

# Caches

**Kevin Walsh**  
**CS 3410, Spring 2010**  
Computer Science  
Cornell University

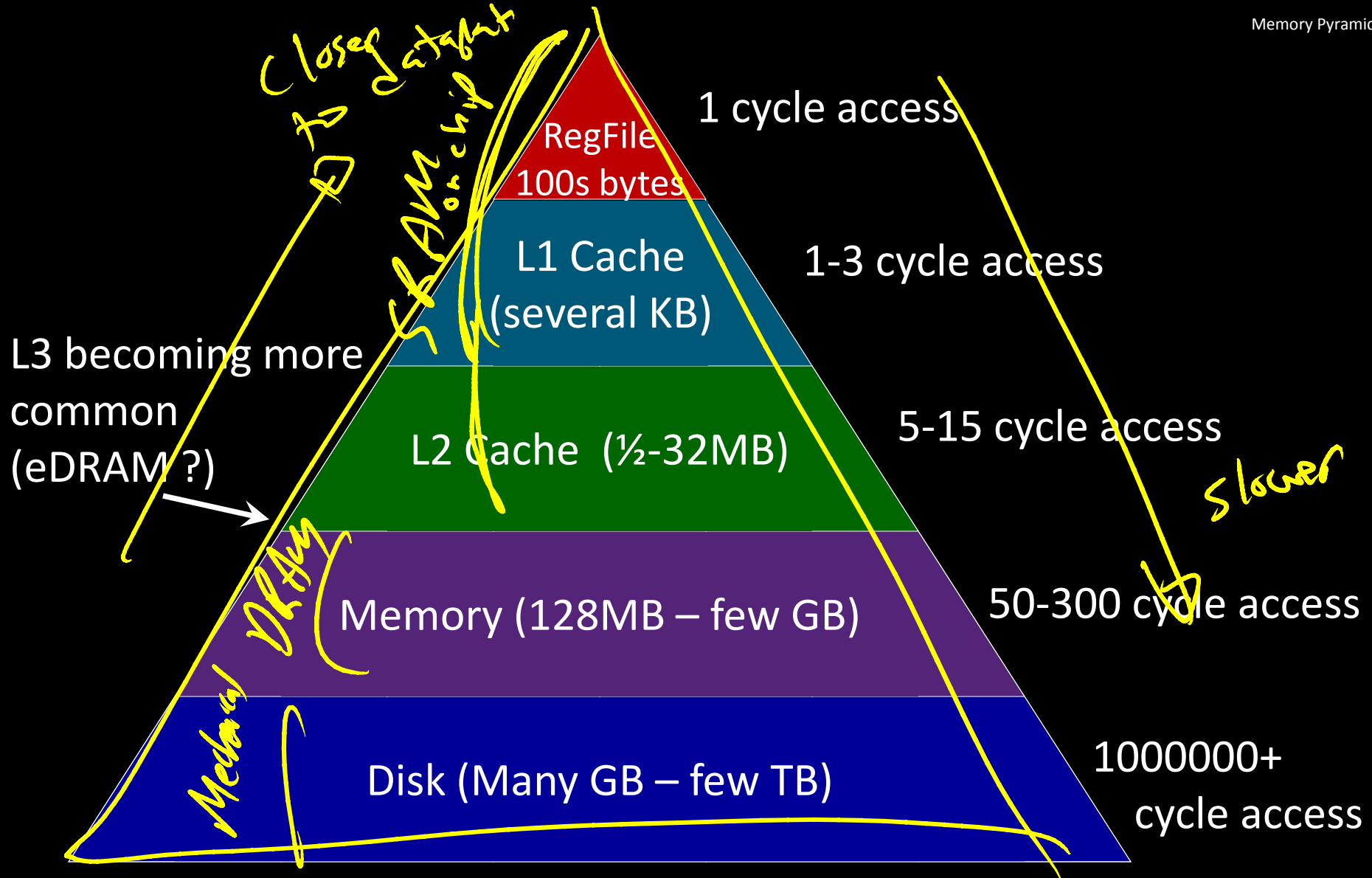
P & H Chapter 5.1, 5.2 (except writes)

CPU clock rates  $\sim 0.2\text{ns} - 2\text{ns}$  (5GHz-500MHz)

Technology	Capacity	\$/GB	Latency
Tape	1 TB	\$.17	100s of seconds
Disk	1 TB	\$.08	Millions cycles (ms)
SSD (Flash)	128GB	\$3	Thousands of cycles (us)
DRAM	4GB	\$25	50-300 cycles (10s of ns)
SRAM off-chip	4MB	\$4k	5-15 cycles (few ns)
SRAM on-chip	256 KB	???	1-3 cycles (ns)

Others: *embedded* **eDRAM** aka **1-T SRAM**, FeRAM, CD, DVD, ...

Q: Can we create illusion of cheap + large + fast?



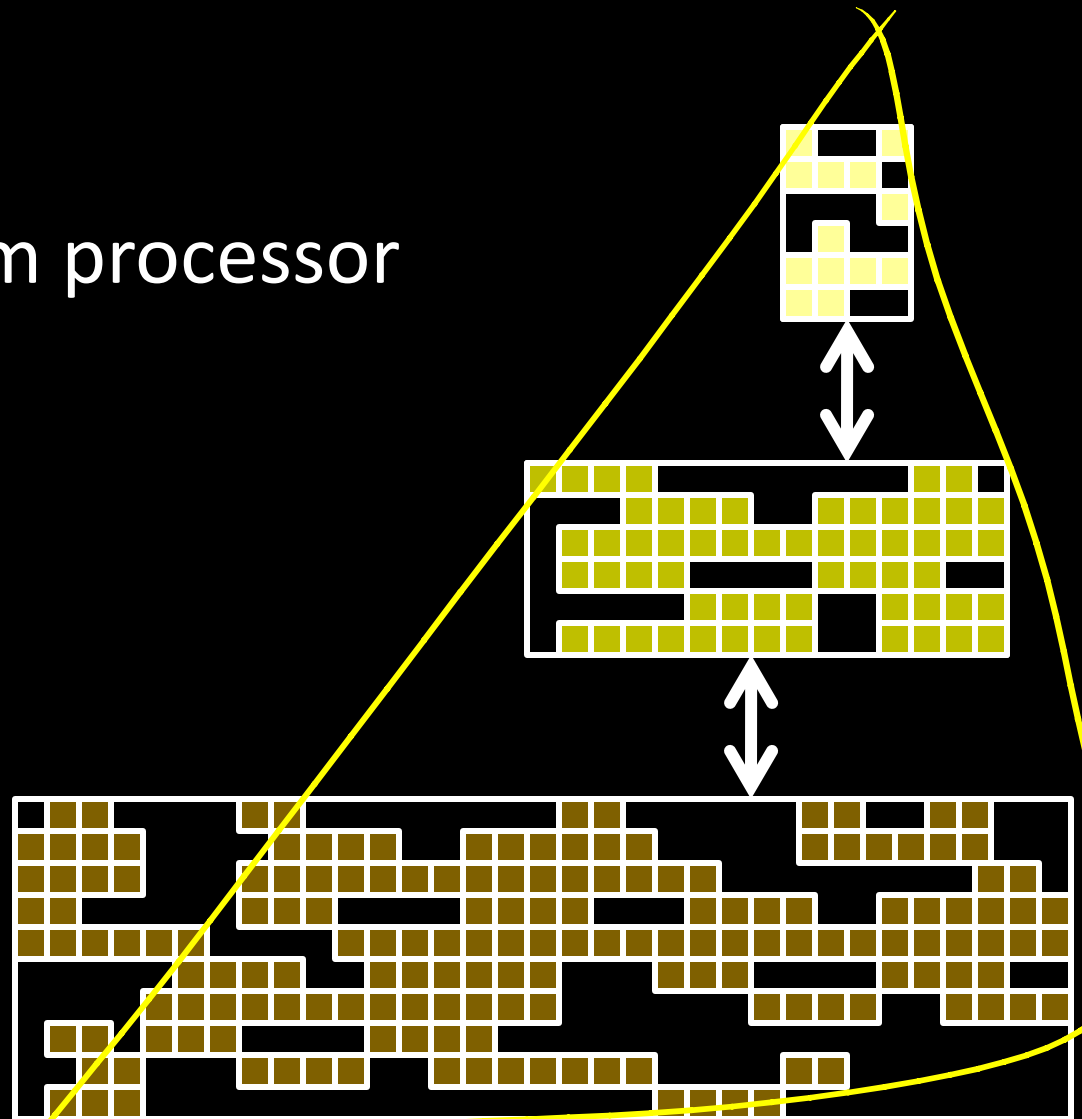
These are rough numbers: mileage may vary for latest/greatest  
Caches usually made of SRAM (or eDRAM)

# Memory closer to processor

- small & fast
- stores active data

# Memory farther from processor

- big & slow
- stores inactive data



Assumption: Most data is not active.

90/10

Q: How to decide what is active?

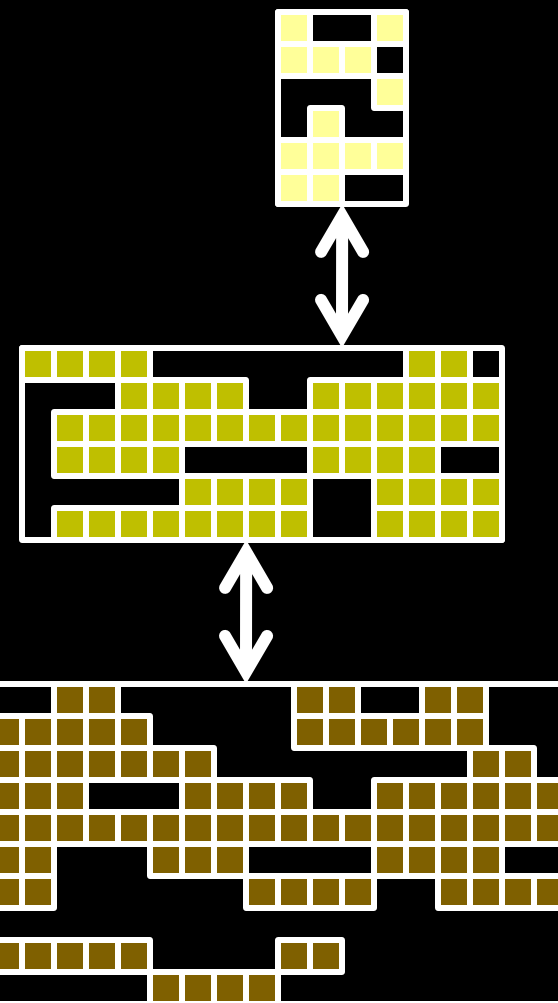
A: Some committee decides

A: Programmer decides *Static*

A: Compiler decides

A: OS decides at run-time

A: Hardware decides  
at run-time



Q: What is “active” data? *frequency of loads*

A: Data that will be used *soon.*

If  $\text{Mem}[x]$  is was accessed *recently*...

... then  $\text{Mem}[x]$  is likely to be accessed *soon*

- Caches exploit **temporal locality** by putting recently accessed  $\text{Mem}[x]$  higher in the pyramid

... then  $\text{Mem}[x \pm \epsilon]$  is likely to be accessed *soon*

- Caches exploit **spatial locality** by putting an entire block containing  $\text{Mem}[x]$  higher in the pyramid

# Memory trace

0x7c9a2b18  
 0x7c9a2b19  
 0x7c9a2b1a  
 0x7c9a2b1b  
 0x7c9a2b1c  
 0x7c9a2b1d

```
int n = 4;
int k[] = { 3, 14, 0, 10 };

int fib(int i) {
    if (i <= 2) return i;
    else return fib(i-1)+fib(i-2);
}
```

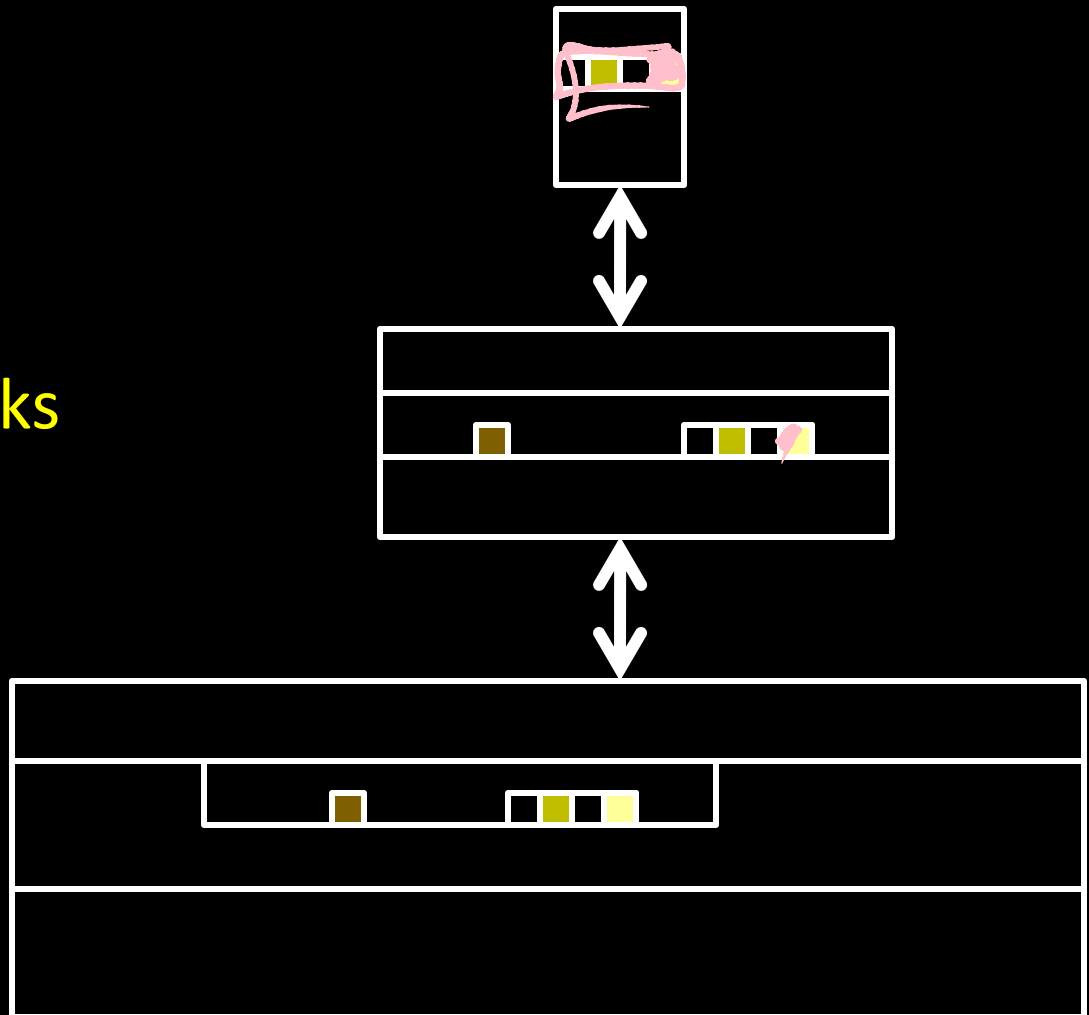
```
int main(int ac, char **av) {
    for (int i = 0; i < n; i++) {
        printf(fib(k[i]));
        printf("\n");
    }
}
```

0x00400318  
 0x0040031c

time

# Memory closer to processor is fast and small

- usually stores **subset** of memory farther from processor
  - “strictly inclusive”
- alternatives:
  - strictly exclusive
  - mostly inclusive
- Transfer whole **blocks** **cache lines**, e.g:
  - 4kb: disk  $\leftrightarrow$  ram
  - 256b: ram  $\leftrightarrow$  L2
  - 64b: L2  $\leftrightarrow$  L1

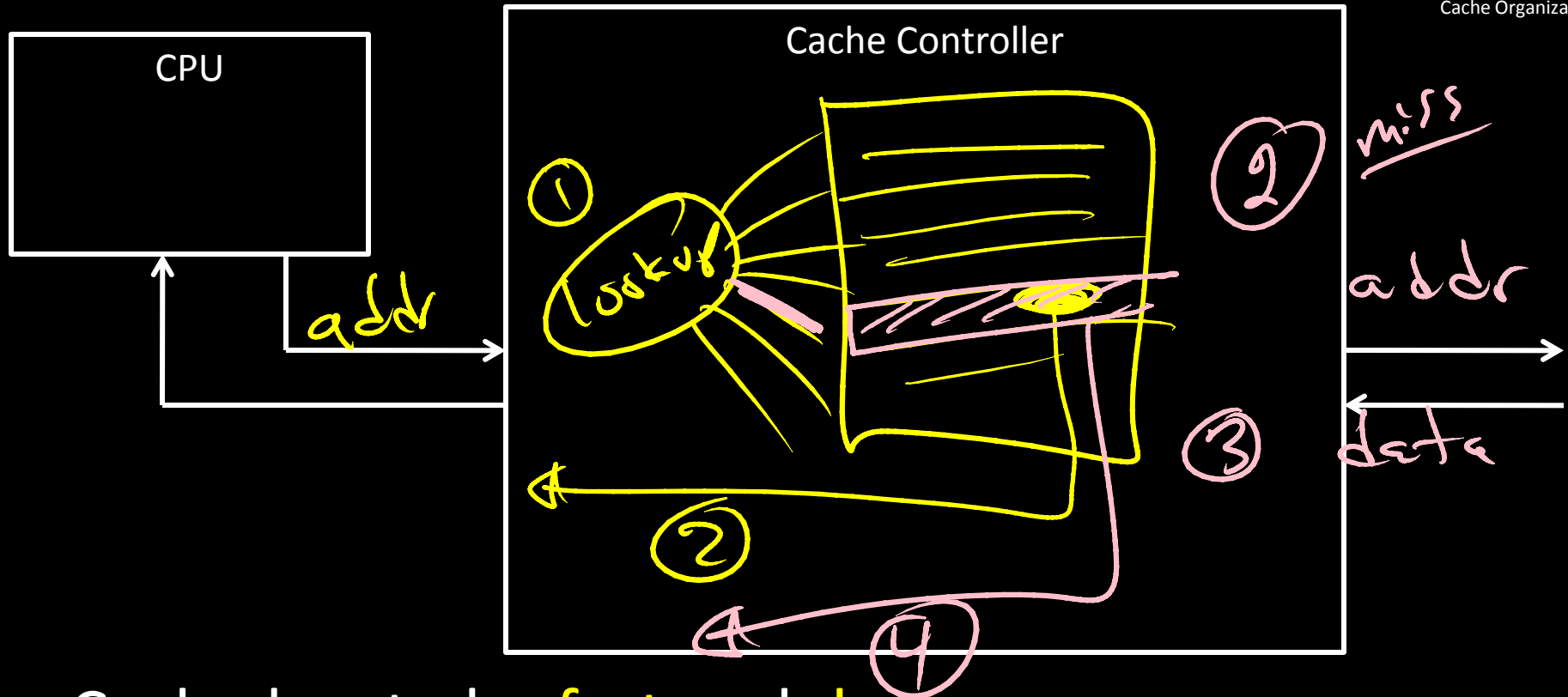




Processor tries to access Mem[x]

Check: is block containing x in the cache?

- Yes: **cache hit**
  - return requested data from cache line
- No: **cache miss**
  - read block from memory (or lower level cache)
  - (evict an existing cache line to make room)
  - place new block in cache
  - return requested data
  - **and stall the pipeline while all of this happens**

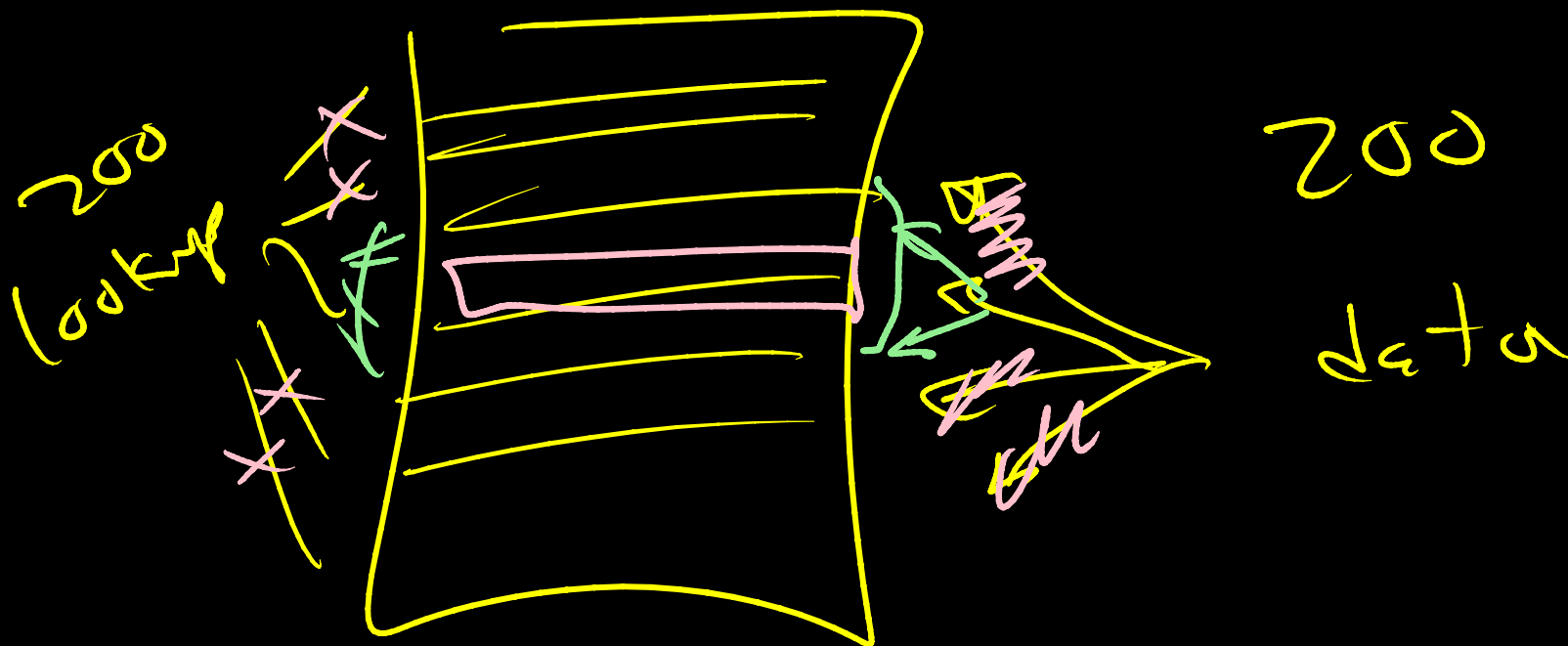


Cache has to be **fast** and **dense**

- Gain speed by performing lookups in parallel
  - but requires die real estate for lookup logic
- Reduce lookup logic by limiting where in the cache a block might be placed
  - but might reduce cache effectiveness

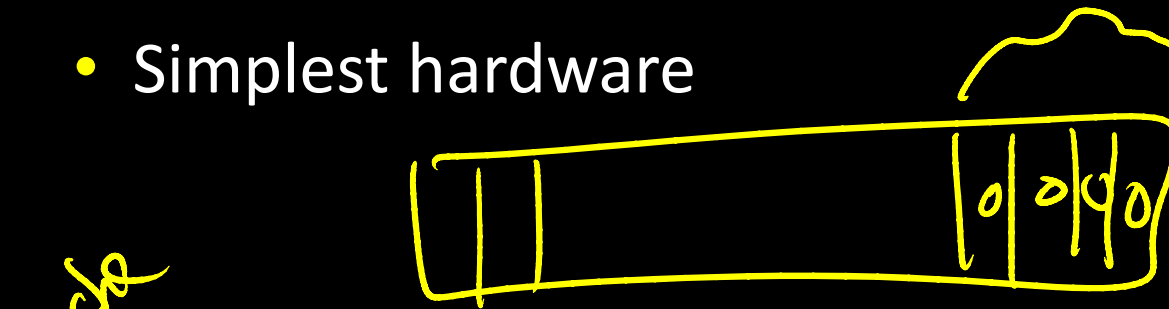
A given data block can be placed...

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



# Direct Mapped Cache

- Each block number mapped to a single cache line index
- Simplest hardware



cache

line 0	4	4
line 1	4	4
line 2	4	4
line 3	4	4

4 lines

2 words

0	(	0x000000	
		0x000004	
1	(	0x000008	
		0x00000c	
2	(	0x000010	
		0x000014	
3	(	0x000018	
		0x00001c	
0	(	0x000020	
		0x000024	
1	(	0x00002c	
		0x000030	
2	(	0x000034	
		0x000038	
3	(	0x00003c	
		0x000040	
4	(	0x000044	
		0x000048	
0	(	0x00004c	

Assume sixteen 64-byte cache lines

0x7FFF3D4D

= 0111 1111 1111 1111 0011 1101 0100 1101

TAG

64b

offset  
into  
cache  
with  
line

Need meta-data for each cache line:

- valid bit: is the cache line non-empty?
- tag: which block is stored in this line (if valid)

Q: how to check if X is in the cache?

Q: how to clear a cache line?

Tag	Index	Offset
-----	-------	--------

$n$  bit index,  $m$  bit offset

Q: How big is cache (data only)?

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

$$2^m \cdot 2^n = 2^{(n+m)} \text{ bytes in cache}$$

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

## Processor

lb \$1 ← M[ 1 ]  
 lb \$2 ← M[ 13 ]  
 lb \$3 ← M[ 0 ]  
 lb \$3 ← M[ 6 ]  
 lb \$2 ← M[ 5 ]  
 lb \$2 ← M[ 6 ]  
 lb \$2 ← M[ 10 ]  
 lb \$2 ← M[ 12 ]

\$1	103
\$2	
\$3	
\$4	

## Direct Mapped Cache

A = 0 0 0 0 1  
 tag index offset with line

V	tag	data	
1	00	101	103
0			
0			
0			

Hits:

Misses: 1

## Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181

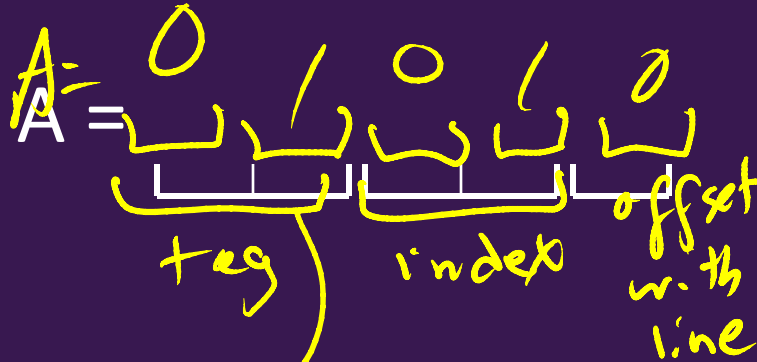
Using **byte addresses** in this example! Addr Bus = 5 bits

## Processor

~~lb \$1 ← M[1]~~ ✓  
~~lb \$2 ← M[13]~~ ✓  
~~lb \$3 ← M[0]~~ ✓  
~~lb \$3 ← M[6]~~ ✓  
~~lb \$2 ← M[5]~~ ✓  
~~lb \$2 ← M[6]~~ ✓  
 lb \$2 ← M[10]  
 lb \$2 ← M[12]

\$1	103
\$2	137
\$3	101
\$4	131

## Direct Mapped Cache



V tag data

1	00	101	103
0			
1	00	113	127
1	00	131	137

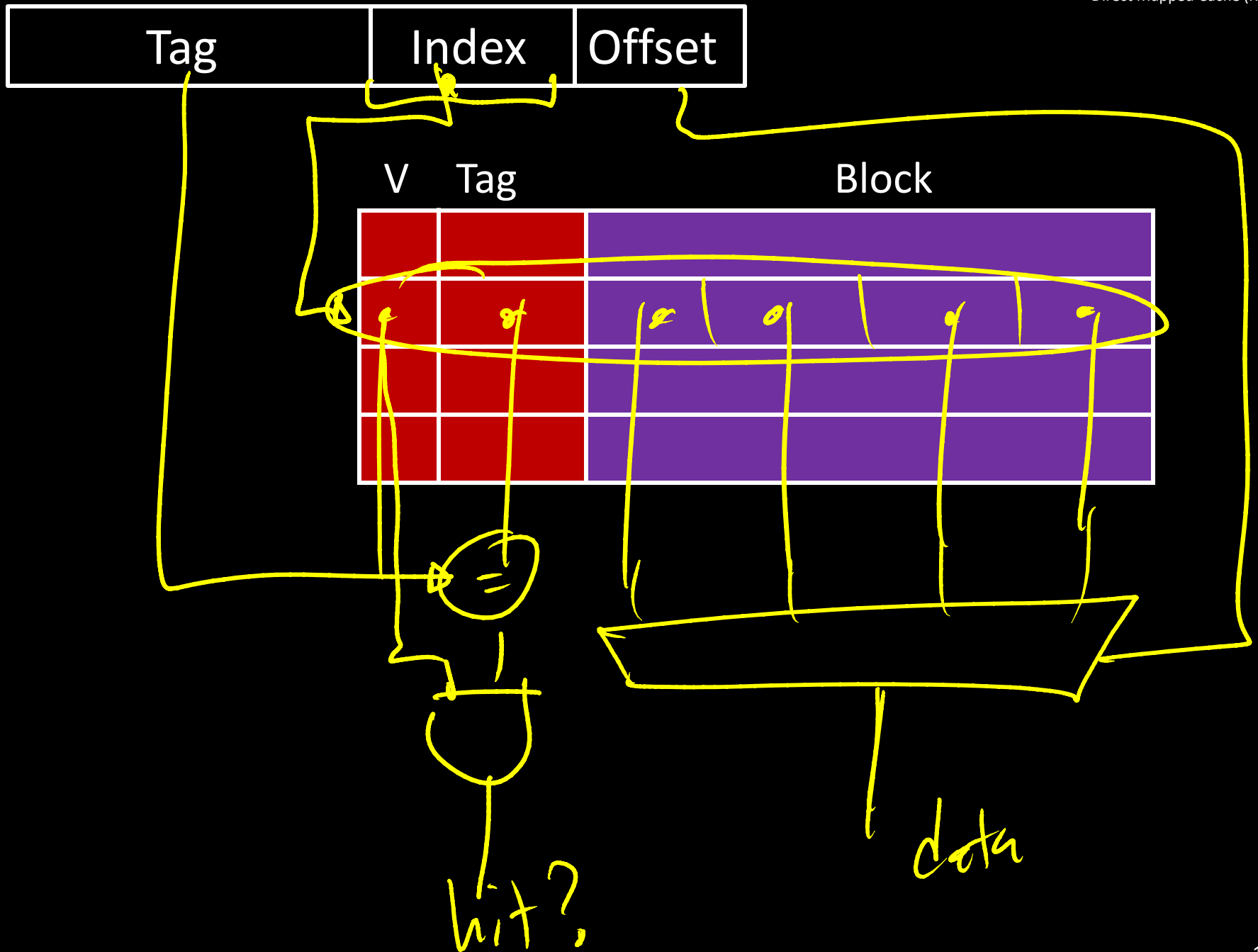
Hits: 11

Misses: 11111111

## Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181





Tag	Index	Offset
-----	-------	--------

$32 - n - m$      $\underbrace{\hspace{1cm}}_n$      $\underbrace{\hspace{1cm}}_m$

$n$  bit index,  $m$  bit offset

Q: How big is cache (data only)?

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

$2^n$  lines  $\cdot 2^m$  bytes per line =  $2^{n+m}$  bytes

$$(32 - n - m + 1) \cdot 2^n \text{ lines} =$$

# 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

5 cycles for cache Hit

5 + 50 + 3.15 = 100 cycles cache Miss

95% Hit Rate  
Performance depends on:

Access time for hit, miss, penalty, hit rate

$$.95 \times 5 + .05 \times 100 = 14.5$$

## 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

Q: How to avoid...

Cold Misses → fix program, prefetch

- Unavoidable? The data was never in the cache...
- Prefetching!

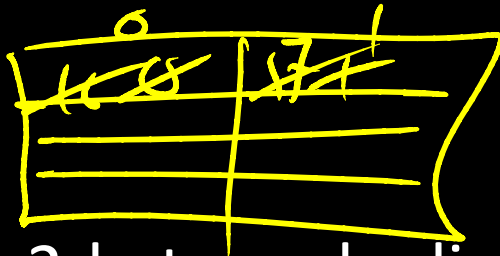
Other Misses

- Buy more SRAM
  - Use a more flexible cache design
- flexible cache layout  
More SRAM

Bigger cache doesn't always help...

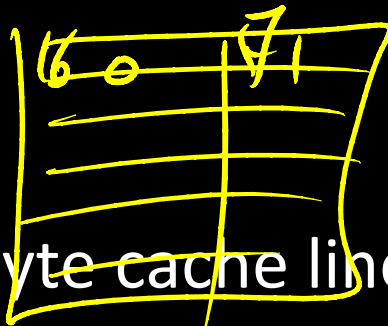
Memcpy access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, ...

Hit rate with four direct-mapped 2-byte cache lines?



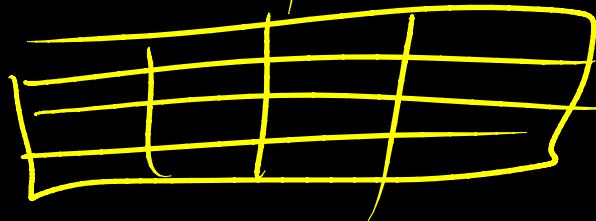
0%

With eight 2-byte cache lines?



0%

With four 4-byte cache lines?



0%

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	

# 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

Q: How to avoid...

## Cold Misses

- Unavoidable? The data was never in the cache...
- Prefetching!

## Capacity Misses

- Buy more SRAM

## Conflict Misses

- Use a more flexible cache design



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

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

## Processor

lb \$1 ← M[ 1 ] *M*  
 lb \$2 ← M[ 13 ] *M*  
 lb \$3 ← M[ 0 ] *H*  
 lb \$3 ← M[ 6 ] *M*  
 lb \$2 ← M[ 5 ] *M*  
 lb \$2 ← M[ 6 ] *H*  
 lb \$2 ← M[ 10 ] *M*  
 lb \$2 ← M[ 12 ] *H*

\$1  
 \$2  
 \$3  
 \$4


## Fully Associative Cache

A = 

0	1	0	1	0
---	---	---	---	---

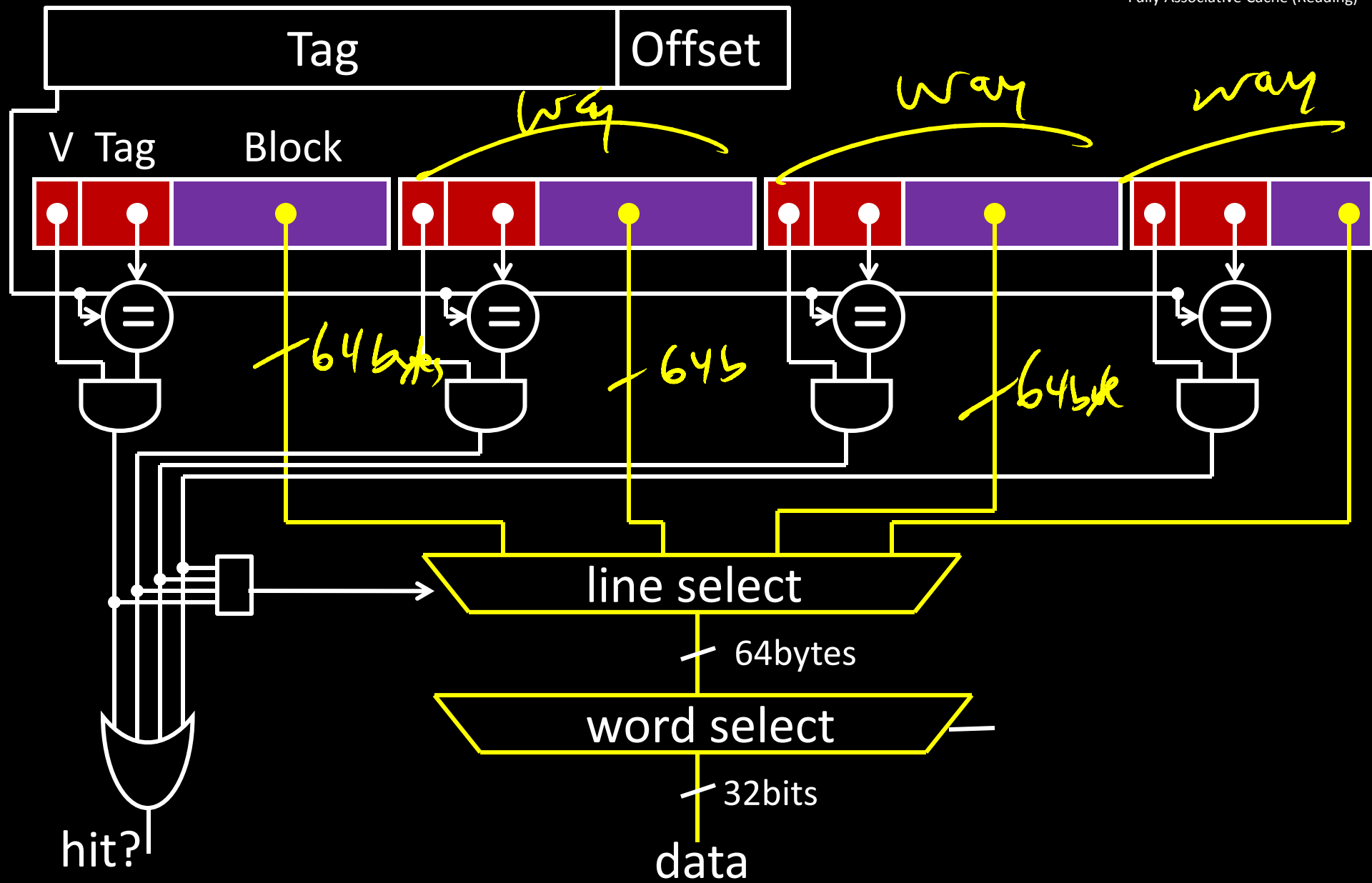
  
*tag*
*offset*

V	tag	data	
1	0101		
1	0110	163	167
1	0011	131	137
1	0010		

Hits: *1* Misses: *11*

## Memory

0	101
1	103
2	107
3	109
4	113
5	127
6	131
7	137
8	139
9	149
10	151
11	157
12	163
13	167
14	173
15	179
16	181



Tag	Offset
-----	--------

$m$  bit offset,  $2^n$  cache lines

Q: How big is cache (data only)?

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

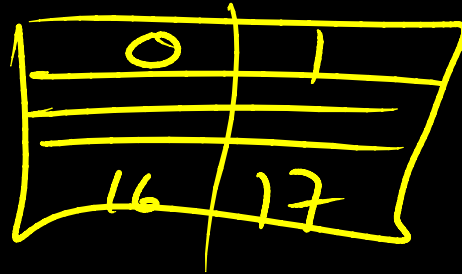
Fully-associative reduces conflict misses...

... assuming good eviction strategy

M M H H

Memcpy access trace: 0, 16, 1, 17, 2, 18, 3, 19, 4, 20, ...

Hit rate with four fully-associative 2-byte cache lines?



50%

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	

... but large block size can still reduce hit rate

vector add access trace: 0, 100, 200, 1, 101, 201, 2, 202, ...

Hit rate with four fully-associative 2-byte cache lines?

With two 4-byte cache lines?

# 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 *O for F.A. cache (ideal)*

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

## Caching assumptions

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

## Benefits

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

## Tradeoffs:

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

- Fully Associative → higher hit cost, higher hit rate
- Larger block size → lower hit cost, higher miss penalty

Next up: other designs; writing to caches