

# Caches and Memory

Anne Bracy

CS 3410

Computer Science

Cornell University

Slides by Anne Bracy with 3410 slides by Professors Weatherspoon, Bala, McKee, and Sirer.

See P&H B.9, Chapter: 5.1-5.4, 5.8, 5.10, 5.13, 5.15, 5.17

# Programs 101

## C Code

```
int main (int argc, char* argv[ ]) {
    int i;
    int m = n;
    int sum = 0;
    for (i = 1; i <= m; i++) {
        sum += i;
    }
    printf (“...”, n, sum);
}
```

## Load/Store Architectures:

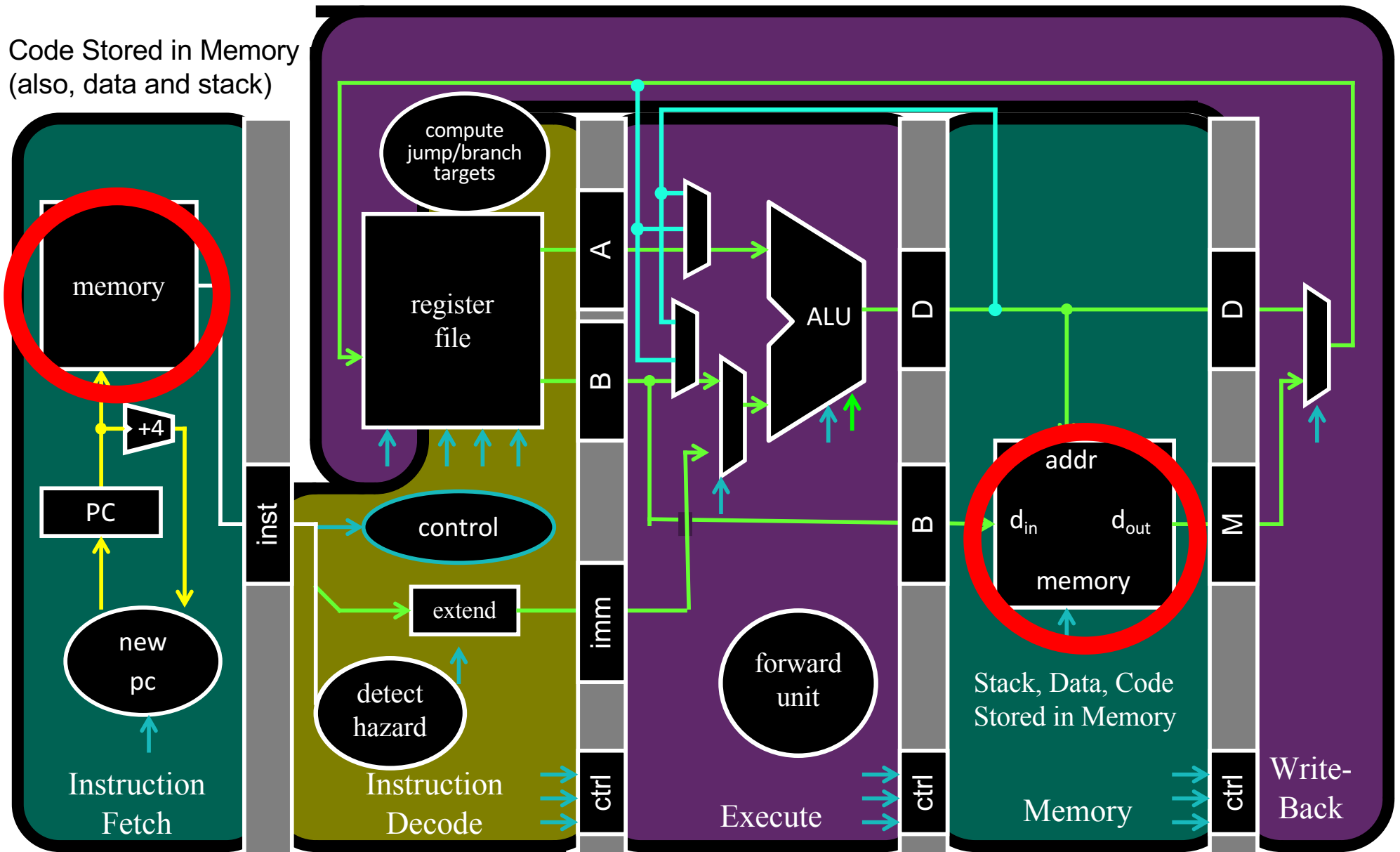
- Read data from memory (put in registers)
- Manipulate it
- Store it back to memory

## MIPS Assembly

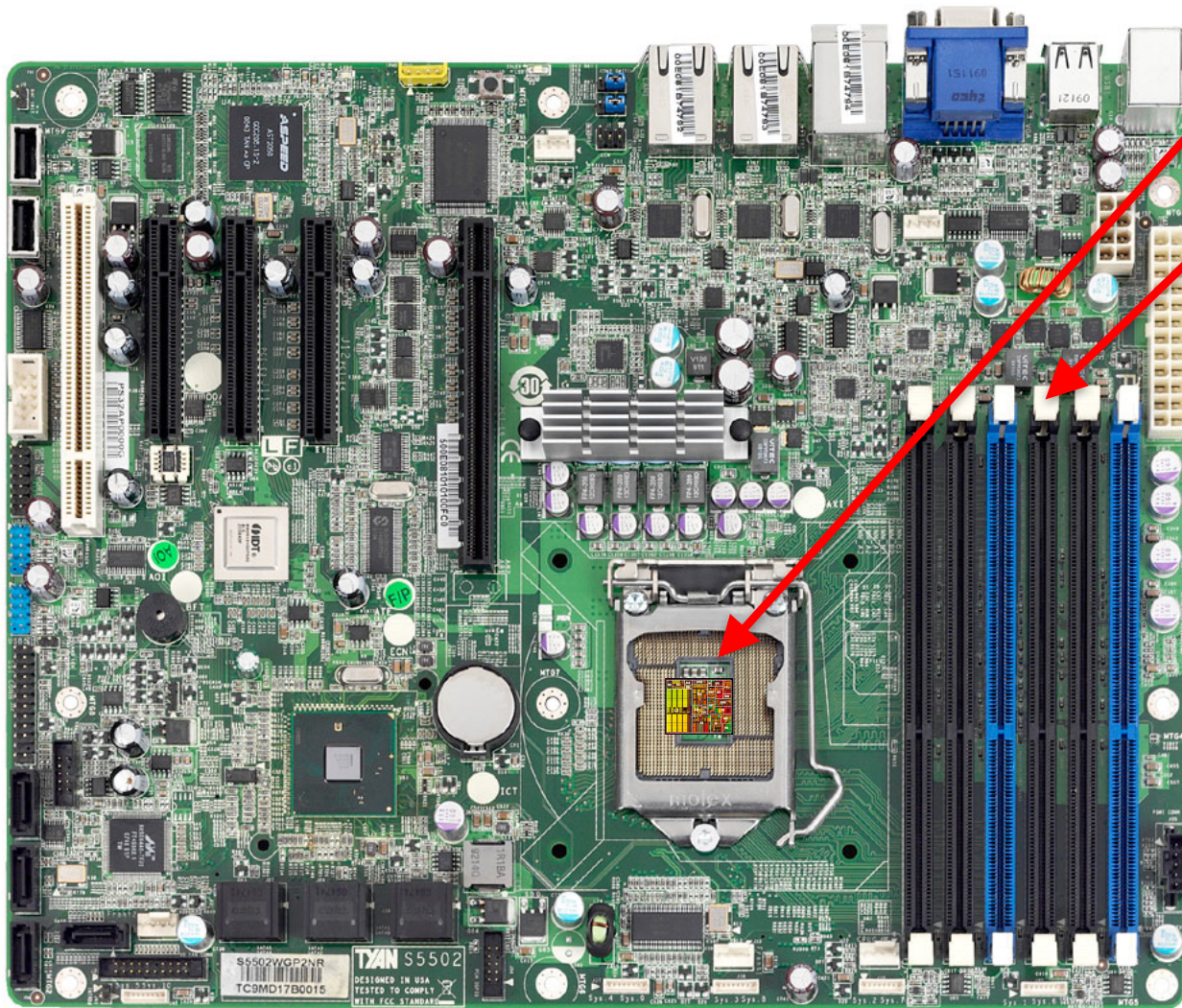
```
main:    addiu    $sp,$sp,-48
        sw     $31,44($sp)
        sw     $fp,40($sp)
        move   $fp,$sp
        sw     $4,48($fp)
        sw     $5,52($fp)
        la    $2,n
        lw     $2,0($2)
        sw     $2,28($fp)
        sw     $0,32($fp)
        li    $2,1
        sw     $2,24($fp)
$L2:    lw     $2,24($fp)
        lw     $3,28($fp)
        slt   $2,$3,$2
        bne   $2,$0,$L3
        . . .
```

■ Instructions that read from or write to memory...

# 1 Cycle Per Stage: the Biggest Lie (So Far)



# What's the problem?



CPU

Main Memory

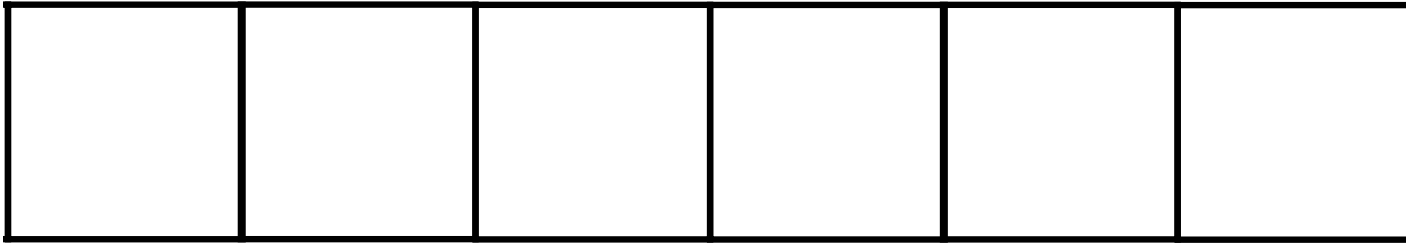
+ big

– slow

– far away

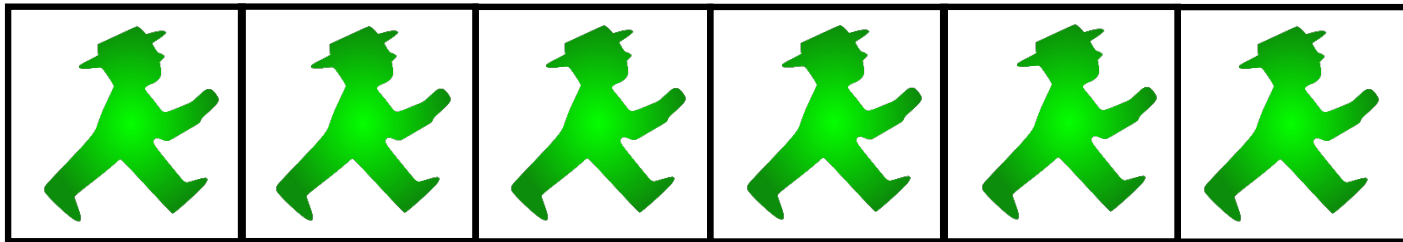
# The Need for Speed

CPU Pipeline



# The Need for Speed

## CPU Pipeline

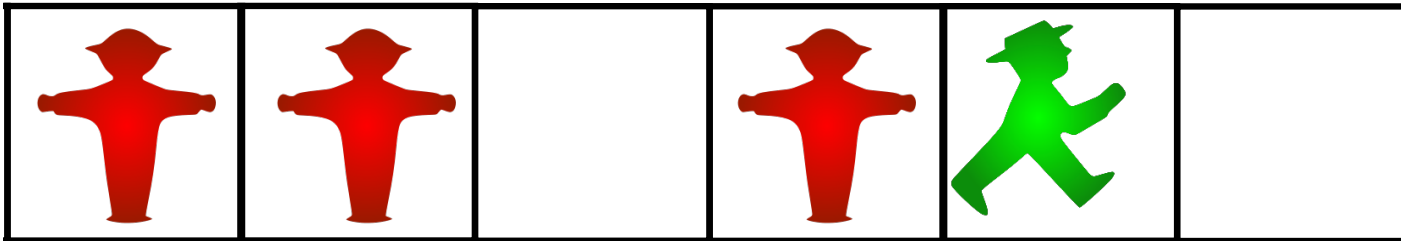


Instruction speeds:

- `add, sub, shift`: 1 cycle
- `mult`: 3 cycles
- `load/store`: **100 cycles**  
off-chip 50(-70)ns  
2(-3) GHz processor → 0.5 ns clock

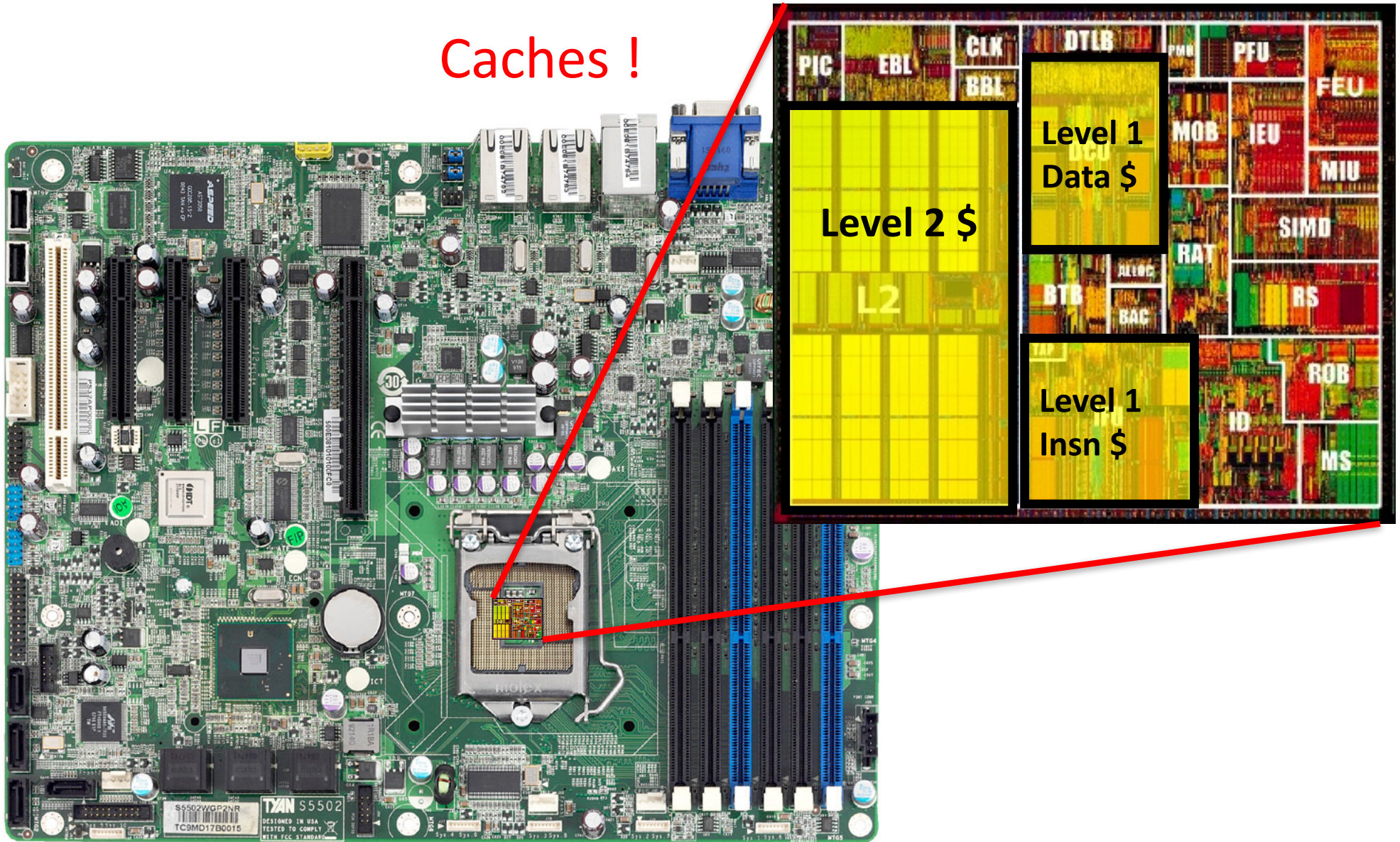
# The Need for Speed

CPU Pipeline



# What's the solution?

Caches !



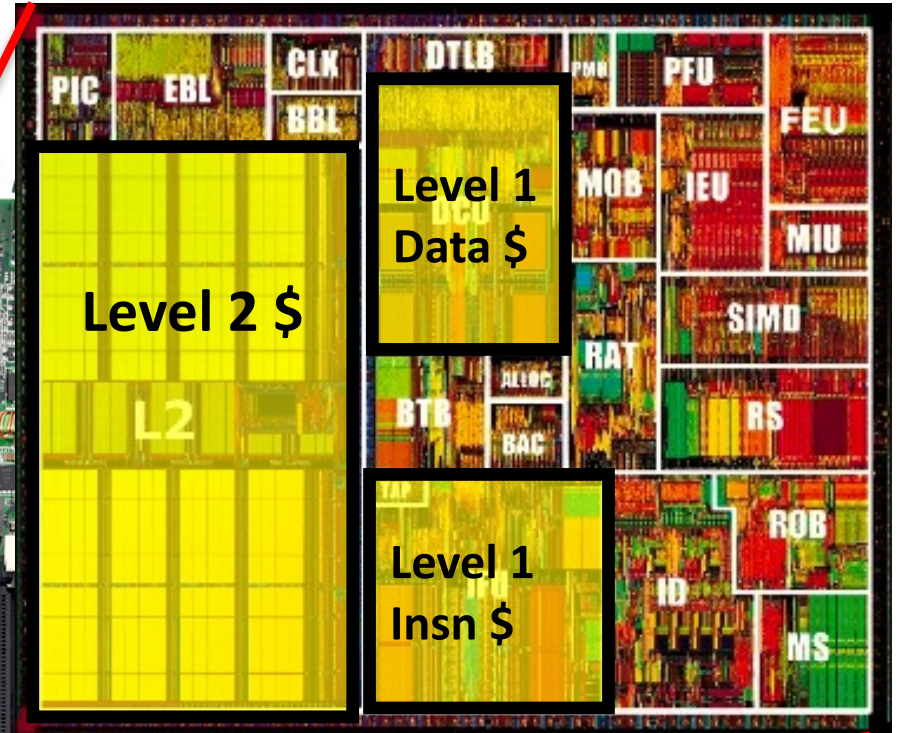
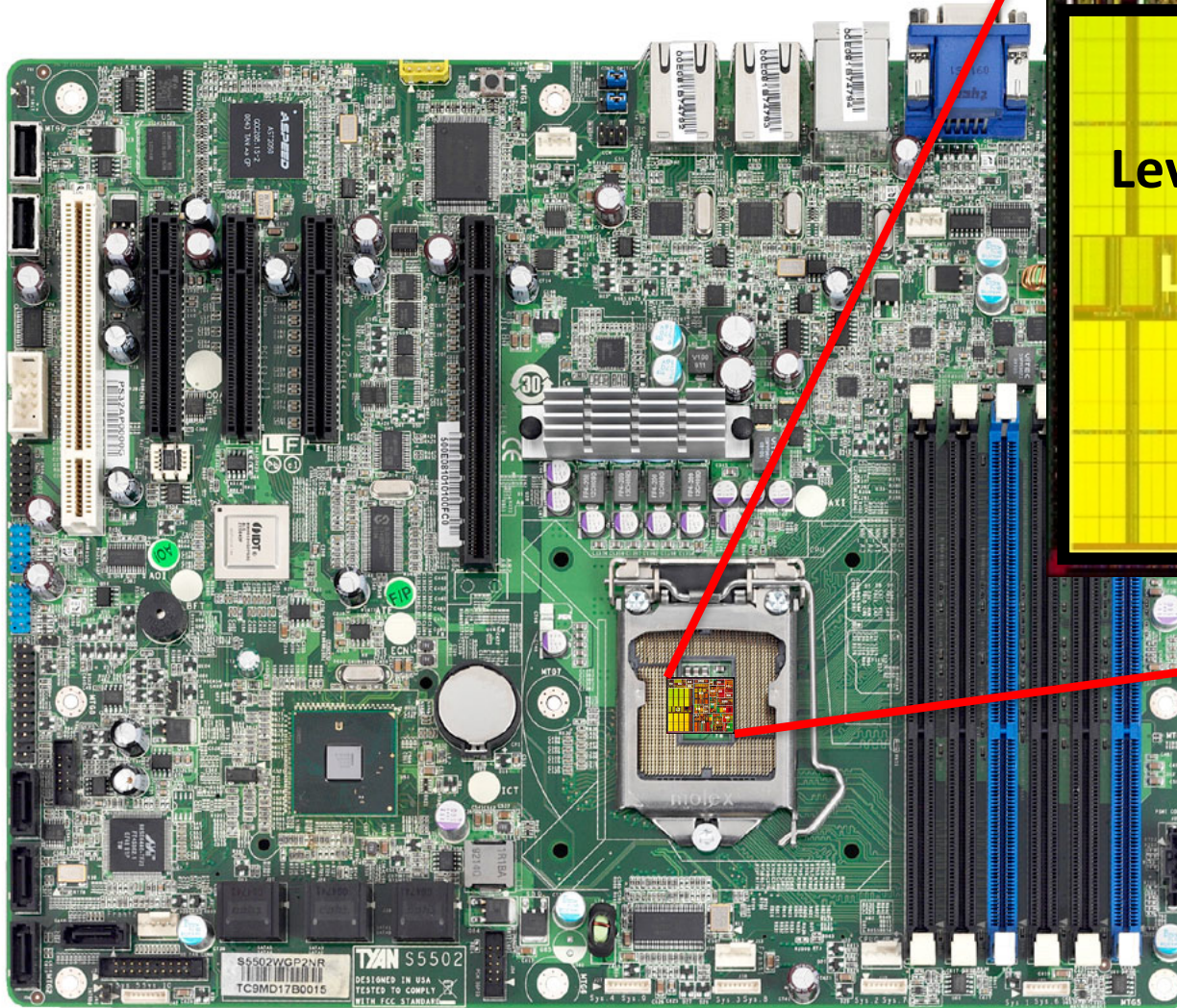


# Aside

- Go back to 04-state and look at how SRAM and DRAM are built.

# What's the solution?

Caches !



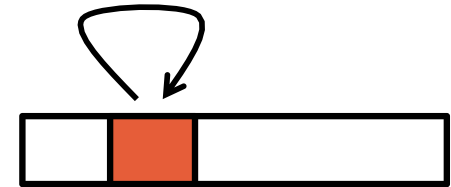
*What lucky data gets to go here?*

# Locality Locality Locality

If you ask for something, you're likely to ask for:

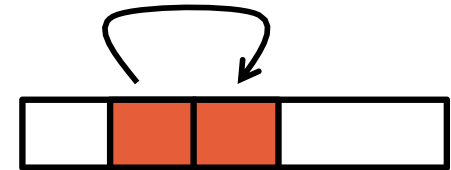
- the same thing again soon

→ Temporal Locality



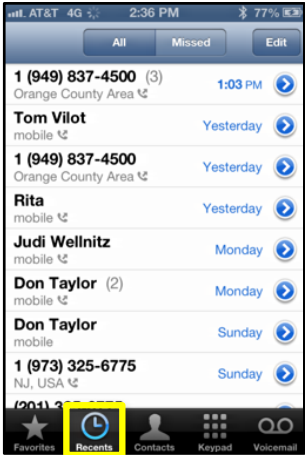
- something near that thing, soon

→ Spatial Locality



```
total = 0;
for (i = 0; i < n; i++)
    total += a[i];
return total;
```

# Your life is full of Locality



Last Called

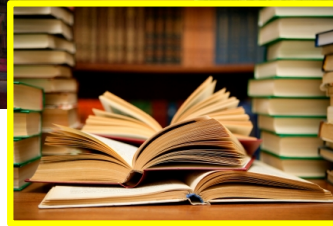
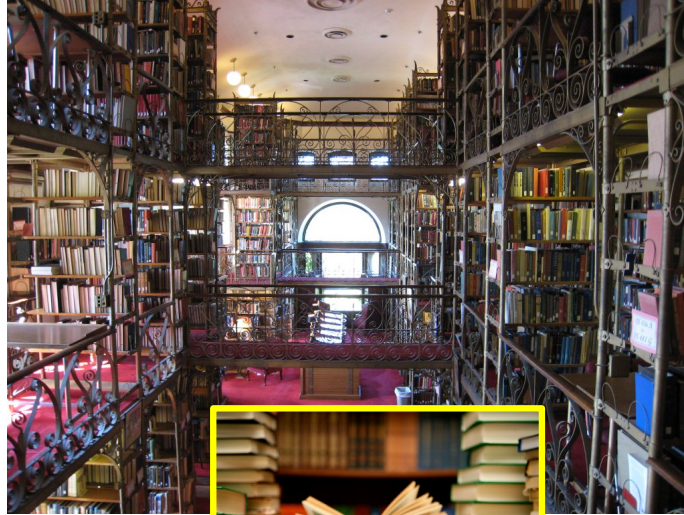
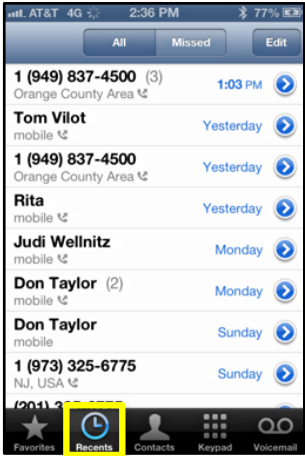
Speed Dial

Favorites

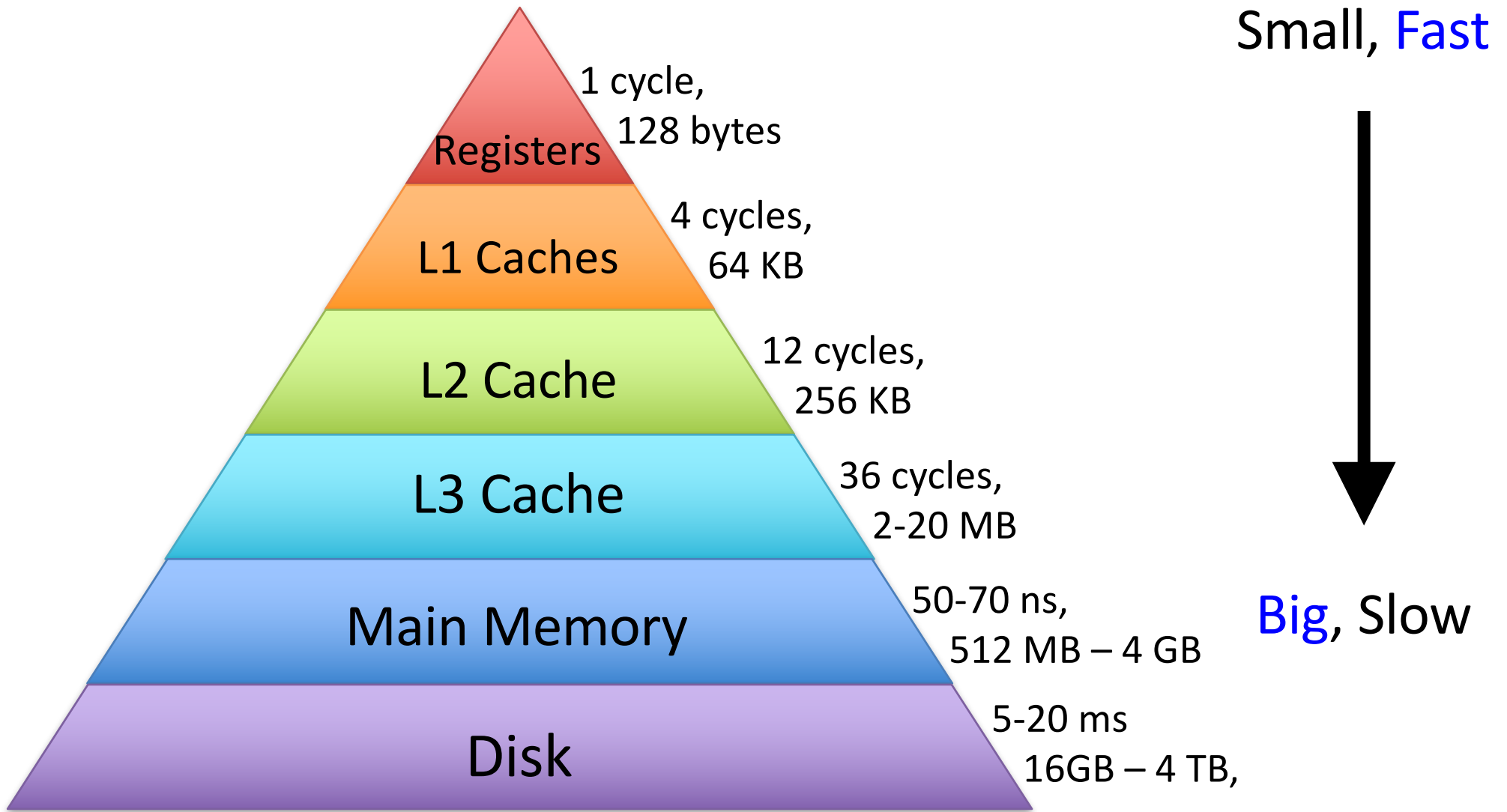
Contacts

Google/Facebook/email

# Your life is full of Locality



# The Memory Hierarchy



# Some Terminology

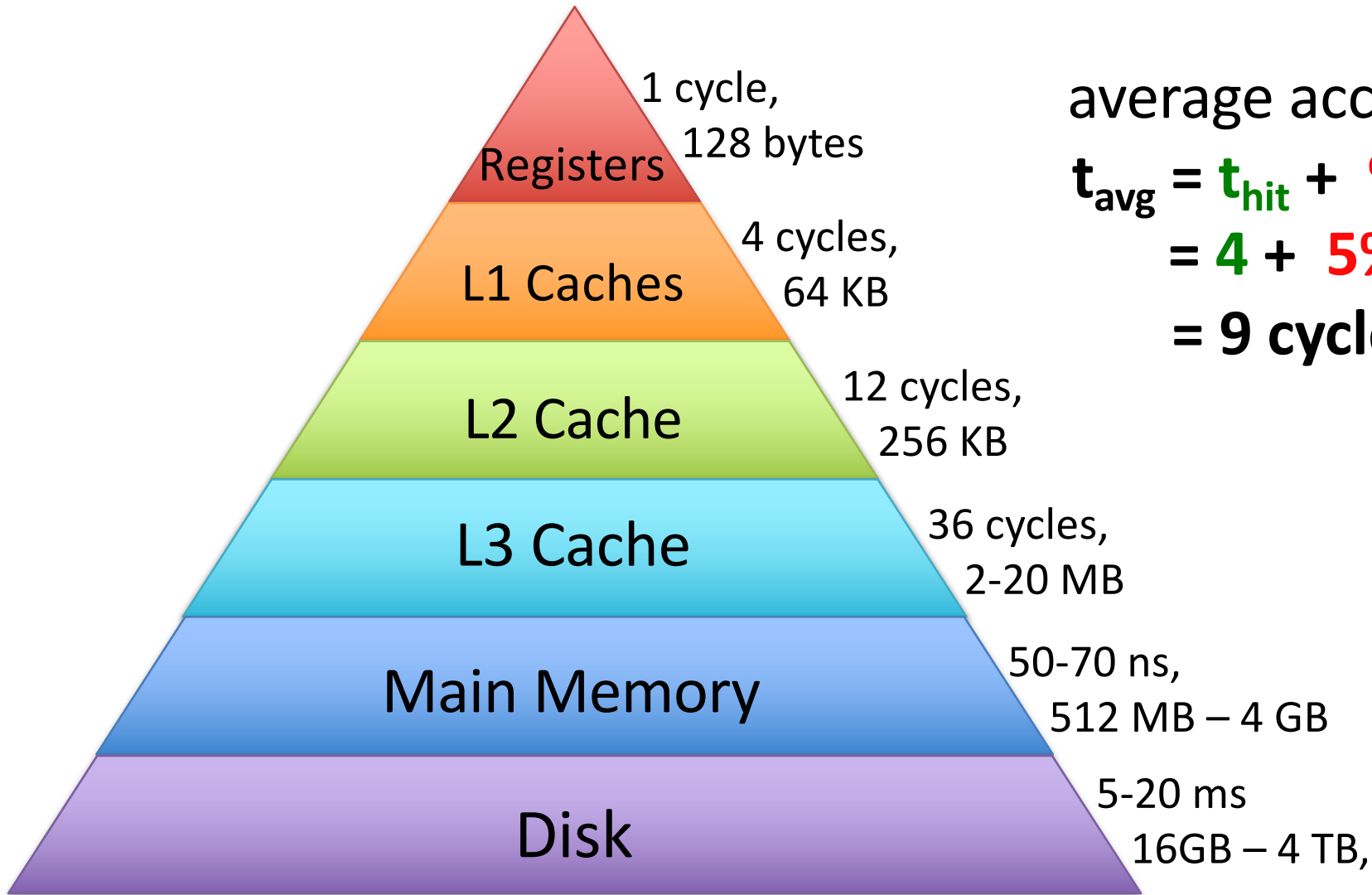
## Cache hit

- data is in the Cache
- $t_{\text{hit}}$ : time it takes to access the cache
- %hit: Hit rate. # cache hits / # cache accesses

## Cache miss

- data is **not** in the Cache
- $t_{\text{miss}}$ : time it takes to get the data from below the \$
- Miss rate (%miss): # cache misses / # cache accesses

# The Memory Hierarchy

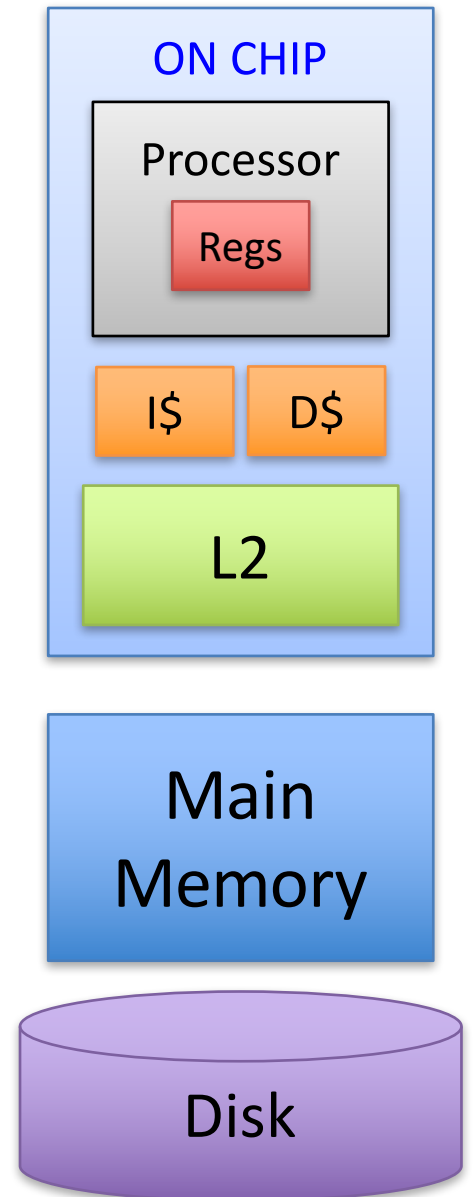
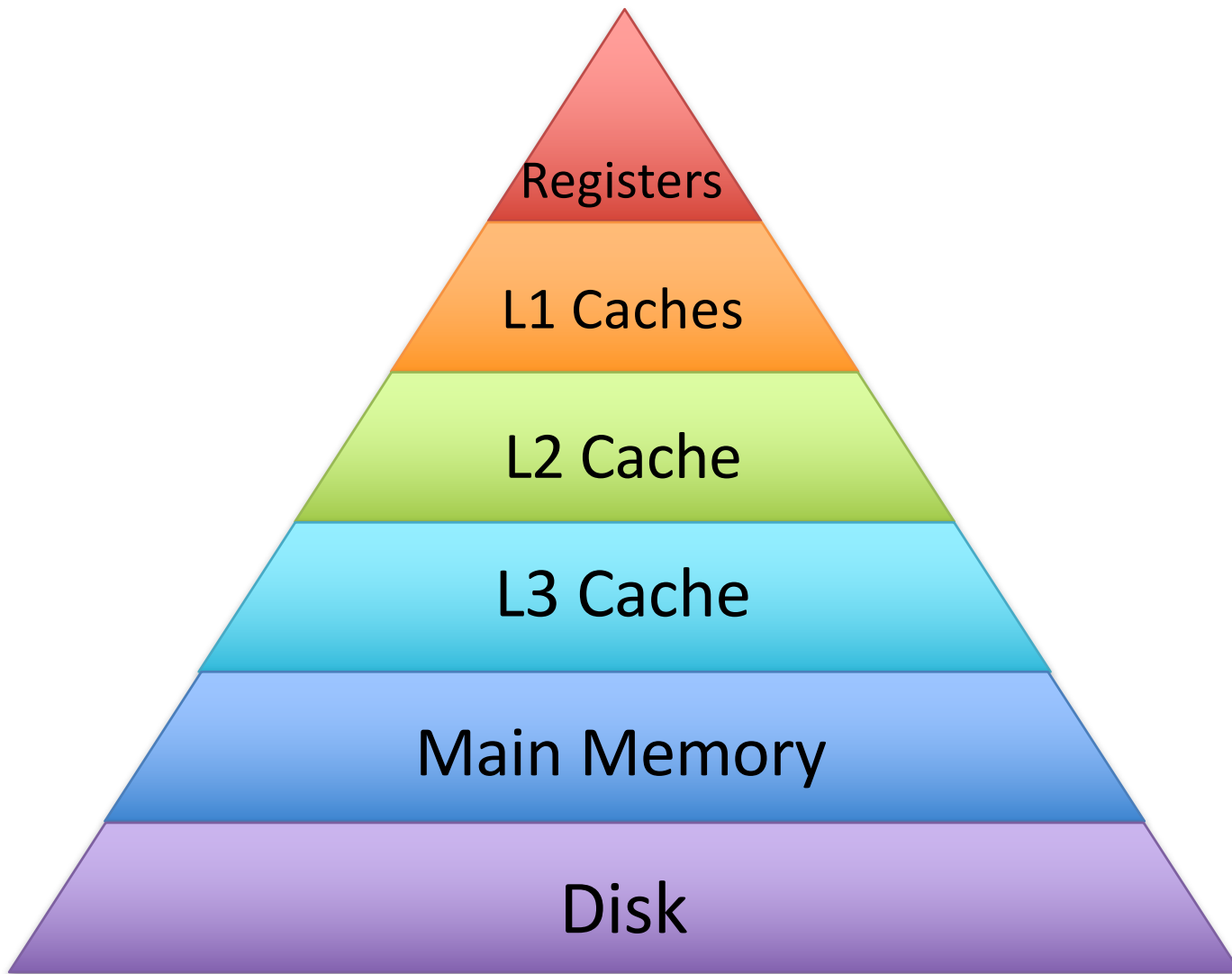


average access time

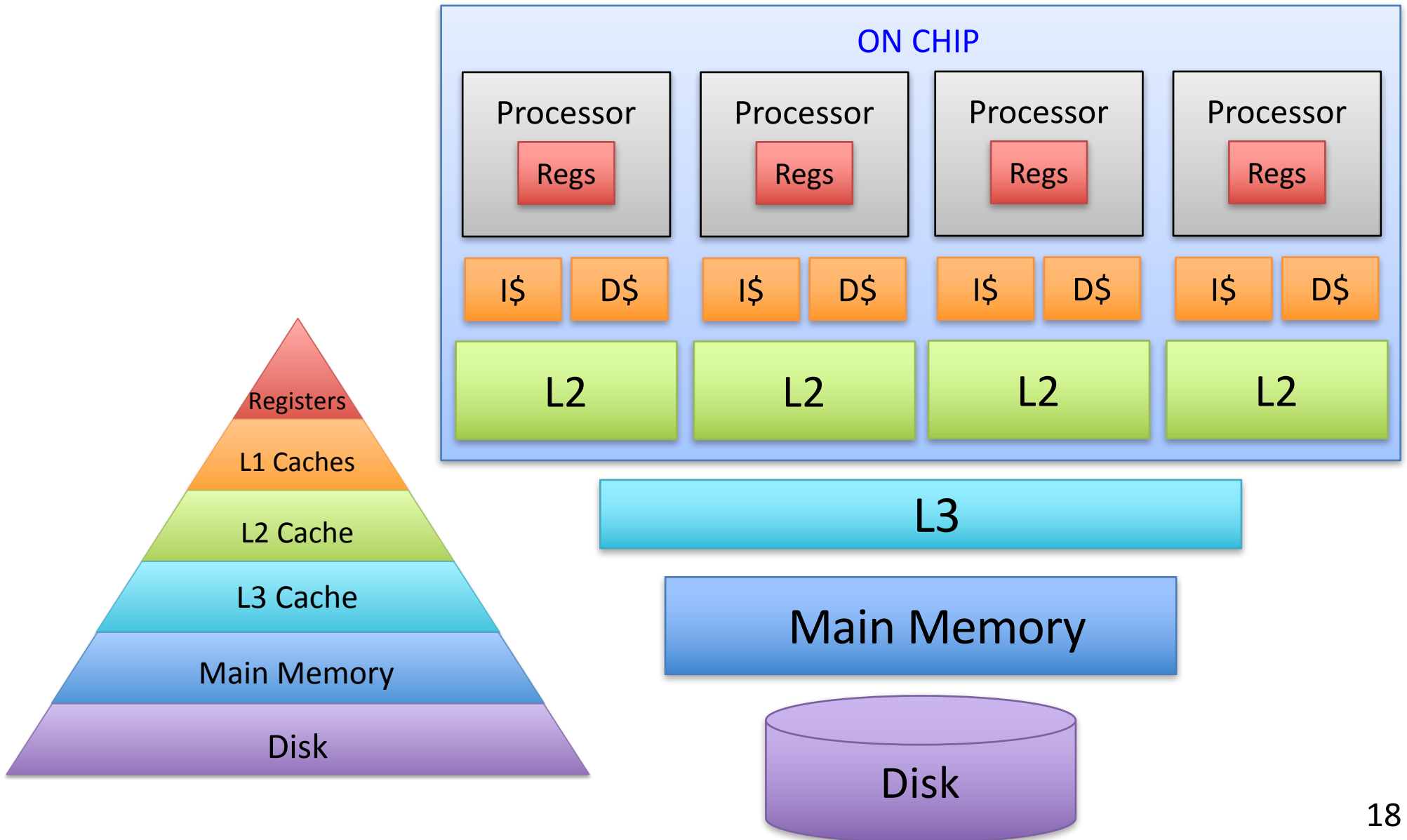
$$\begin{aligned} t_{\text{avg}} &= t_{\text{hit}} + \%_{\text{miss}} * t_{\text{miss}} \\ &= 4 + 5\% \times 100 \\ &= 9 \text{ cycles} \end{aligned}$$



# Single Core Memory Hierarchy



# Multi-Core Memory Hierarchy



# Memory Hierarchy by the Numbers

CPU clock rates  $\sim 0.33\text{ns} - 2\text{ns}$  (3GHz-500MHz)

Memory technology	Transistor count*	Access time	Access time in cycles	\$ per GIB in 2012	Capacity
SRAM (on chip)	6-8 transistors	0.5-2.5 ns	1-3 cycles	\$4k	256 KB
SRAM (off chip)		1.5-30 ns	5-15 cycles	\$4k	32 MB
DRAM	1 transistor (needs refresh)	50-70 ns	150-200 cycles	\$10-\$20	8 GB
SSD (Flash)		5k-50k ns	Tens of thousands	\$0.75-\$1	512 GB
Disk		5M-20M ns	Millions	\$0.05-\$0.1	4 TB

\*Registers,D-Flip Flops: 10-100's of registers

# Basic Cache Design

## Direct Mapped Caches



# 16 Byte Memory

load 1100 → r1

- Byte-addressable memory
- 4 address bits → 16 bytes total
- b addr bits →  $2^b$  bytes in memory

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

21

# 4-Byte, Direct Mapped Cache

## CACHE

**index**  
**XXXX**

index	data
00	A
01	B
10	C
11	D

← Cache entry  
= row  
= (cache) line  
= (cache) block  
**Block Size: 1 byte**

## Direct mapped:

- Each address maps to 1 cache block
- 4 entries  $\rightarrow$  2 index bits ( $2^n \rightarrow n$  bits)

## Index with LSB:

- Supports spatial locality

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Analogy to a Spice Rack



**Spice Rack  
(Cache)**

index spice



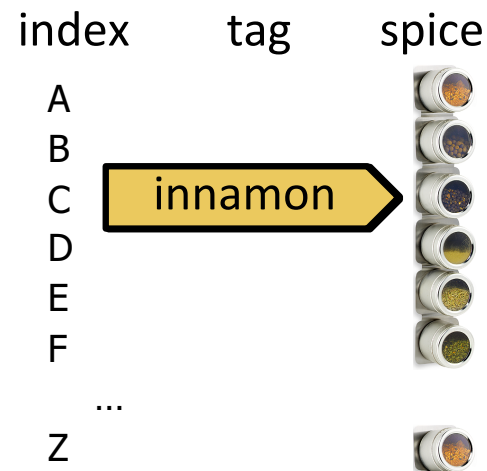
**Spice Wall  
(Memory)**



- Compared to your spice wall
  - Smaller
  - Faster
  - More costly (per oz.)

# Analogy to a Spice Rack

## Spice Rack (Cache)



## Spice Wall (Memory)



- How do you know what's in the jar?
- Need labels

**Tag** = Ultra-minimalist label



# 4-Byte, Direct Mapped Cache

tag | index  
XXXX

## CACHE

index	tag	data
00	00	A
01	00	B
10	00	C
11	00	D

**Tag:** minimalist label/address

address = tag + index

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

25

# 4-Byte, Direct Mapped Cache

## CACHE

index	V	tag	data
00	0	00	X
01	0	00	X
10	0	00	X
11	0	00	X

One last tweak: **valid bit**

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

26

# Simulation #1 of a 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	0	11	X
01	0	11	X
10	0	11	X
11	0	11	X

load

1100

Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #1 of a 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

load

1100

Miss

Lookup:

- Index into \$
- Check tag
- Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #1 of a 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	0	11	X
10	0	11	X
11	0	11	X

load 1100  
...  
load 1100

Miss

Hit!

Lookup:

➔ Index into \$

➔ Check tag

➔ Check valid bit

*Awesome!*

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Clicker Questions

Processor A is a 1 GHz processor (1 cycle = 1 ns). There is an L1 cache with a 1 cycle access time and a 50% hit rate. There is an L2 cache with a 10 cycle access time and a 90% hit rate. It takes 100 ns to access memory.

**Q1:** What is the average memory access time of Processor A in ns?    A: 5    B: 7    C: 9    D: 11    E: 13

Proc. B attempts to improve upon Proc. A by making the L1 cache larger. The new hit rate is 90%. To maintain a 1 cycle access time, Proc. B runs at 500 MHz (1 cycle = 2 ns).

**Q2:** What is the average memory access time of Processor B in ns?    A: 5    B: 7    C: 9    D: 11    E: 13

**Q3:** Which processor should you buy?

A: Processor A    B: Processor B    C: They are equally good. 30

# Clicker Answers

- Q1:

$$1 + 50\% (10 + 10\% \times 100) =$$

$$1 + 5 + 5 = 11 \text{ ns}$$

- Q2:

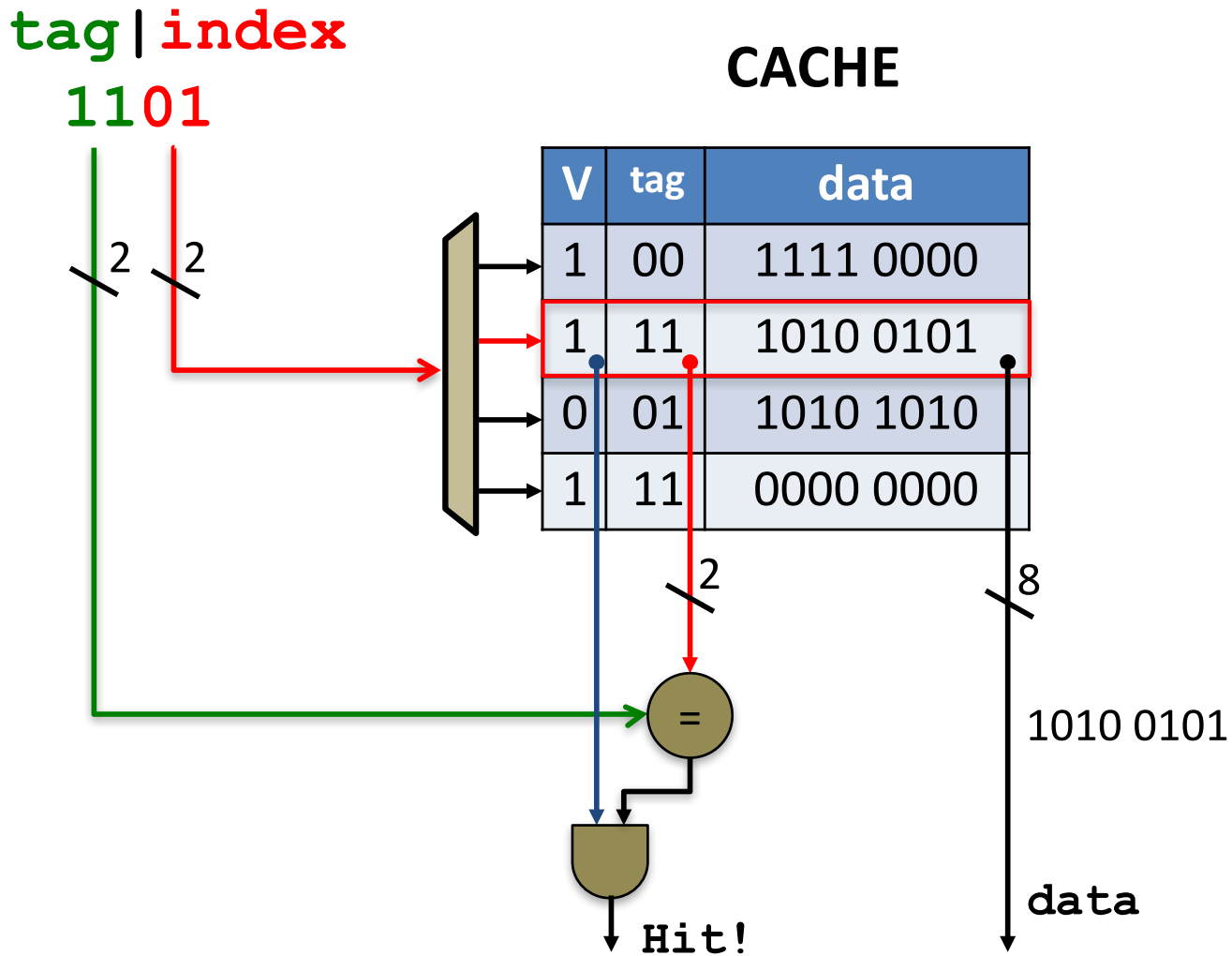
$$2 + 10\% (20 + 10\% \times 100) =$$

$$2 + 2 + 1 = 5 \text{ ns}$$

**Processor A.** Although memory access times are over 2x faster on Processor B, the frequency of Processor B is 2x slower. Since most workloads are not more than 50% memory instructions, Processor A is still likely to be the faster processor for most work-loads. (If I did have workloads that were over 50% memory instructions, Processor B might be the better choice.)

# Block Diagram

## 4-entry, direct mapped Cache



*Great!*  
*Are we done?*



# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	0	11	X
01	0	11	X
10	0	11	X
11	0	11	X

load 1100  
load 1101  
load 0100  
load 1100

Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

load 1100  
load 1101  
load 0100  
load 1100

Miss

Lookup:

- Index into \$
- Check tag
- Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	0	11	X
10	0	11	X
11	0	11	X

load 1100  
load 1101  
load 0100  
load 1100

Miss  
Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	11	X
11	0	11	X

load 1100  
load 1101  
load 0100  
load 1100

Miss

Miss

Lookup:

- Index into \$
- Check tag
- Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	xx	X
11	0	xx	X

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

load 1100 Miss  
 load 1101 Miss  
 load 0100 Miss  
 load 1100

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	11	X
11	0	11	X

load 1100 Miss  
 load 1101 Miss  
 load 0100 Miss  
 load 1100

Lookup:

- Index into \$
- Check tag
- Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	11	X
11	0	11	X

load 1100 Miss  
 load 1101 Miss  
 load 0100 Miss  
 load 1100 Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #2: 4-byte, DM Cache

tag | index  
XXXX

## CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	11	X
11	0	11	X

load 1100 Miss cold  
 load 1101 Miss cold  
 load 0100 Miss cold  
 load 1100 Miss

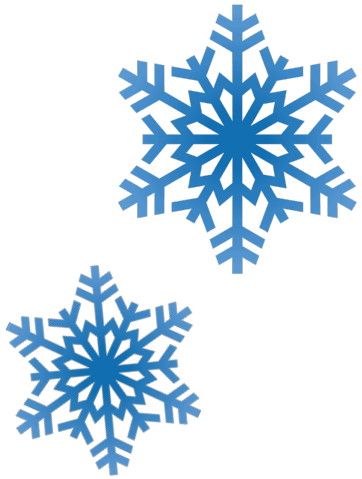
*Disappointed!*



## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q





# Reducing Cold Misses by Increasing Block Size

Leveraging Spatial Locality



# Increasing Block Size


**CACHE**

offset XXXX				
	index	V	tag	data
	00	0	x	A   B
	01	0	x	C   D
	10	0	x	E   F
	11	0	x	G   H

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

42

- **Block Size:** 2 bytes
- **Block Offset:** least significant bits indicate where you live in the block 
- Which bits are the index? tag?

# Simulation #3: 8-byte, DM Cache

## CACHE

tag | index | offset  
 XXXX

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	0	x	X   X
11	0	x	X   X

load 1100  
 load 1101  
 load 0100  
 load 1100

Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #3: 8-byte, DM Cache

tag | index | offset  
 XXXX

## CACHE

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	1	N   O
11	0	x	X   X

load 1100  
 load 1101  
 load 0100  
 load 1100

Miss

Lookup:

- Index into \$
- Check tag
- Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #3: 8-byte, DM Cache

## CACHE

tag | index | offset  
 XXXX

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	1	N   O
11	0	x	X   X

load 1100  
 load 1101  
 load 0100  
 load 1100

Miss  
 Hit!

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #3: 8-byte, DM Cache

## CACHE

tag | index | offset  
 XXXX

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	1	N   O
11	0	x	X   X

load 1100 Miss  
 load 1101 Hit!  
 load 0100 Miss  
 load 1100

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #3: 8-byte, DM Cache

## CACHE

tag | index | offset  
 XXXX

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	0	E   F
11	0	x	X   X

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

load 1100 Miss  
 load 1101 Hit!  
 load 0100 Miss  
 load 1100

Lookup:

- Index into \$
- Check tag
- Check valid bit

# Simulation #3: 8-byte, DM Cache

## CACHE

tag | index | offset  
 XXXX

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	0	E   F
11	0	x	X   X

load 1100 Miss  
 load 1101 Hit!  
 load 0100 Miss  
 load 1100 Miss

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q



# Simulation #3: 8-byte, DM Cache

## CACHE

index	V	tag	data
00	0	x	X   X
01	0	x	X   X
10	1	0	E   F
11	0	x	X   X

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

load 1100 Miss cold  
 load 1101 Hit!  
 load 0100 Miss cold  
 load 1100 Miss conflict

*1 hit, 3 misses*

*3 bytes don't fit in  
a 4 entry cache?*

# Removing Conflict Misses with Fully-Associative Caches



# 8 byte, fully-associative Cache

XXXX

## CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X   X	0	xxx	X   X	0	xxx	X   X	0	xxx	X   X

What should the **offset** be?

What should the **index** be?

What should the **tag** be?

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #4: 8-byte, FA Cache

XXXX  
tag | offset

## CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X   X	0	xxx	X   X	0	xxx	X   X	0	xxx	X   X



load 1100  
load 1101  
load 0100  
load 1100

Miss

Lookup:

• Index into \$

➔ Check tags

➔ Check valid bits

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

➔ LRU Pointer

# Simulation #4: 8-byte, FA Cache

XXXX  
tag | offset

## CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N   O	0	xxx	X   X	0	xxx	X   X	0	xxx	X   X



load 1100 Miss  
 load 1101 Hit!  
 load 0100  
 load 1100

Lookup:  
 • Index into \$  
 → Check tags  
 → Check valid bits

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

# Simulation #4: 8-byte, FA Cache

XXXX  
tag | offset

## CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N   O	0	xxx	X   X	0	xxx	X   X	0	xxx	X   X



load 1100 Miss  
load 1101 Hit!  
load 0100 Miss  
load 1100

Lookup:

• Index into \$

➔ Check tags

➔ Check valid bits

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

➔ LRU Pointer

# Simulation #4: 8-byte, FA Cache

XXXX  
tag | offset

## CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N   O	1	010	E   F	0	xxx	X   X	0	xxx	X   X



load 1100 Miss  
load 1101 Hit!  
load 0100 Miss  
load 1100 Hit!

Lookup:

• ~~Index~~ into \$

➔ Check tags

➔ Check valid bits

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

➔ LRU Pointer

# Pros and Cons of Full Associativity

- + No more conflicts!
- + Excellent utilization!
- But either:
  - Parallel Reads
    - lots of reading!
  - Serial Reads
    - lots of waiting



$$t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} * t_{\text{miss}}$$

$$= 4 + 5\% \times 100$$
$$= 9 \text{ cycles}$$

$$= 6 + 3\% \times 100$$
$$= 9 \text{ cycles}$$





# Pros & Cons

## Direct Mapped

## Fully Associative

Tag Size

Smaller

Larger

SRAM Overhead

Less

More

Controller Logic

Less

More

Speed

Faster

Slower

Price

Less

More

Scalability

Very

Not Very

# of conflict misses

Lots

Zero

Hit Rate

Low

High

Pathological Cases

Common

?

# Reducing Conflict Misses with Set-Associative Caches

Not too conflict-y. Not too slow.

... Just Right!



# 8 byte, 2-way set associative Cache

XXXX

## CACHE

index	V	tag	data
0	0	xx	E   F
1	0	xx	C   D

V	tag	data
0	xx	N   O
0	xx	P   Q

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

What should the **offset** be?

What should the **index** be?

What should the **tag** be?

# Clicker Question

XXXXX

5 bit address

2 byte block size

24 byte, 3-Way Set Associative CACHE

index	V	tag	data
00	0	?	X   Y
01	0	?	X   Y
10	0	?	X   Y
11	0	?	X   Y

V	tag	data
0	?	X'   Y'
0	?	X'   Y'
0	?	X'   Y'
0	?	X'   Y'

V	tag	data
0	?	X''   Y''
0	?	X''   Y''
0	?	X''   Y''
0	?	X''   Y''

How many tag bits?

- A) 0
- B) 1
- C) 2
- D) 3
- E) 4

# 8 byte, 2-way set associative Cache

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

XXXX  
tag | offset  
index

## CACHE

index	V	tag	data
0	0	xx	X   X
1	0	xx	X   X

load 1100  
load 1101  
load 0100  
load 1100

Miss

Lookup:

- ➔ Index into \$
- ➔ Check tag
- ➔ Check valid bit

↑ LRU Pointer

# 8 byte, 2-way set associative Cache

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

XXXX  
tag | offset  
index

## CACHE

index	V	tag	data
0	1	11	N   O
1	0	xx	X   X

V	tag	data
0	xx	X   X
0	xx	X   X

load 1100 Miss  
 load 1101 Hit!  
 load 0100  
 load 1100

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

↑ LRU Pointer

# 8 byte, 2-way set associative Cache

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

XXXX  
tag | offset  
index

## CACHE

index	V	tag	data
0	1	11	N   O
1	0	xx	X   X

V	tag	data
0	xx	X   X
0	xx	X   X

load 1100 Miss  
 load 1101 Hit!  
 load 0100 Miss  
 load 1100

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

↑ LRU Pointer

# 8 byte, 2-way set associative Cache

## MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

XXXX  
tag | offset  
index

## CACHE

index	V	tag	data
0	1	11	N   O
1	0	xx	X   X

V	tag	data
1	01	E   F
0	xx	X   X

load 1100 Miss  
load 1101 Hit!  
load 0100 Miss  
load 1100 Hit!

Lookup:  
 → Index into \$  
 → Check tag  
 → Check valid bit

↑ LRU Pointer



# Eviction Policies

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

# Clicker Question

This cache is a direct-mapped, 8B cache with 2B blocks. Addresses are 4 bits.

This cache was filled via access to addresses in the order: 0000, 0010, 0100, 0110.

Cache contents (prior to access)				address of cache access	outcome of cache access
Set 00	Set 01	Set 10	Set 11		
0000   0001	0010   0011	0100   0101	0110   0111	1100	cold miss
				1010	
				1011	
				0011	
				0101	
				1100	
				1001	

How many capacity misses are there? (A) 0 (B) 1 (C) 2 (D) 3 (E) 4

# Clicker Answer

This cache is a direct-mapped, 8B cache with 2B blocks. Addresses are 4 bits.

This cache was filled via access to addresses in the order: 0000, 0010, 0100, 0110.

Cache contents (prior to access)				address of cache access	outcome of cache access
Set 00	Set 01	Set 10	Set 11		
0000   0001	0010   0011	0100   0101	0110   0111	1100	cold miss
		1100   1101		1010	cold miss
	1010   1011			1011	Hit
				0011	capacity miss
	0010   0011			0101	capacity miss
		0100   0101		1100	conflict miss
		1100   1101		1001	cold miss

# Clicker Explanation

0 0000 | 0001

1 0010 | 0011

2 0100 | 0101

3 0110 | 0111

0-3 are the initial accesses that populate the cache

Now the question starts:

4 1100 | 1101

cold miss

5 1010 | 1011

cold miss

1010 | 1011

hit (this is a repeat from #5, not distinct)

6 0010 | 0011

capacity miss – we haven't seen this since **time #1**

7 0100 | 0101

capacity miss – we haven't seen this since **time #2**

8 1100 | 1101

conflict miss – there are 4 cache lines in this cache, and we brought this line in 4 ago – it would still be there if F.A.

9 1000 | 1001

cold miss

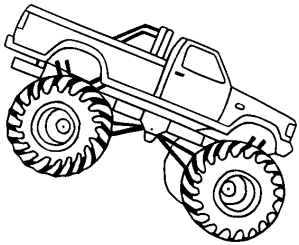
# Misses: the Three C's



Cold (compulsory) Miss:  
never seen this address before

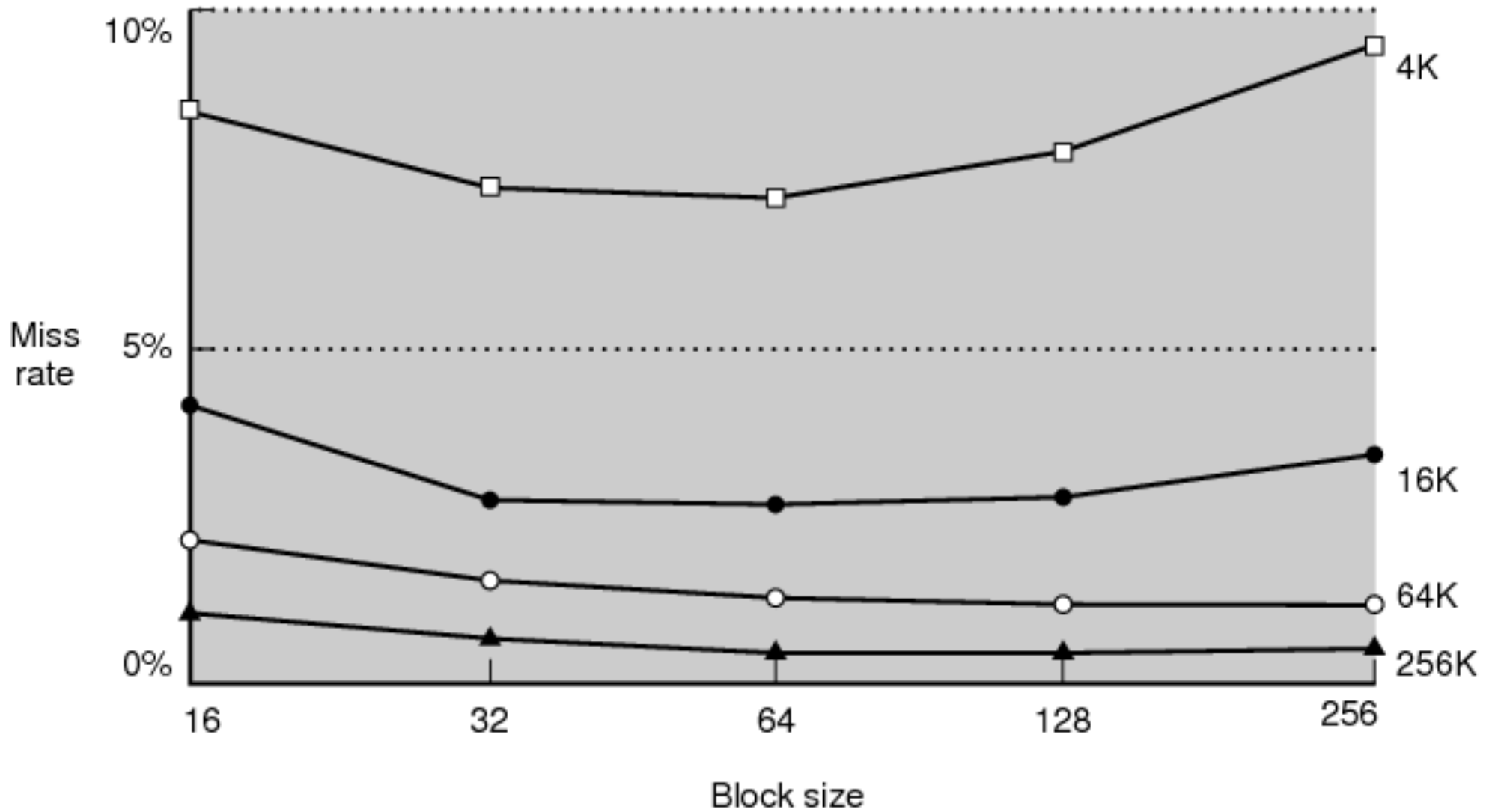


Conflict Miss:  
cache associativity is too low



Capacity Miss:  
cache is too small

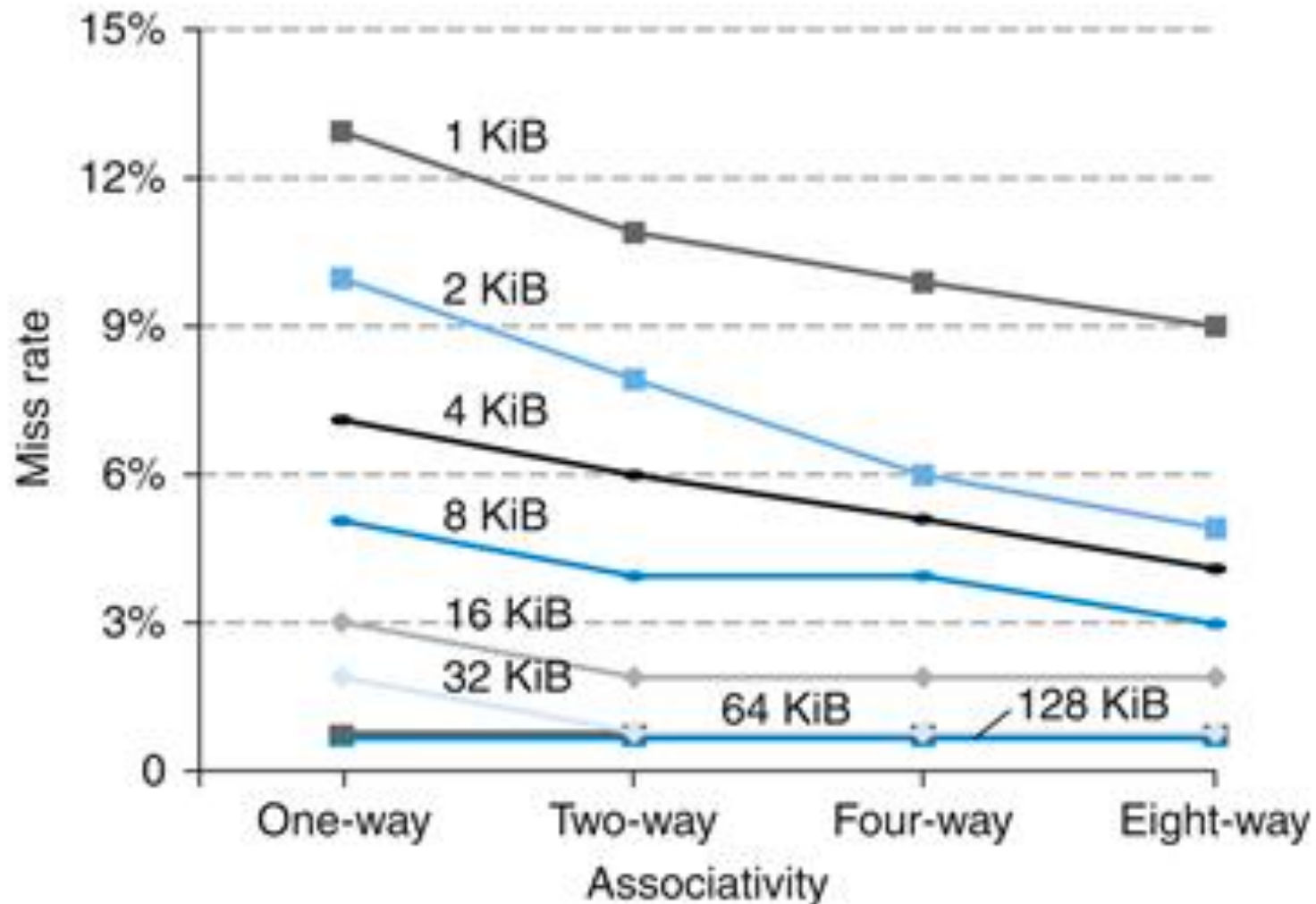
# Miss Rate vs. Block Size



# Block Size Tradeoffs

- For a given total cache size,  
Larger block sizes mean....
  - fewer lines
  - so fewer tags, less overhead
  - and fewer cold misses (within-block “prefetching”)
- But also...
  - fewer blocks available (for scattered accesses!)
  - so more conflicts
  - can decrease performance if working set can't fit in \$
  - and larger miss penalty (time to fetch block)

# Miss Rate vs. Associativity





# ABCs of Caches

$$t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} * t_{\text{miss}}$$

+ Associativity:

↓ conflict misses 😊

↑ hit time ☹️

+ Block Size:

↓ cold misses 😊

↑ conflict misses ☹️

+ Capacity:

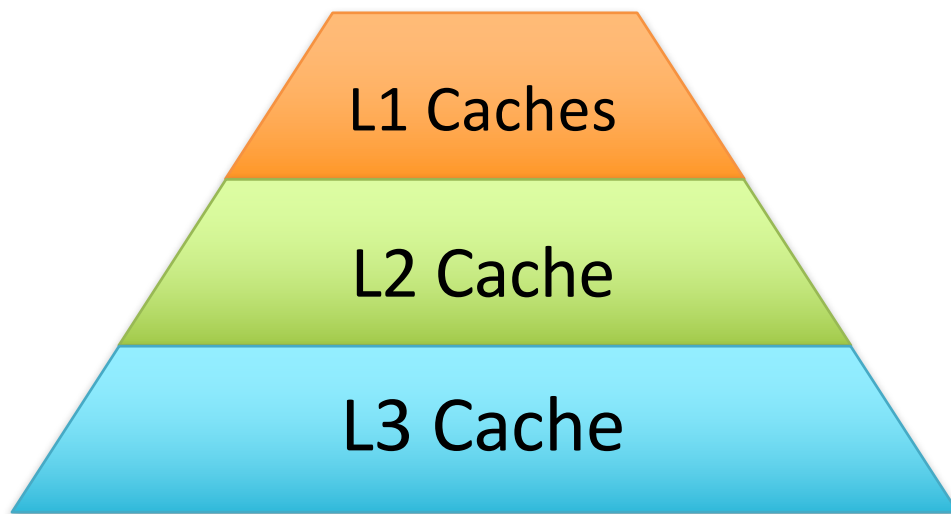
↓ capacity misses 😊

↑ hit time ☹️



# Which caches get what properties?

$$t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} * t_{\text{miss}}$$



Fast



Big

*Design with speed in mind*



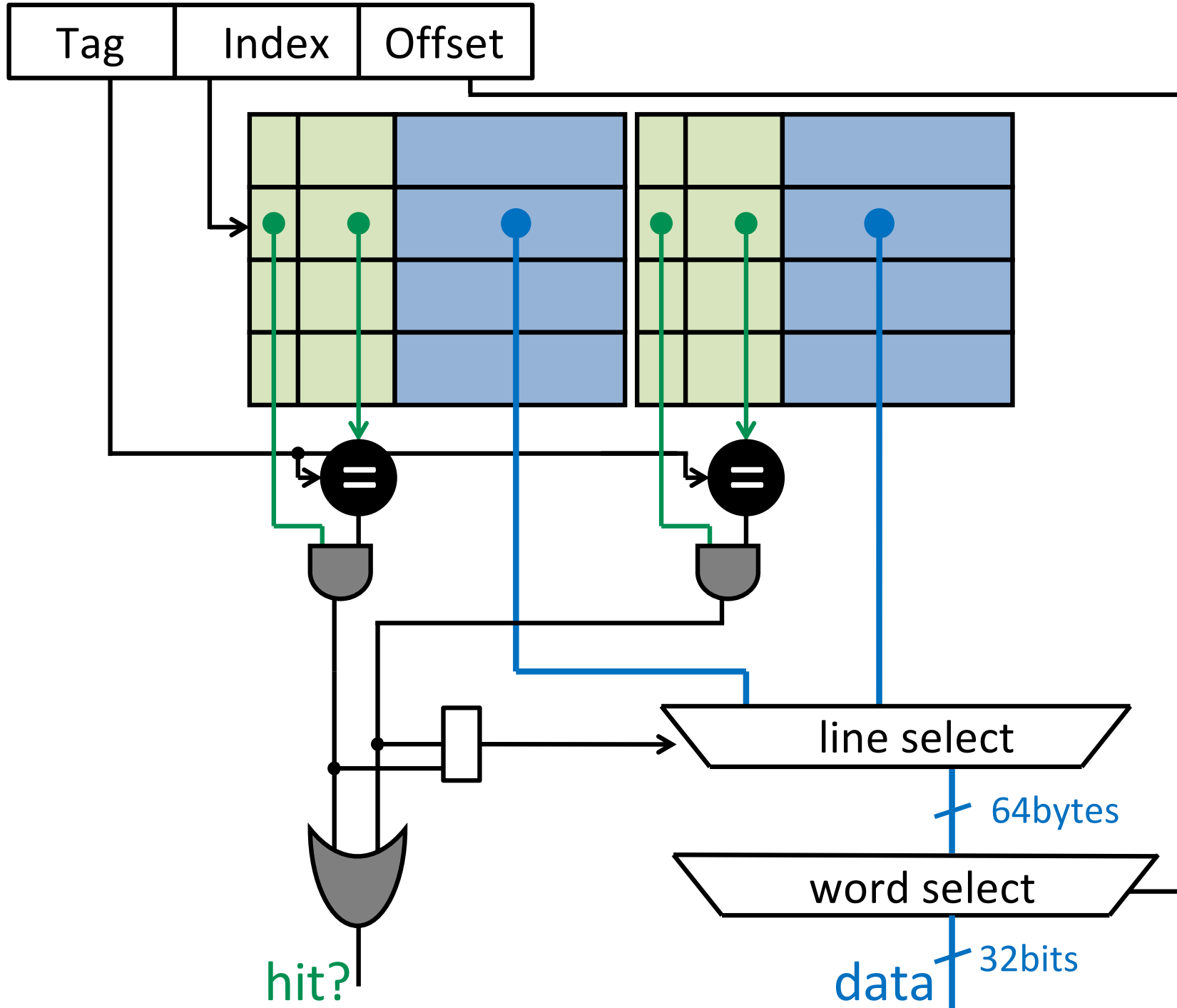
*More Associative  
Bigger Block Sizes  
Larger Capacity*

*Design with miss rate in mind*

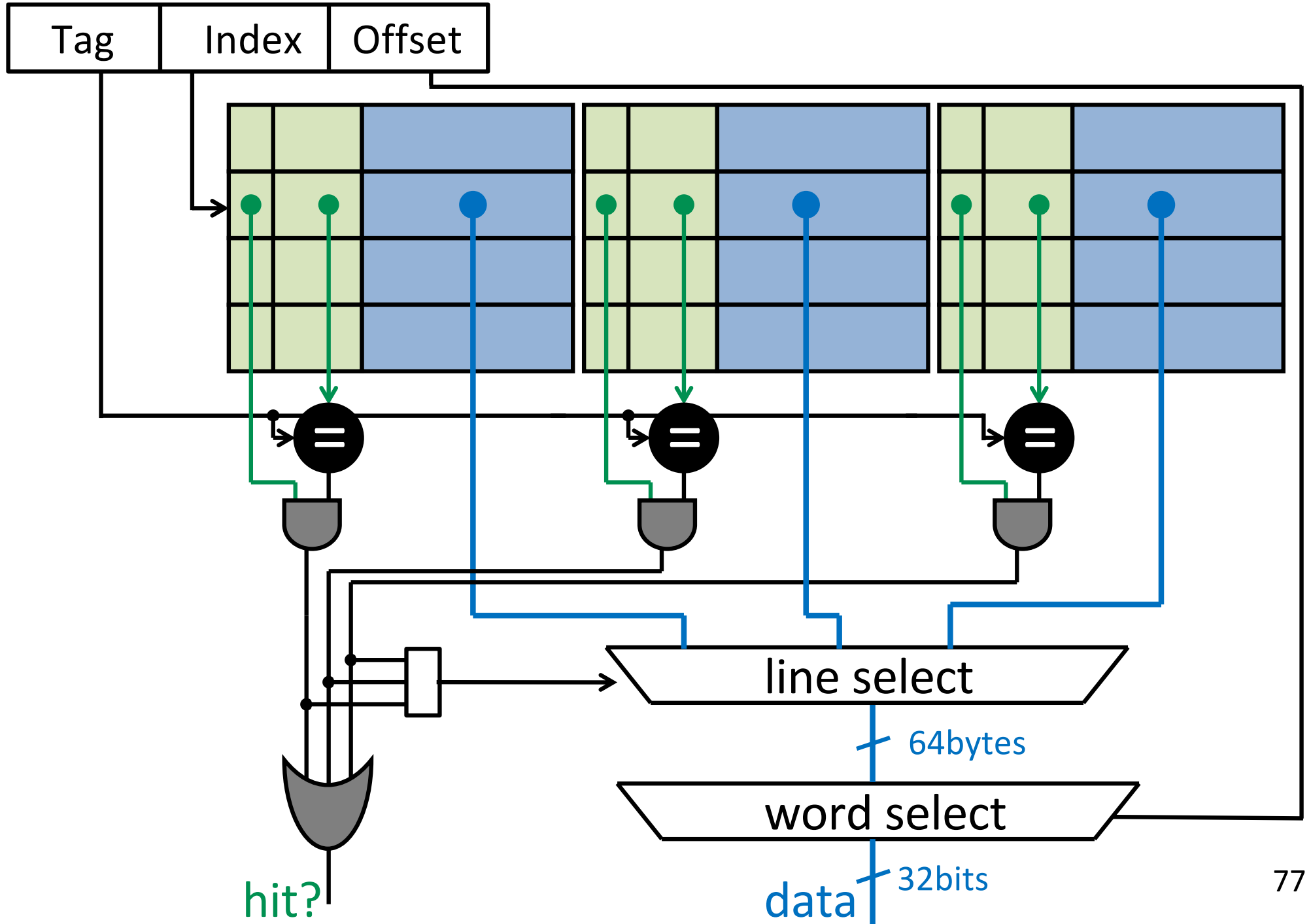
# Roadmap

- Things we have covered:
  - The Need for Speed
  - Locality to the Rescue!
  - Calculating average memory access time
  - \$ Misses: Cold, Conflict, Capacity
  - \$ Characteristics: Associativity, Block Size, Capacity
- Things we will now cover:
  - Cache Figures
  - Cache Performance Examples
  - Writes

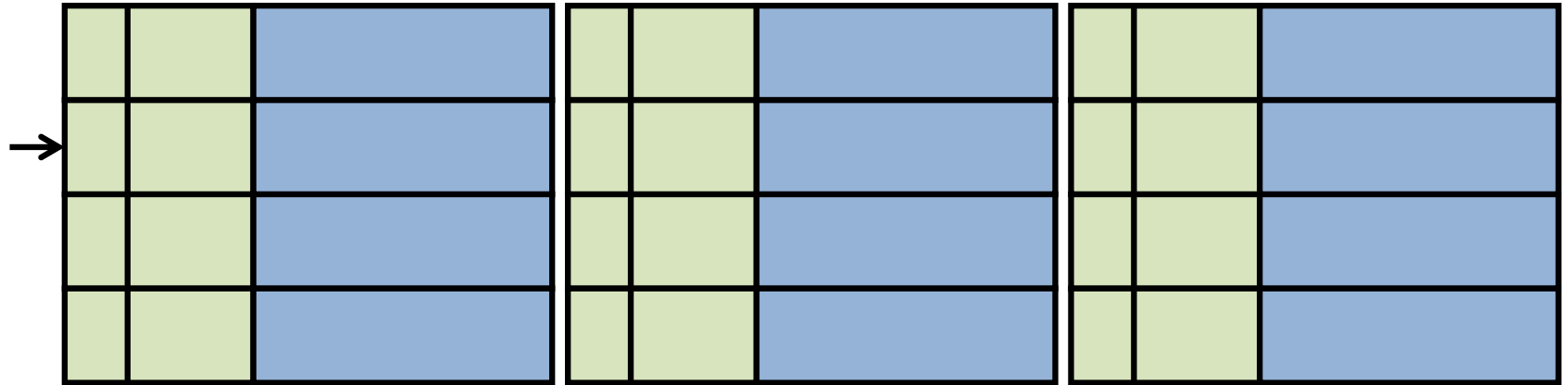
# 2-Way Set Associative Cache (Reading)



# 3-Way Set Associative Cache (Reading)



# How Big is the Cache?



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

Question: How big is cache?

- ***Data only?***

(what we usually mean when we ask “how big” is the cache)

- ***Data + overhead?***



# Performance Calculation with \$ Hierarchy

$$t_{\text{avg}} = t_{\text{hit}} + \%_{\text{miss}} * t_{\text{miss}}$$

- **Parameters**

- Reference stream: all loads
- D\$:  $t_{\text{hit}} = 1\text{ns}$ ,  $\%_{\text{miss}} = 5\%$
- L2:  $t_{\text{hit}} = 10\text{ns}$ ,  $\%_{\text{miss}} = 20\%$  (local miss rate)
- Main memory:  $t_{\text{hit}} = 50\text{ns}$

- **What is  $t_{\text{avgD\$}}$  without an L2?**

- $t_{\text{missD\$}} =$
- $t_{\text{avgD\$}} =$

- **What is  $t_{\text{avgD\$}}$  with an L2?**

- $t_{\text{missD\$}} =$
- $t_{\text{avgL2}} =$
- $t_{\text{avgD\$}} =$



# Performance Calculation with \$ Hierarchy

- **Parameters**

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

- Reference stream: all loads

- D\$:  $t_{hit} = 1ns$ ,  $\%_{miss} = 5\%$

- L2:  $t_{hit} = 10ns$ ,  $\%_{miss} = 20\%$  (local miss rate)

- Main memory:  $t_{hit} = 50ns$

- **What is  $t_{avgD\$}$  without an L2?**

- $t_{missD\$} = t_{hitM}$

- $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{hitM} = 1ns + (0.05 * 50ns) = 3.5ns$

- **What is  $t_{avgD\$}$  with an L2?**

- $t_{missD\$} = t_{avgL2}$

- $t_{avgL2} = t_{hitL2} + \%_{missL2} * t_{hitM} = 10ns + (0.2 * 50ns) = 20ns$

- $t_{avgD\$} = t_{hitD\$} + \%_{missD\$} * t_{avgL2} = 1ns + (0.05 * 20ns) = 2ns$



# Performance Summary

Average memory access time (AMAT) depends on:

- cache architecture and size
- Hit and miss rates
- Access times and miss penalty

Cache design a very complex problem:

- Cache size, block size (aka line size)
- Number of ways of set-associativity (1, N,  $\infty$ )
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

# Takeaway

Direct Mapped → fast, but low hit rate

Fully Associative → higher hit cost, higher hit rate

Set Associative → middleground

Line size matters. Larger cache lines can increase performance due to prefetching. BUT, can also decrease performance if **working set** size cannot fit in cache.

Cache performance is measured by the average memory access time (AMAT), which depends on cache architecture and size, but also on the access time for hit, miss penalty, and hit rate.

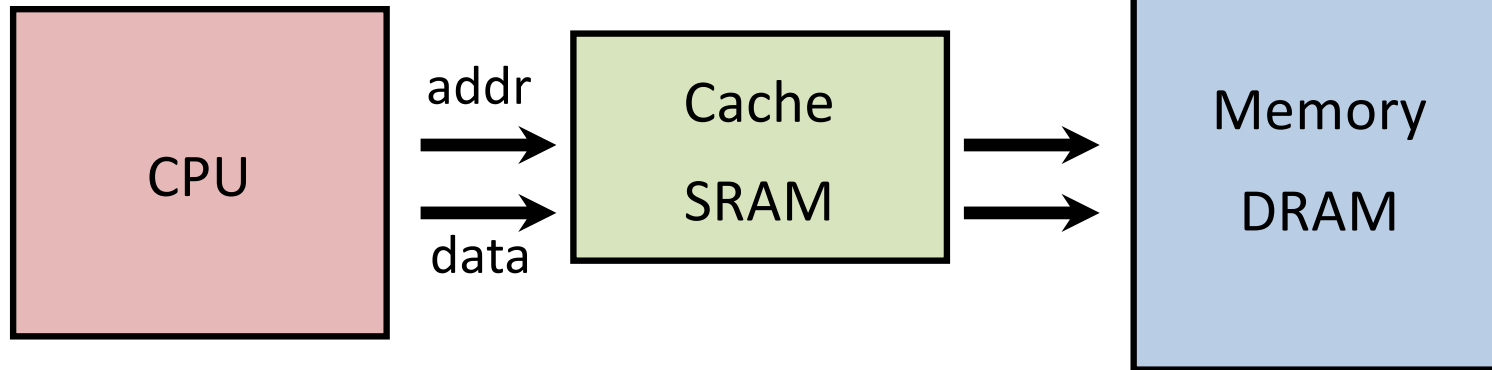
# What about Stores?

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 evict the block (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)

# Cache 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

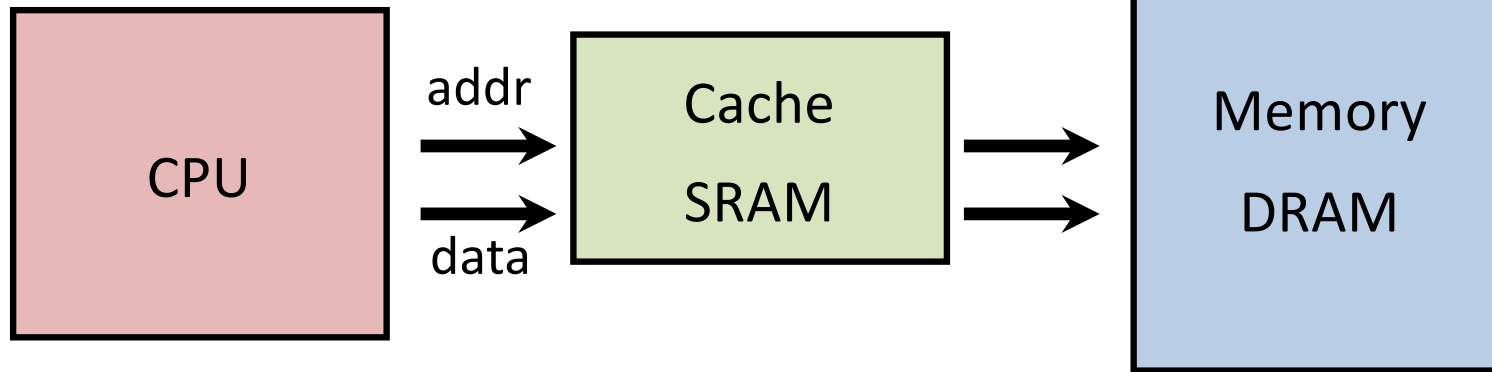
**Write-Back**

CPU writes only to cache

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

# 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

# Write-Through Stores

16 byte, byte-addressed memory

4 byte, fully-associative cache:

2-byte blocks, write-allocate

4 bit addresses:

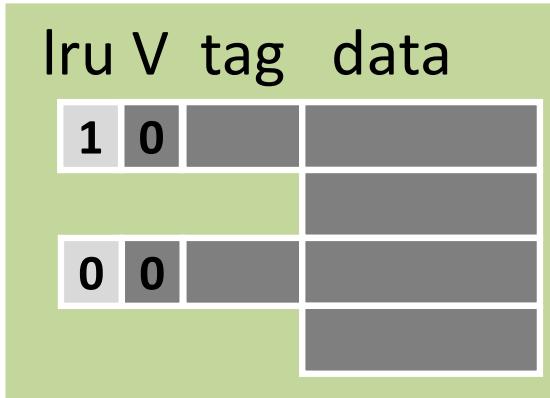
3 bit tag, 1 bit offset

```

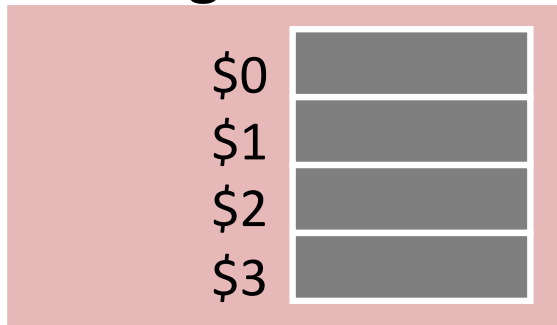
Instructions:
LB $1 ← M[ 1 ]
LB $2 ← M[ 7 ]
SB $2 → M[ 0 ]
SB $1 → M[ 5 ]
LB $2 ← M[ 10 ]
SB $1 → M[ 5 ]
SB $1 → M[ 10 ]
    
```

## 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



## Register File



## Cache

**Misses:** 0  
**Hits:** 0  
**Reads:** 0  
**Writes:** 0

# Write-Through (REF 1)

Instructions:

➔ 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 ]

**Register File**

\$0	
\$1	
\$2	
\$3	

lru V tag data

1	0		
0	0		

**Cache**

**Misses:** 0  
**Hits:** 0  
**Reads:** 0  
**Writes:** 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

# Write-Through (REF 1)

Instructions:

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

Register File

\$0	
\$1	29
\$2	
\$3	

lru	V	tag	data
0	1	000	78
			29
1	0		

Cache

**Misses:** 1  
**Hits:** 0  
**Reads:** 1  
**Writes:** 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

Addr: 0001

block offset



# Write-Through (REF 2)

Instructions:

```

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

**Register File**

\$0	
\$1	29
\$2	
\$3	

lru V tag data

0	1	000	78
			29
1	0		

**Cache**

**Misses:** 1  
**Hits:** 0  
**Reads:** 1  
**Writes:** 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

# Write-Through (REF 2)

Instructions:

```

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Addr: 0111

lru	V	tag	data
0	1	000	78
			29
1	1	011	162
			173

Cache

**Misses:** 2  
**Hits:** 0  
**Reads:** 2  
**Writes:** 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

# Write-Through (REF 3)

Instructions:

```

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

**Register File**

\$0	
\$1	29
\$2	173
\$3	

lru V tag data

1	1	000	78
			29
0	1	011	162
			173

**Cache**

**Misses:** 2  
**Hits:** 0  
**Reads:** 2  
**Writes:** 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

# Write-Through (REF 3)

Instructions:

```

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Addr: 0000

lru	V	tag	data
0	1	000	173
			29
1	1	011	162
			173

Cache

**Misses:** 2  
**Hits:** 1  
**Reads:** 2  
**Writes:** 1

Memory

0	173
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

# Write-Through (REF 4)

Instructions:

```

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

Addr: 0101

lru V tag data

0	1	000	173
			29
1	1	010	71
			150

Memory

0	173
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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

**Misses:** 2  
**Hits:** 1  
**Reads:** 2  
**Writes:** 1

# Write-Through (REF 4)

Instructions:

```

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

lru	V	tag	data
1	1	000	173
			29
0	1	010	71
			29

Memory

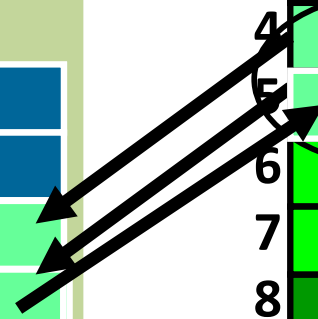
0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

**Misses:** 3  
**Hits:** 1  
**Reads:** 3  
**Writes:** 2



# Write-Through (REF 5)

Instructions:

```

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

Addr: 1010

lru V tag data

1	1	000	173
			29
0	1	010	71
			29

Memory

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

**Misses:** 3  
**Hits:** 1  
**Reads:** 3  
**Writes:** 2

# Write-Through (REF 5)

Instructions:

```

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

Addr: 1010

lru V tag data

0	1	101	33
			28
1	1	010	71
			29

Cache

**Misses:** 4  
**Hits:** 1  
**Reads:** 4  
**Writes:** 2

Memory

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

Register File

\$0	
\$1	29
\$2	33
\$3	



# Write-Through (REF 6)

Instructions:

```

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

Register File

\$0	
\$1	29
\$2	33
\$3	

Addr: 0101

lru	V	tag	data
0	1	101	33
			28
1	1	010	71
			29

Cache

**Misses:** 4  
**Hits:** 1  
**Reads:** 4  
**Writes:** 2

Memory

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

# Write-Through (REF 6)

Instructions:

```

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

**Register File**

\$0	
\$1	29
\$2	33
\$3	

**Cache**

lru	V	tag	data
0	1	101	33
			28
1	1	010	71
			29

**Memory**

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

**Misses:** 4  
**Hits:** 2  
**Reads:** 4  
**Writes:** 3

# Write-Through (REF 7)

Instructions:

```

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

Register File

\$0	
\$1	29
\$2	33
\$3	

Addr: 1011

lru	V	tag	data
0	1	101	33
			28
1	1	010	71
			29

Cache

**Misses:** 4  
**Hits:** 2  
**Reads:** 4  
**Writes:** 3

Memory

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

# Write-Through (REF 7)

Instructions:

LB \$1 ← M[ 1 ] **M**  
 LB \$2 ← M[ 7 ] **M**  
 SB \$2 → M[ 0 ] **Hit**  
 SB \$1 → M[ 5 ] **M**  
 LB \$2 ← M[ 10 ] **M**  
 SB \$1 → M[ 5 ] **Hit**  
 → SB \$1 → M[ 10 ] **Hit**

**Register File**

\$0	
\$1	29
\$2	33
\$3	

lru V tag data

0	1	101	29
			28
1	1	010	71
			29

**Cache**

**Misses:** 4  
**Hits:** 3  
**Reads:** 4  
**Writes:** 4

**Memory**

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

# Summary: Write Through

Write-through policy with write allocate

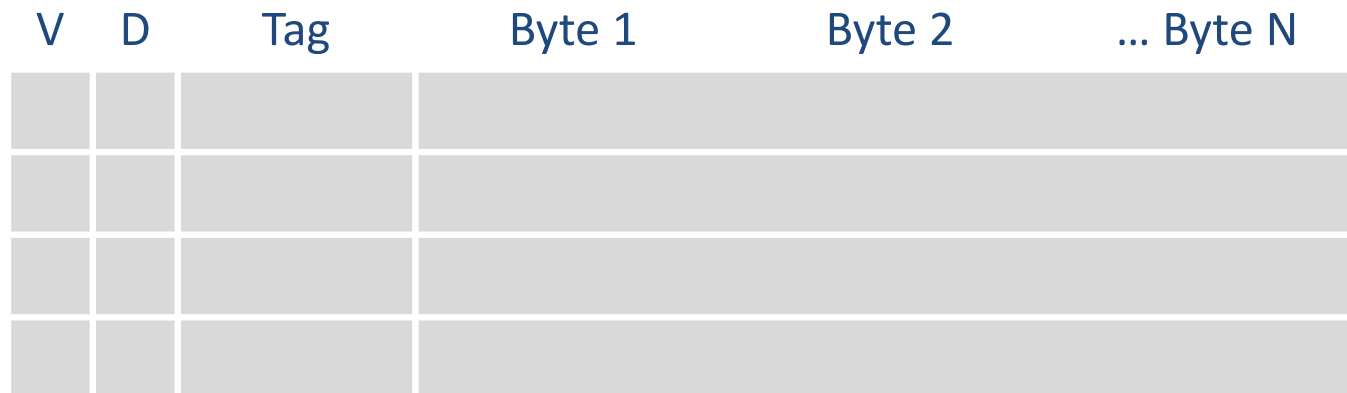
- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory

# Next Goal: Write-Through vs. Write-Back

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

- Keep the current copy in cache, and update memory when data is **evicted** (**write-back policy**)
- Write-back all evicted lines?
  - No, only written-to blocks

# Write-Back Meta-Data (Valid, Dirty Bits)



- $V = 1$  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  $D = 0$ ,  $V = 0$

# Write-back Example

- Example: How does a **write-back** cache work?
- Assume **write-allocate**



# Handling Stores (Write-Back)

16 byte, byte-addressed memory

4 byte, fully-associative cache:

2-byte blocks, write-allocate

4 bit addresses:

3 bit tag, 1 bit offset

```

Instructions:
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 ]
    
```

## 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

lru	V	d	tag	data
1	0			
0	0			

## Register File

\$0	
\$1	
\$2	
\$3	

## Cache

Misses: 0

Hits: 0

# Write-Back (REF 1)

Instructions:

→ 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 ]

Register File

\$0	
\$1	
\$2	
\$3	

lru V d tag data

1	0			
0	0			

Cache

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

# Write-Back (REF 1)

Instructions:

➔ LB \$1 ← M[ 1 ] **M**  
LB \$2 ← M[ 7 ]  
SB \$2 → M[ 0 ]  
SB \$1 → M[ 5 ]  
LB \$2 ← M[ 10 ]  
SB \$1 → M[ 5 ]  
SB \$1 → M[ 10 ]

lru	V	d	tag	data
0	1	0	000	78
				29
1	0			

Addr: 0001

block offset

Register File

\$0	
\$1	29
\$2	
\$3	

Cache

Misses: 1  
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

# Write-Back (REF 1)

Instructions:

→ LB \$1 ← M[ 1 ] **M**

LB \$2 ← M[ 7 ]

SB \$2 → M[ 0 ]

SB \$1 → M[ 5 ]

LB \$2 ← M[ 10 ]

SB \$1 → M[ 5 ]

SB \$1 → M[ 10 ]

lru V d tag data

0	1	0	000	78
				29
1	0			

Register File

\$0	
\$1	29
\$2	
\$3	

Cache

Misses: 1  
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

# Write-Back (REF 2)

Instructions:

LB \$1 ← M[ 1 ] **M**

← LB \$2 ← M[ 7 ]

SB \$2 → M[ 0 ]

SB \$1 → M[ 5 ]

LB \$2 ← M[ 10 ]

SB \$1 → M[ 5 ]

SB \$1 → M[ 10 ]

Register File

\$0	
\$1	29
\$2	
\$3	

lru V d tag data

0	1	0	000	78
				29
1	0			

Cache

Misses: 1

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

# Write-Back (REF 2)

Instructions:

```

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

Addr: 0111

lru	V	d	tag	data
1	1	0	000	78
				29
0	1	0	011	162
				173

block offset

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 2  
Hits: 0

# Write-Back (REF 3)

Instructions:

```

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

lru V d tag data

1	1	0	000	78
				29
0	1	0	011	162
				173

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 2  
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

# Write-Back (REF 3)

Instructions:

LB \$1 ← M[ 1 ] **M**

LB \$2 ← M[ 7 ] **M**

→ SB \$2 → M[ 0 ] **Hit**

SB \$1 → M[ 5 ]

LB \$2 ← M[ 10 ]

SB \$1 → M[ 5 ]

SB \$1 → M[ 10 ]

Addr: 0000

lru V d tag data

0	1	1	000	173
				29
1	1	0	001	162
				173

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 2

Hits: 1

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



# Write-Back (REF 4)

Instructions:

```

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

lru V d tag data

0	1	1	000	173
				29
1	1	0	011	162
				173

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 2  
Hits: 1

# Write-Back (REF 4)

Instructions:

```

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

Addr: 0101

lru	V	d	tag	data
1	1	1	000	173
				29
0	1	1	010	71
				29

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

Register File

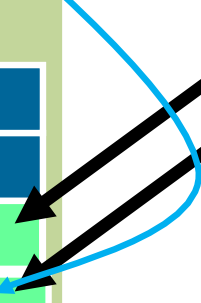
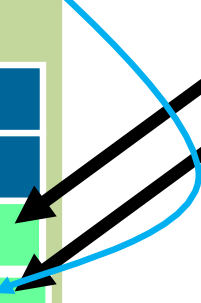
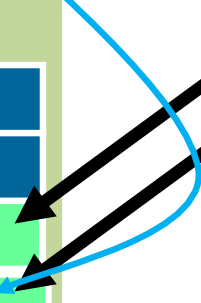
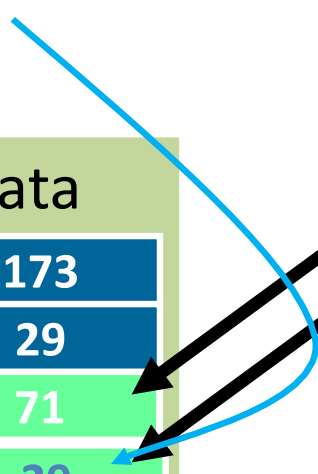
\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 3  
Hits: 1



Addr: 0101



# Write-Back (REF 5)

Instructions:

```

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

Addr: 1010

lru	V	d	tag	data
1	1	1	000	173
				29
0	1	1	010	71
				29

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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 3  
Hits: 1

# Write-Back (REF 5)

Instructions:

```

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

Addr: 1010

lru V d tag data

1	1	1	000	173
				29
0	1	1	010	71
				29

Memory

0	173
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

Register File

\$0	
\$1	29
\$2	173
\$3	

Cache

Misses: 3

Hits: 1

# Write-Back (REF 5)

Instructions:

```

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

Addr: 1010

lru	V	d	tag	data
0	1	0	101	33
				28
1	1	1	010	71
				29

Memory

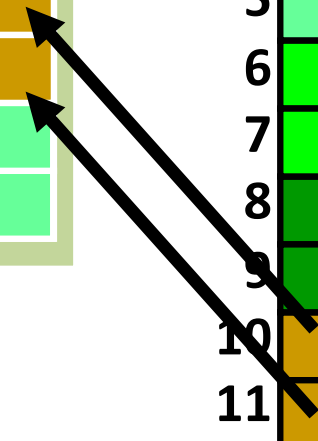
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

\$0	
\$1	29
\$2	33
\$3	

Cache

Misses: 4  
Hits: 1



# Write-Back (REF 6)

Instructions:

```

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

Addr: **0101**

lru V d tag data

0	1	0	101	33
				28
1	1	1	010	71
				29

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

Register File

\$0	
\$1	29
\$2	33
\$3	

Cache

Misses: 4

Hits: 1

# Write-Back (REF 6)

Instructions:

```

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

lru V d tag data

1	1	0	101	33
				28
0	1	1	010	71
				29

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

Register File

\$0	
\$1	29
\$2	33
\$3	

Cache

Misses: 4  
Hits: 2

# Write-Back (REF 7)

Instructions:

```

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

lru V d tag data

1	1	0	101	33
				28
0	1	1	010	71
				29

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

Register File

\$0	
\$1	29
\$2	33
\$3	

Cache

Misses: 4  
Hits: 2



# Write-Back (REF 7)

Instructions:

LB \$1 ← M[ 1 ] **M**  
 LB \$2 ← M[ 7 ] **M**  
 SB \$2 → M[ 0 ] **Hit**  
 SB \$1 → M[ 5 ] **M**  
 LB \$2 ← M[ 10 ] **M**  
 SB \$1 → M[ 5 ] **Hit**  
 → SB \$1 → M[ 10 ] **Hit**

lru V d tag data

0	1	1	101	29
				28
1	1	1	010	71
				29

Register File

\$0	
\$1	29
\$2	33
\$3	

Cache

Misses: 4  
 Hits: 3

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

# Write-Back (REF 8,9)

**M**  
 Instructions: **M**  
 ... **Hit**  
 SB \$1 → M[ 5 ] **M**  
 LB \$2 ← M[ 10 ] **M**  
 SB \$1 → M[ 5 ] **Hit**  
 SB \$1 → M[ 10 ] **Hit**  
**SB \$1 → M[ 5 ]**  
**SB \$1 → M[ 10 ]**

lru	V	d	tag	data
0	1	1	101	29
				28
1	1	1	010	71
				29

## 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

## Register File

\$0	
\$1	29
\$2	33
\$3	

## Cache

**Misses: 4**  
**Hits: 3**

# Write-Back (REF 8,9)

**M**  
 Instructions: **M**  
 ... **Hit**  
 SB \$1 → M[ 5 ] **M**  
 LB \$2 ← M[ 10 ] **M**  
 SB \$1 → M[ 5 ] **Hit**  
 SB \$1 → M[ 10 ] **Hit**  
**SB \$1 → M[ 5 ] Hit**  
**SB \$1 → M[ 10 ] Hit**

lru	V	d	tag	data
0	1	1	101	29
				28
1	1	1	010	71
				29

## 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

## Register File

\$0	
\$1	29
\$2	33
\$3	

## Cache

**Misses: 4**  
**Hits: 5**

# How Many Memory References?

Write-back performance

- How many reads?
- How many writes?



# How Many Memory References?

## Write-back performance

- How many reads?
  - Each miss (read or write) reads a block from mem
  - 4 misses  $\rightarrow$  8 mem reads
- How many writes?
  - *Some* evictions write a block to mem
  - 1 dirty eviction  $\rightarrow$  2 mem writes
  - (+ 2 dirty evictions later  $\rightarrow$  +4 mem writes)

# Write-back vs. Write-through Example

Assume: large associative cache, 16-byte lines

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

N words

Write-through:  $n/16$  reads

**n writes**

Write-back:  $n/16$  reads

**1 write**

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

Write-through:  $2 \times n/16$  reads

**n writes**

Write-back:  $2 \times n/16$  reads

**n write**

# So is write back just better?

**Short Answer:** Yes (fewer writes is a good thing)

**Long Answer:** It's complicated.

- Evictions require entire line be written back to memory (vs. just the data that was written)
- Write-back can lead to incoherent caches on multi-core processors (later lecture)







# By the end of the cache lectures...

## MacBook Pro

Retina, Mid 2012

**Processor** 2.7 GHz Intel Core i7

**Memory** 16 GB 1600 MHz DDR3

**Graphics** NVIDIA GeForce GT 650M 1024 MB

**Serial Number** C02J70TTDKQ5

**Software** OS X 10.9.2 (13C64)

Model Name:	MacBook Pro
Model Identifier:	MacBookPro10,1
Processor Name:	Intel Core i7
Processor Speed:	2.7 GHz
Number of Processors:	1
Total Number of Cores:	4
L2 Cache (per Core):	256 KB
L3 Cache:	8 MB
Memory:	16 GB
Boot ROM Version:	MBP101.00EE.B02
SMC Version (system):	2.3f36
Serial Number (system):	C02J70TTDKQ5
Hardware UUID:	F588E08C-60BF-5B35-A087-07714C2B2D11

- 32 KB data + 32 KB instruction **L1 cache** (3 clocks) and 256 KB **L2 cache** (8 clocks) per core.
- Shared L3 cache includes the processor graphics (**LGA 1155**).
- 64-byte **cache** line size.

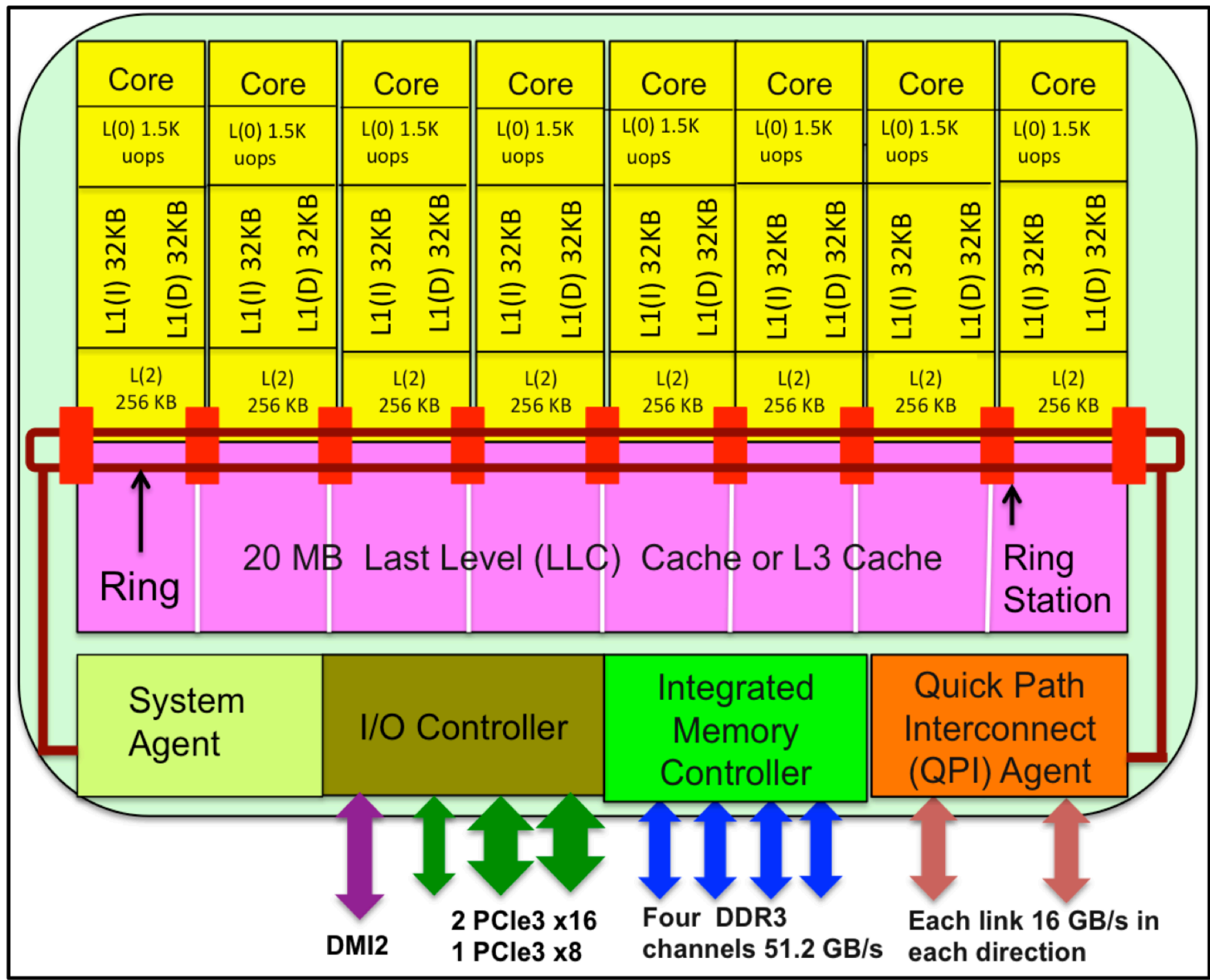


Figure 1. Schematic diagram of a Sandy Bridge processor.

# 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 → programs → os → hardware
- **NEXT: Multi-core processors are complicated**
  - Inconsistent views of memory
  - Extremely complex protocols, very hard to get right