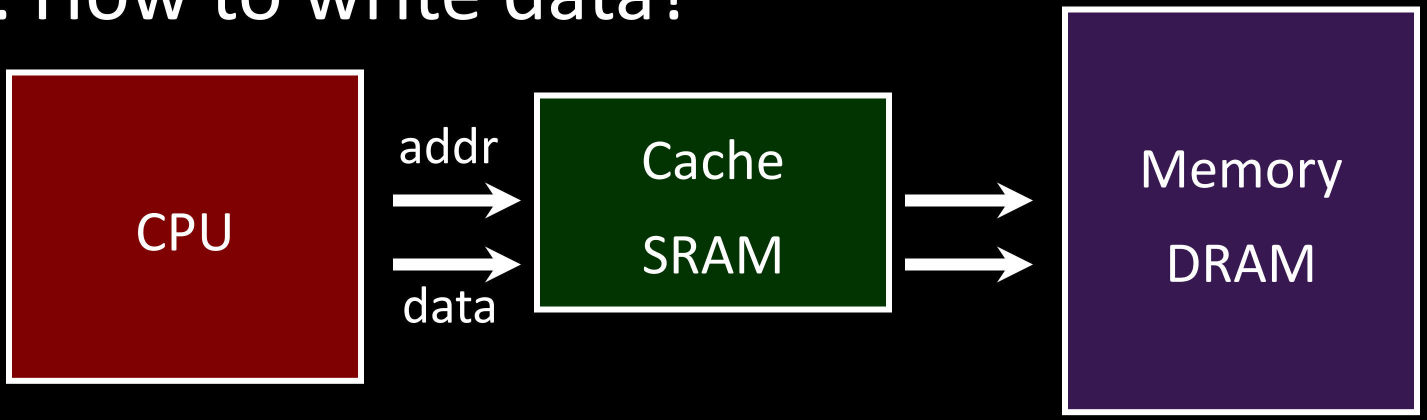
Eviction

Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped
  - no choice, must evict line selected by index
- Associative caches
  - random: select one of the lines at random
  - round-robin: similar to random
  - FIFO: replace oldest line
  - LRU: replace line that has not been used in the longest time

Cached Write Policies

Q: How to write data?



If data is already in the cache…

No-Write

- writes invalidate the cache and go directly to memory

Write-Through

- writes go to main memory and cache

Write-Back

- CPU writes only to cache
- cache writes to main memory later (when block is evicted)

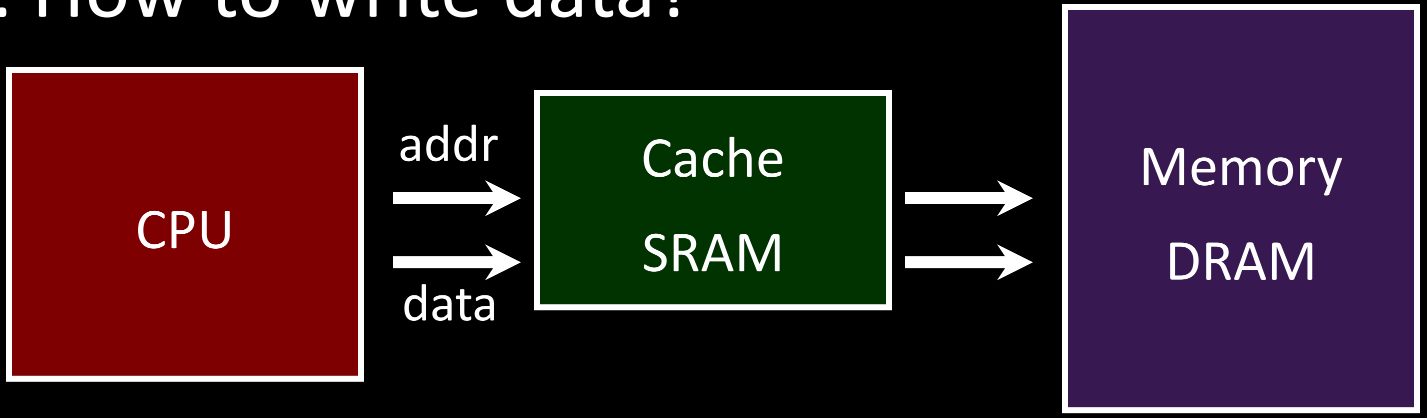
What about Stores?

Where should you write the result of a store?

- If that memory location is in the cache?
  - Send it to the cache
  - Should we also send it to memory right away? (write-through policy)
  - Wait until we kick the block out (write-back policy)
- If it is not in the cache?
  - Allocate the line (put it in the cache)? (write allocate policy)
  - Write it directly to memory without allocation? (no write allocate policy)

Write Allocation Policies

Q: How to write data?



If data is not in the cache…

Write-Allocate

- allocate a cache line for new data (and maybe write-through)

No-Write-Allocate

- ignore cache, just go to main memory

Handling Stores (Write-Through)

Using **byte addresses** in this example! Addr Bus = 4 bits

Processor

Assume write-allocate policy

```
LB  $1 ← M[  1  ]
LB  $2 ← M[  7  ]
SB  $2 → M[  0  ]
SB  $1 → M[  5  ]
LB  $2 ← M[  10 ]
SB  $1 → M[  5  ]
SB  $1 → M[  10 ]
```

$0	
$1	
$2	
$3	

Cache

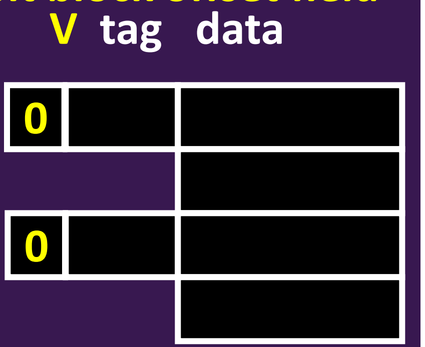
Fully Associative Cache

2 cache lines

2 word block

3 bit tag field

1 bit block offset field



Misses: 0

Hits: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

How Many Memory References?

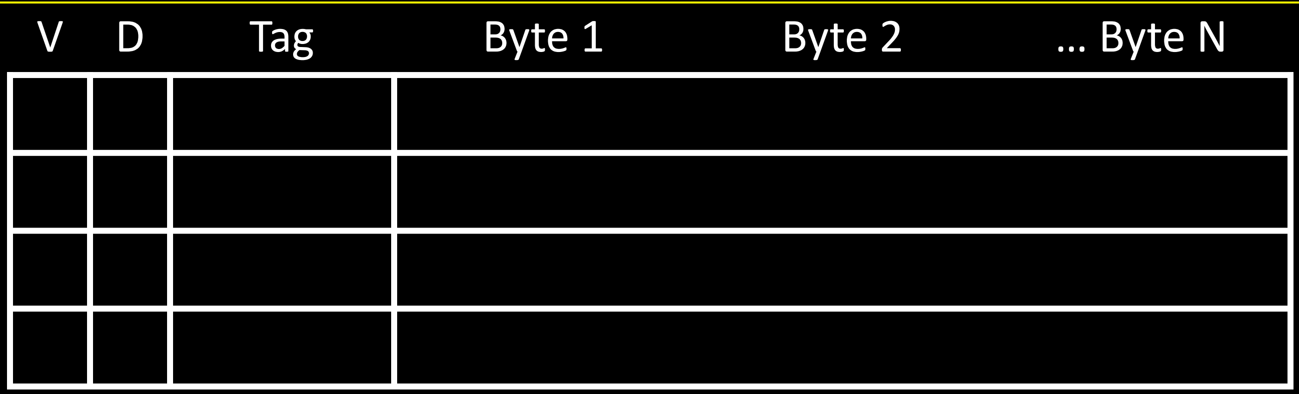
Write-through performance

Write-Through vs. Write-Back

Can we also design the cache NOT to write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is evicted (write-back policy)
- Do we need to write-back all evicted lines?
- No, only blocks that have been stored into (written)

Write-Back Meta-Data



V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

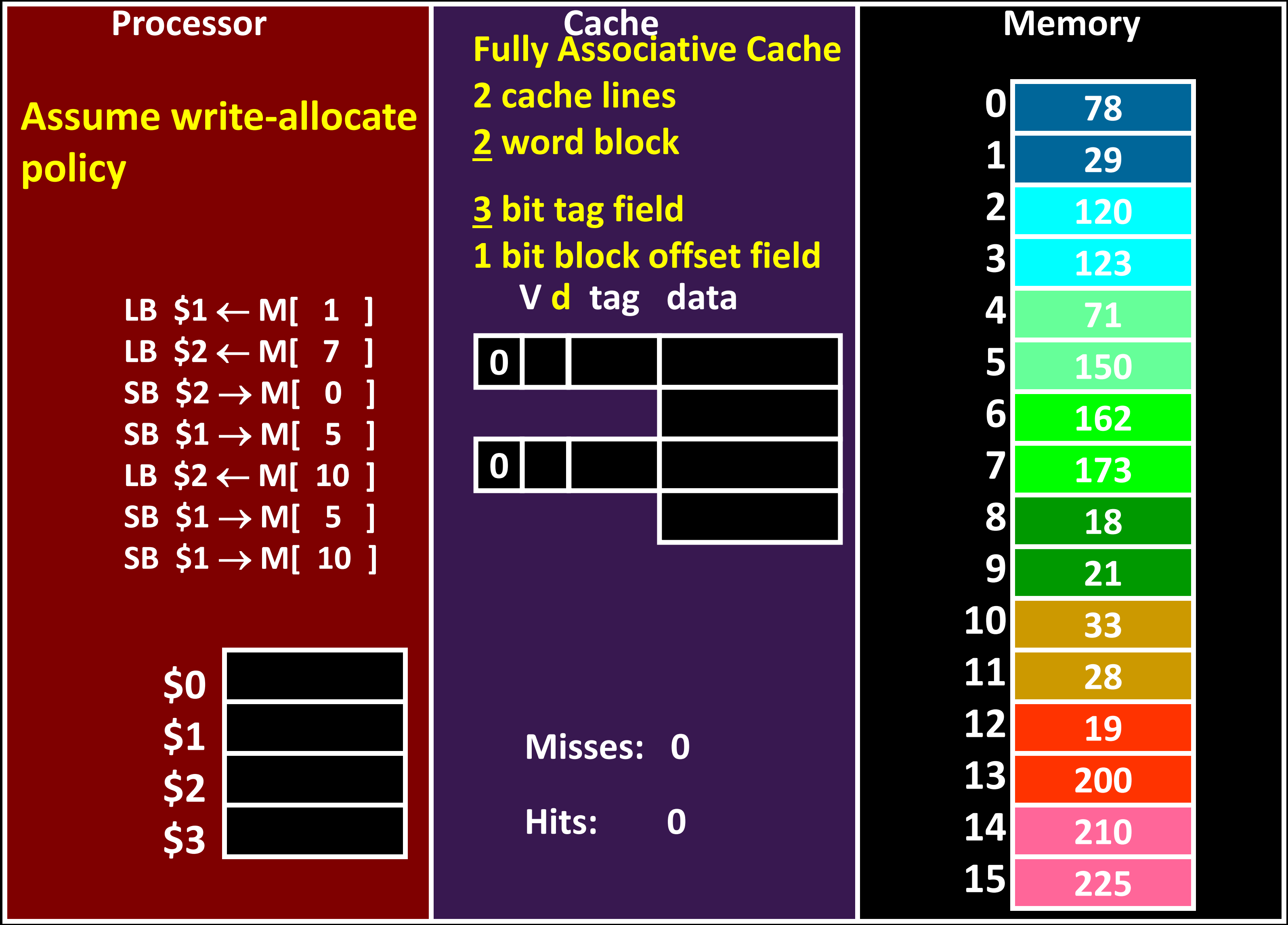
- Set D = 1

When evicting line:

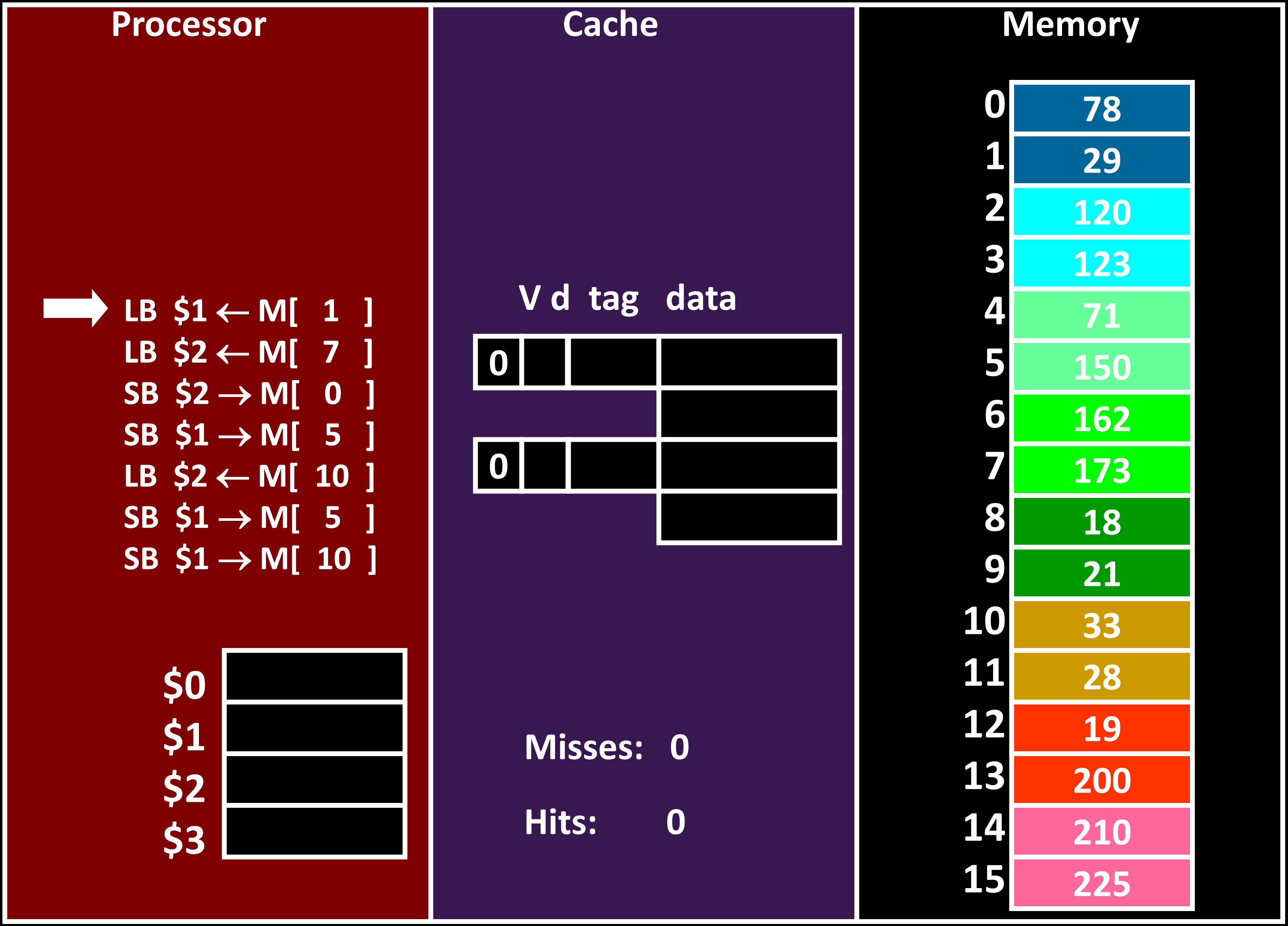
- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0

Handling Stores (Write-Back)

Using **byte addresses** in this example! Addr Bus = 4 bits



Write-Back (REF 1)



How Many Memory References?

Write-back performance

Write-through vs. Write-back

Write-through is slower

- But cleaner (memory always consistent)

Write-back is faster

- But complicated when multi cores sharing memory

Performance: An Example

Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```
for (i=1; i<n; i++)
           A[0] += A[i];
```

```
for (i=0; i<n; i++)
           B[i] = A[i]
```

Performance Tradeoffs

Q: Hit time: write-through vs. write-back?

Q: Miss penalty: write-through vs. write-back?

Write Buffering

Q: Writes to main memory are **slow!**

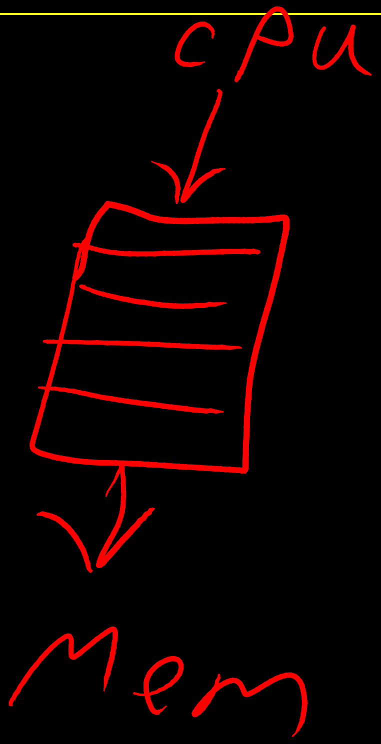
A: Use a write-back buffer

- A small queue holding dirty lines
- Add to end upon eviction
- Remove from front upon completion

Q: What does it help?

A: short bursts of writes (but not sustained writes)

A: fast eviction reduces miss penalty



Write-through vs. Write-back

Write-through is slower

- But simpler (memory always consistent)

Write-back is almost always faster

- write-back buffer hides large eviction cost
- But what about multiple cores with separate caches but sharing memory?

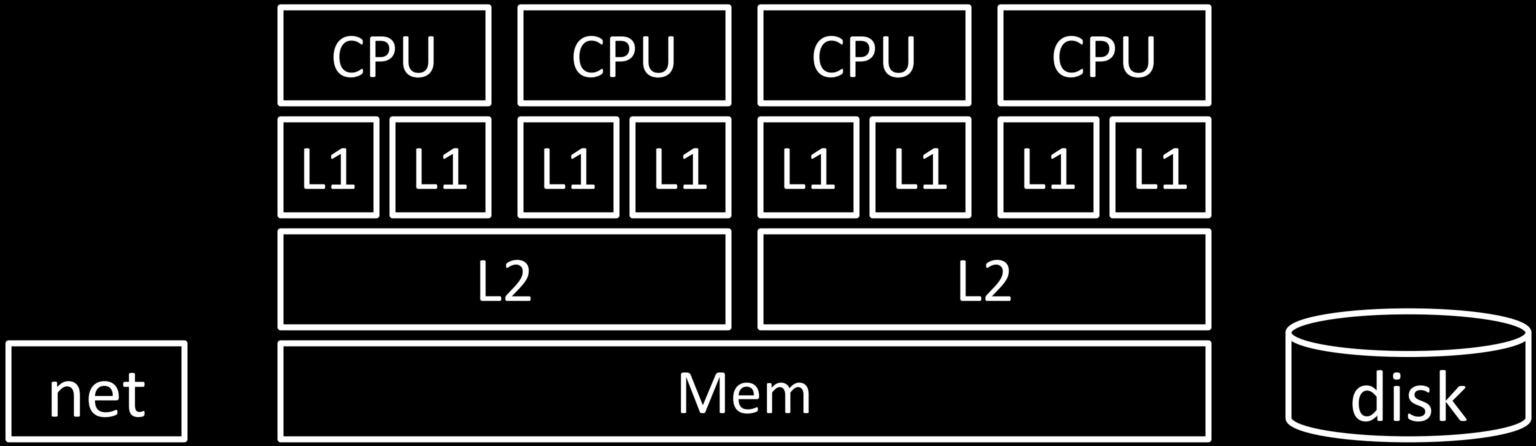
Write-back requires a cache coherency protocol

- Inconsistent views of memory
- Need to “snoop” in each other’s caches
- Extremely complex protocols, very hard to get right

Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



Cache coherency protocol

- May need to snoop on other CPU's cache activity
- Invalidate cache line when other CPU writes
- Flush write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right

Administrivia

Prelim1: ***Thursday***, March 28th in evening

- Time: We will start at ***7:30pm sharp***, so come early
- **Two Location: PHL101 and UPSB17**
  - **If NetID ends with even number, then go to PHL101 (Phillips Hall rm 101)**
  - **If NetID ends with odd number, then go to UPSB17 (Upson Hall rm B17)**
- Prelim Review: Yesterday, Mon, at 7pm and today, Tue, at 5:00pm.
  Both in Upson Hall rm B17

- Closed Book: ***NO NOTES, BOOK, ELECTRONICS, CALCULATOR, CELL PHONE***
- Practice prelims are online in CMS
- Material covered everything up to end of ***week before spring break***
  - Lecture: Lectures 9 to 16 (new since last prelim)
  - Chapter 4: Chapters 4.7 (Data Hazards) and 4.8 (Control Hazards)
  - Chapter 2: Chapter 2.8 and 2.12 (Calling Convention and Linkers), 2.16 and 2.17 (RISC and CISC)
  - Appendix B: B.1 and B.2 (Assemblers), B.3 and B.4 (linkers and loaders), and B.5 and B.6 (Calling Convention and process memory layout)
  - Chapter 5: 5.1 and 5.2 (Caches)
  - HW3, Project1 and Project2

Administrivia

Next six weeks

- Week 9 (May 25): Prelim2
- Week 10 (Apr 1): Project2 due and Lab3 handout
- Week 11 (Apr 8): Lab3 due and Project3/HW4 handout
- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due

Summary

Caching assumptions

- small working set: 90/10 rule
- can predict future: spatial & temporal locality

Benefits

- (big & fast) built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate

Summary

Memory performance matters!

- often more than CPU performance
- … because it is the bottleneck, and not improving much
- … because most programs move a LOT of data

Design space is huge

- Gambling against program behavior
- Cuts across all layers:
  users $\rightarrow$ programs $\rightarrow$ os $\rightarrow$ hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right