Caches & Memory

CS 3410

Computer System Organization & Programming

These slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, and Sirer.

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:

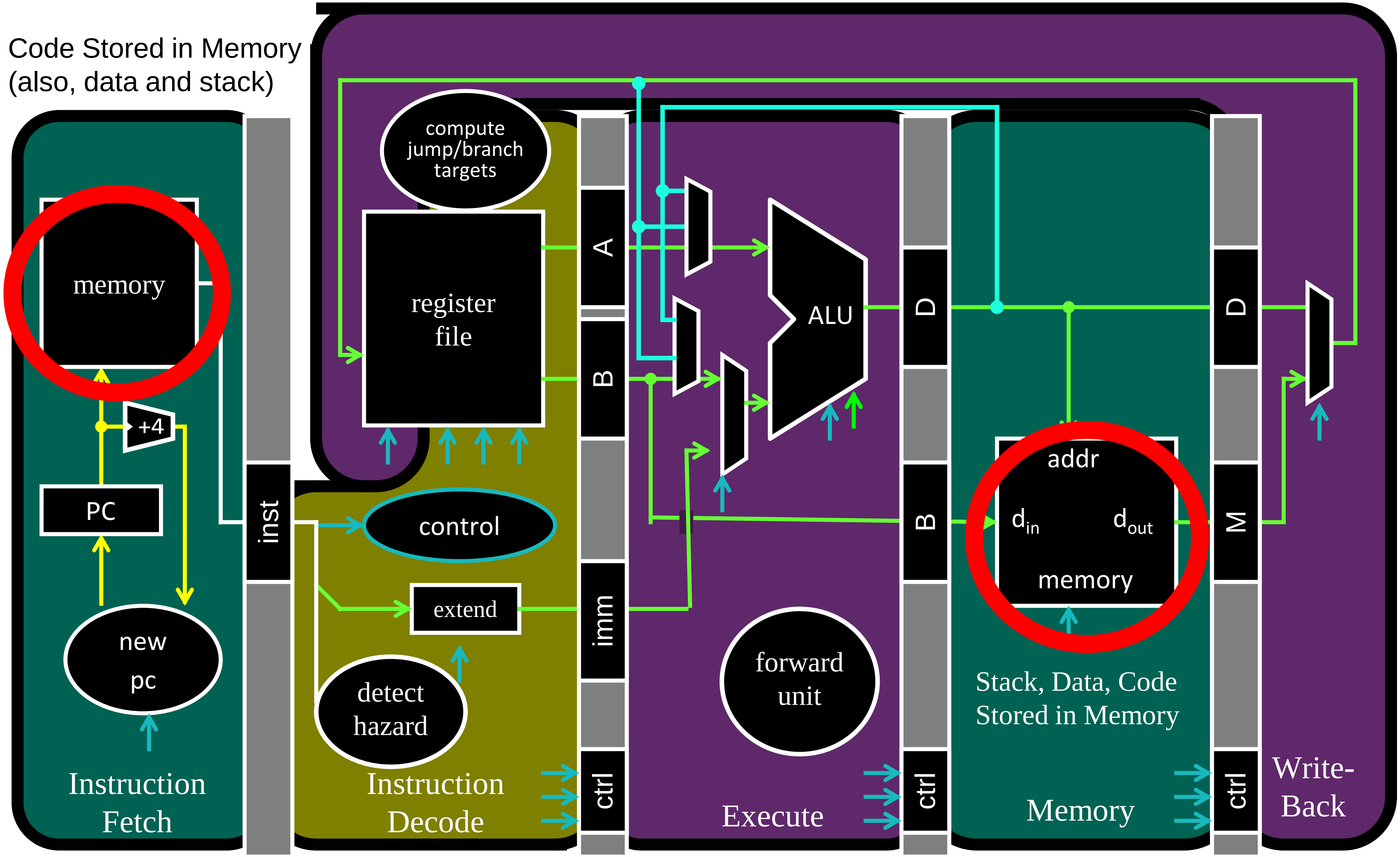
- Read data from memory (put in registers)
- Manipulate it
- Store it back to memory

MIPS Assembly

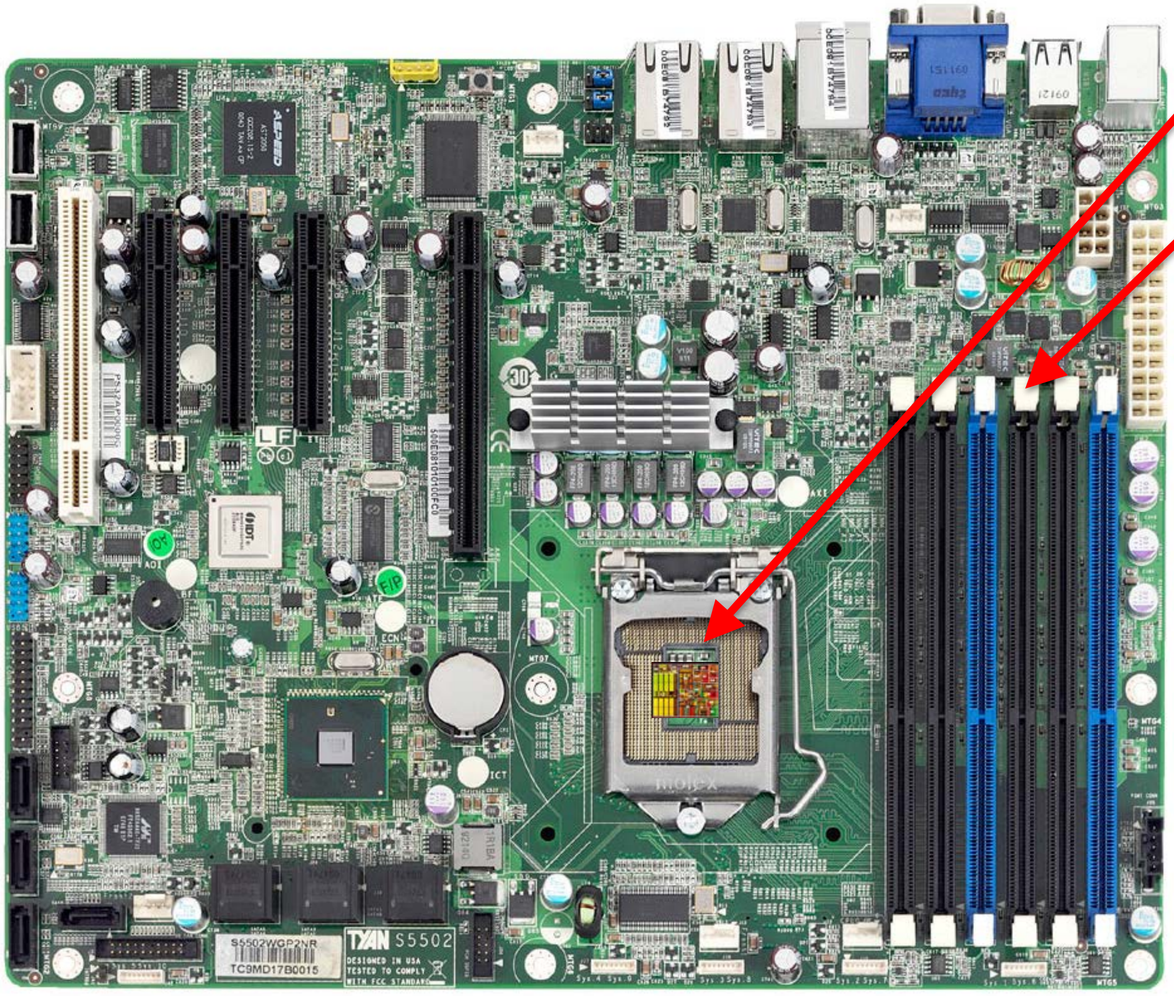
```
main:  addiu   $sp,$sp,-48
       sw      $31,44($sp)
       sw      $fp,40($sp)
       move    $fp,$sp
       sw      $4,48($fp)
       sw      $5,52($fp)
       la      $2,n
       lw      $2,0($2)
       sw      $2,28($fp)
       sw      $0,32($fp)
       li      $2,1
       sw      $2,24($fp)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

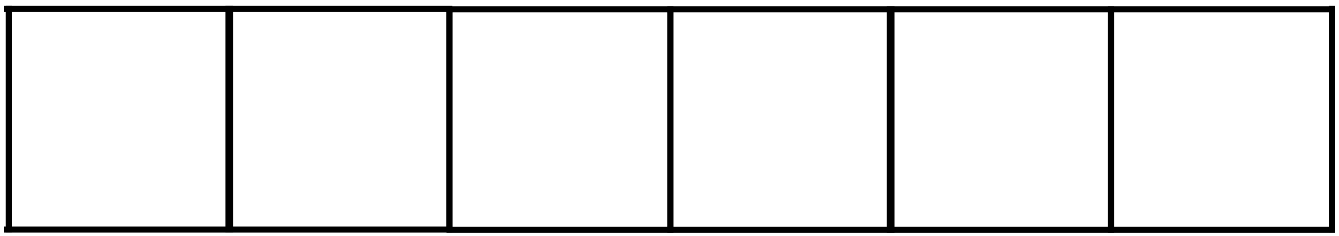
– slow

– far away

SandyBridge Motherboard, 2011
http://news.softpedia.com

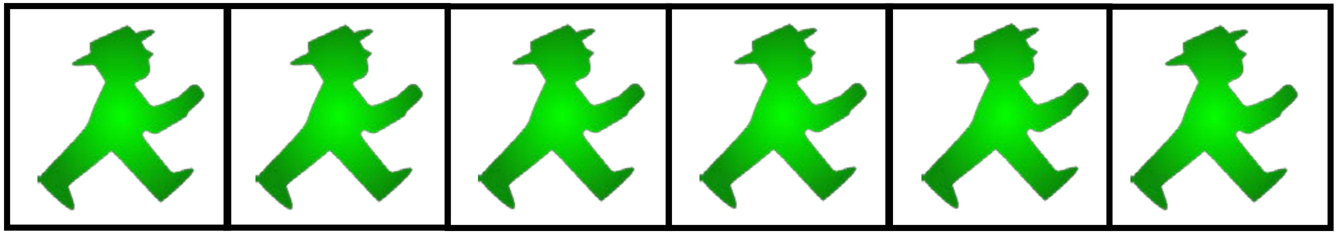
The Need for Speed

CPU Pipeline



The Need for Speed

CPU Pipeline



Instruction speeds:

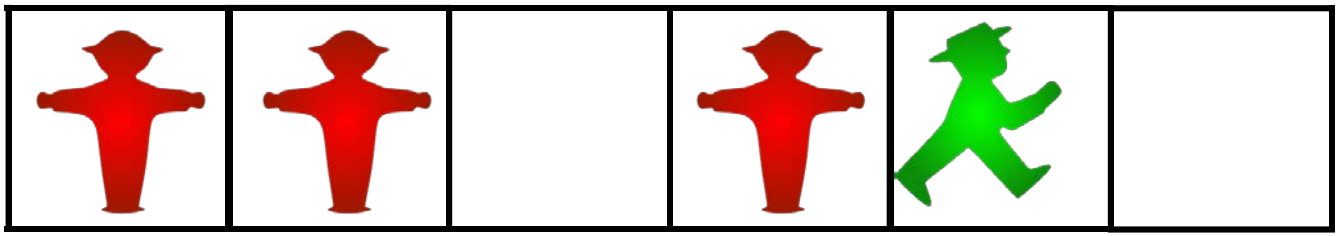
- **`add,sub,shift`**: 1 cycle
- **`mult`**: 3 cycles
- **`load/store`**: **100 cycles**

  off-chip 50(-70)ns

  2(-3) GHz processor → 0.5 ns clock

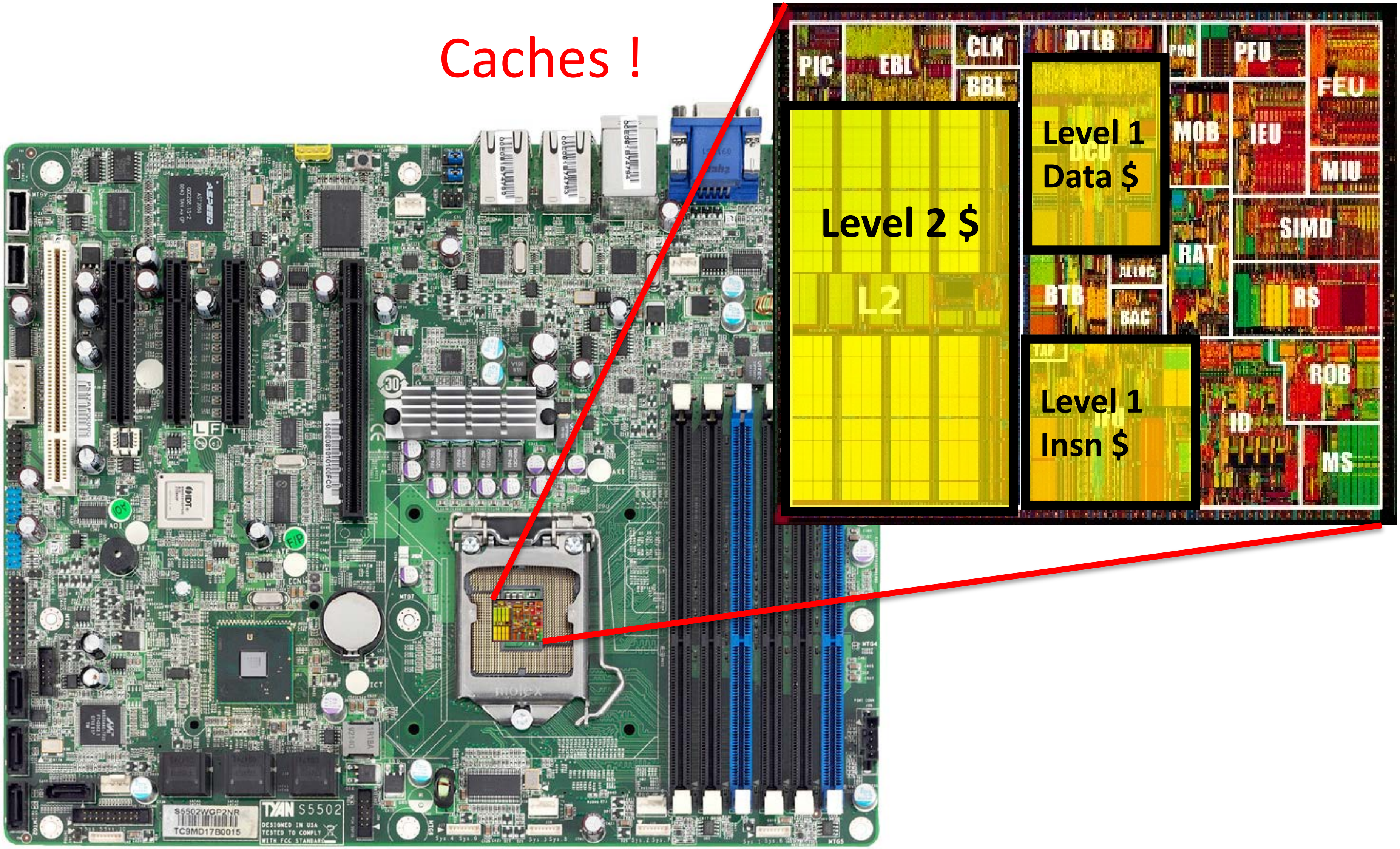
The Need for Speed

CPU Pipeline



What's the solution?

Caches !



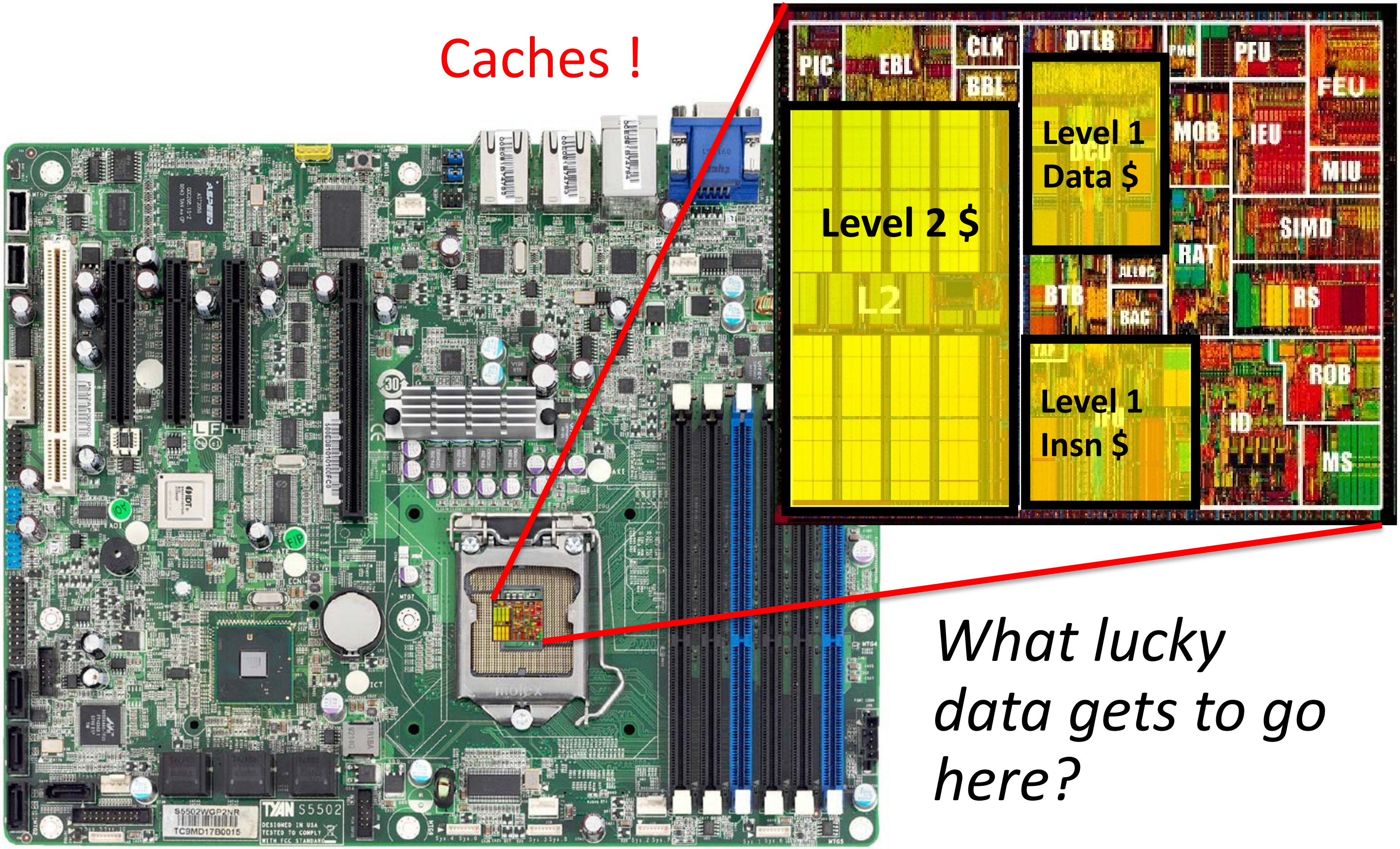
Intel Pentium 3, 1999

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?

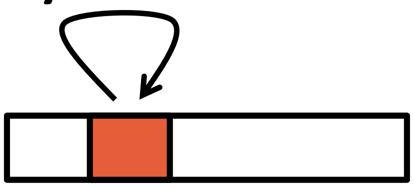
Intel Pentium 3, 1999

Locality Locality Locality

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

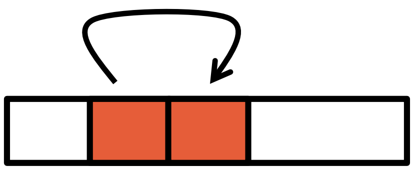
- the same thing again soon

  → Temporal Locality



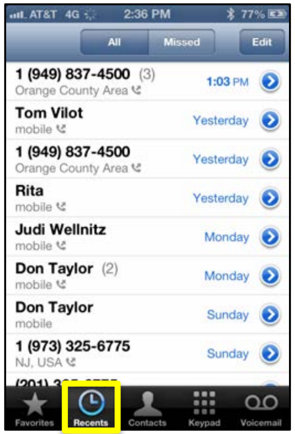
- something near that thing, soon

  → Spatial Locality



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

Your life is full of Locality



Last Called

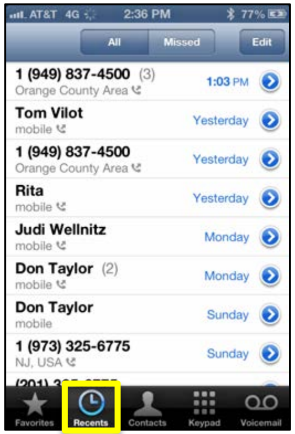
Speed Dial

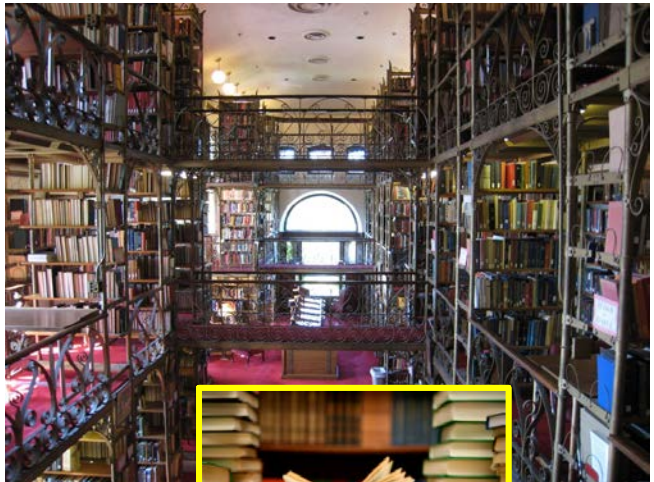
Favorites

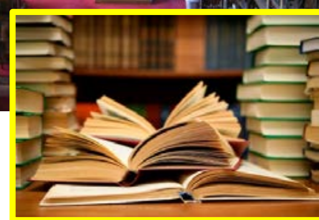
Contacts

Google/Facebook/email

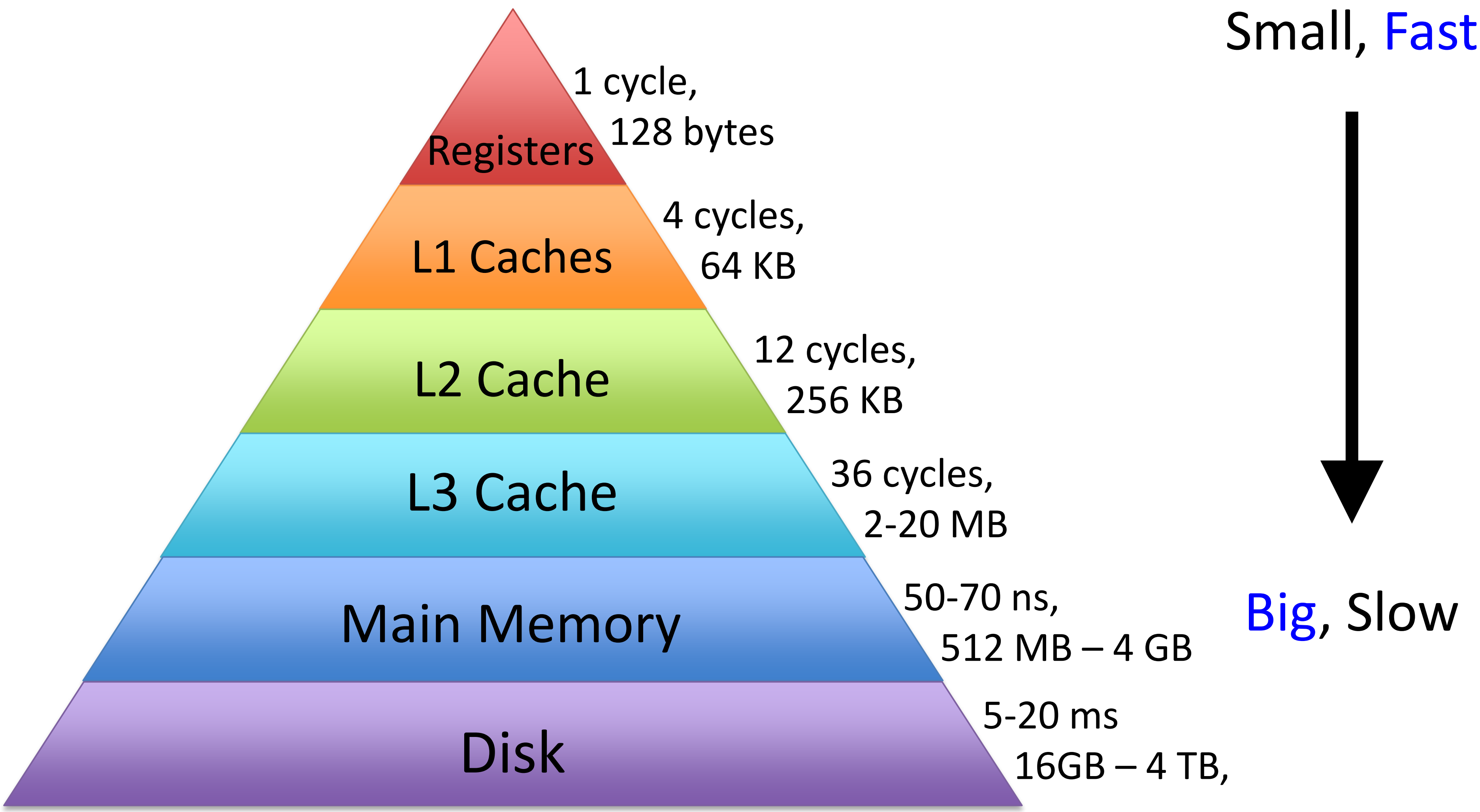
Your life is full of Locality







The Memory Hierarchy



Intel Haswell Processor, 2013

Some Terminology

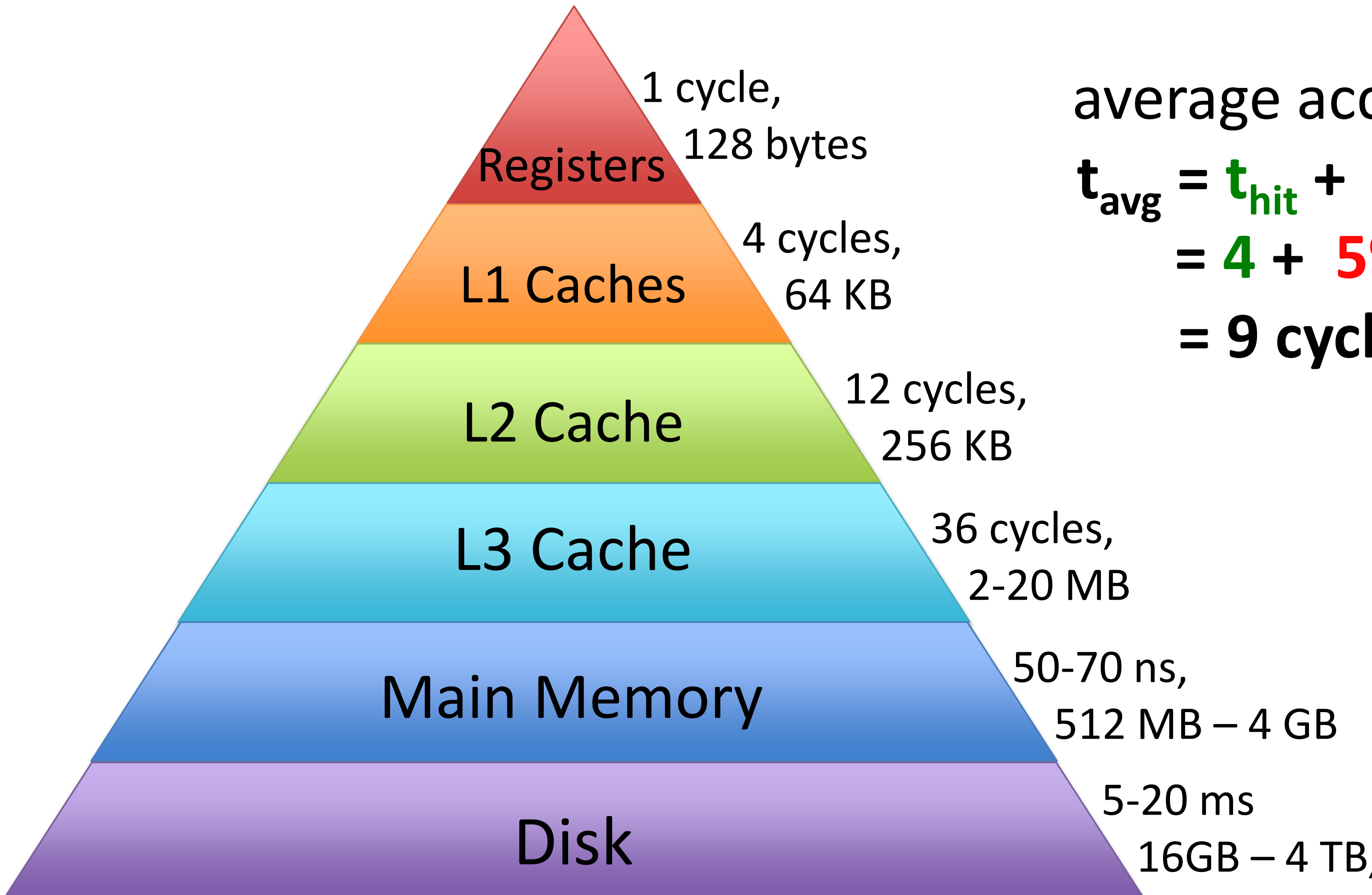
Cache hit

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

Cache miss

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

The Memory Hierarchy



average access time

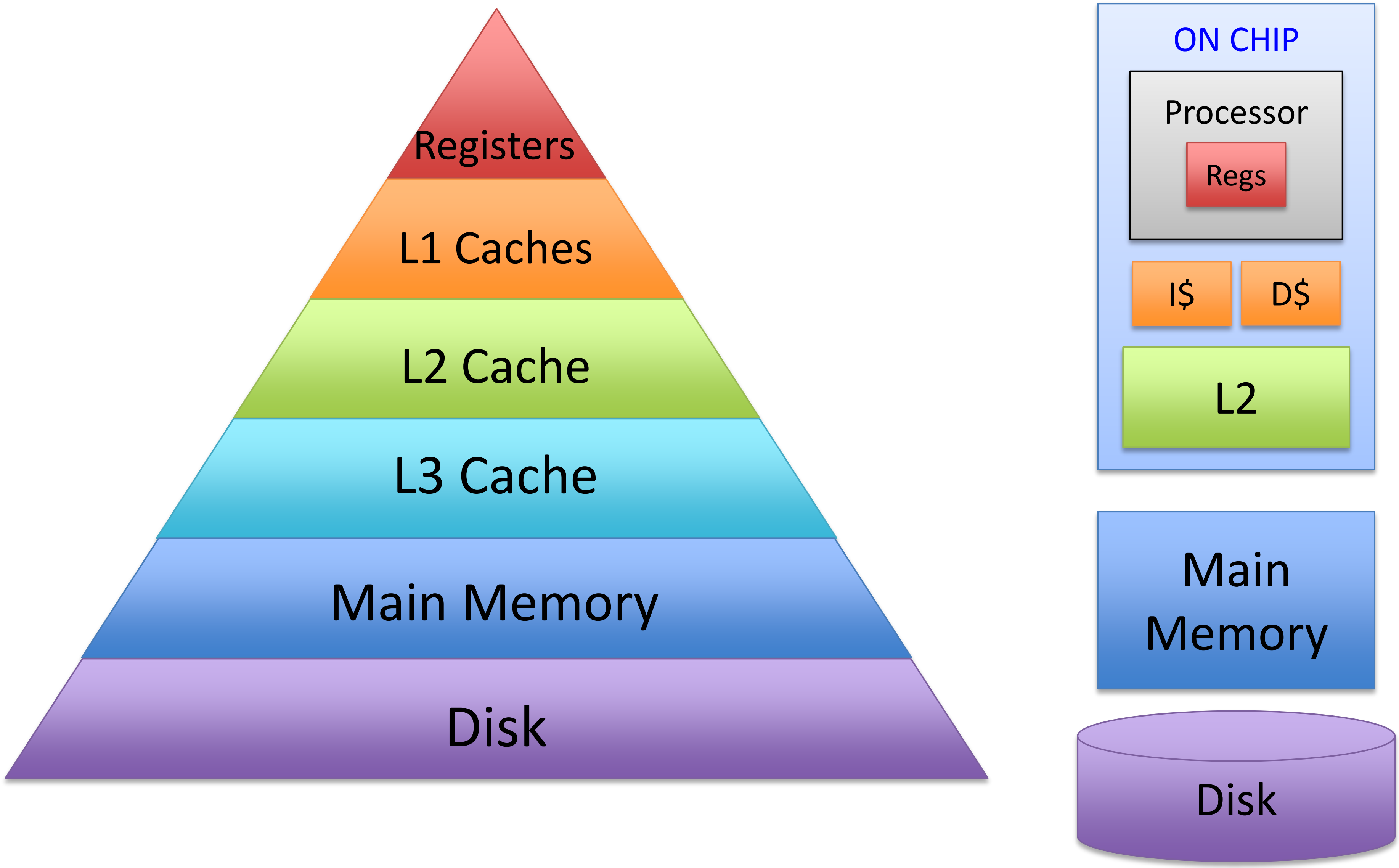
$$\mathbf{t_{avg} = t_{hit} + \%_{miss} * t_{miss}}$$

$$\mathbf{= 4 + 5\% \times 100}$$

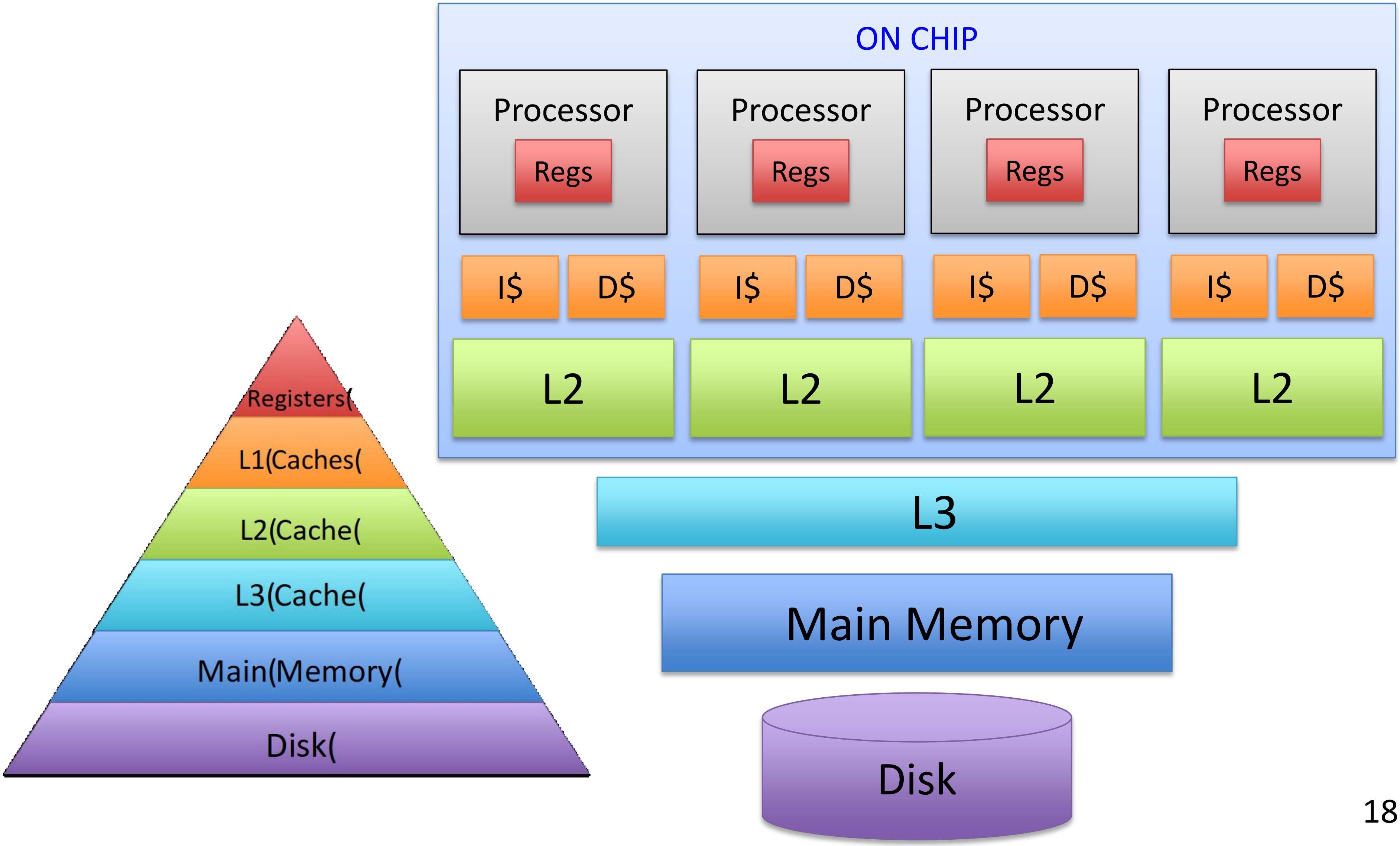
$$\mathbf{= 9\ cycles}$$

Intel Haswell Processor, 2013

Single Core Memory Hierarchy



Multi-Core Memory Hierarchy



Memory Hierarchy by the Numbers

CPU clock rates ~0.33ns – 2ns (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
$\boxed{1100}$	$\boxed{N}$
1101	O
1110	P
1111	Q

4-Byte, Direct Mapped Cache

index
XXXX

CACHE

index	data
00	A
01	B
10	C
11	D

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

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

Direct mapped:

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

Index with LSB:

- Supports spatial locality

Analogy to a Spice Rack

Spice Rack

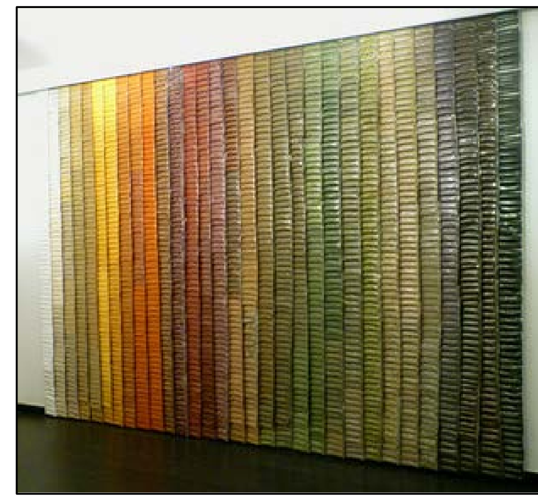
(Cache)

index spice

A
B
C
D
E
F
...
Z

Spice Wall

(Memory)

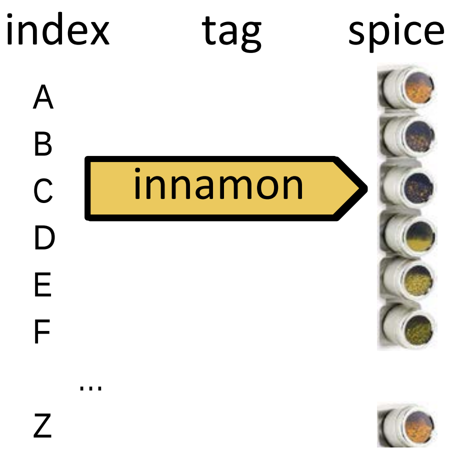


- Compared to your spice wall
  - Smaller
  - Faster
  - More costly (per oz.)

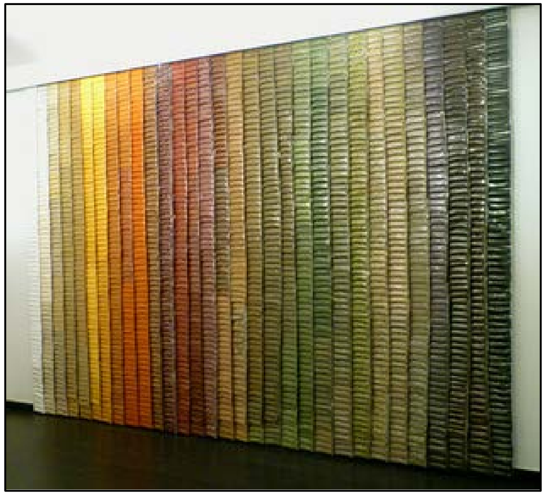
http://www.bedbathandbeyond.com

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

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

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

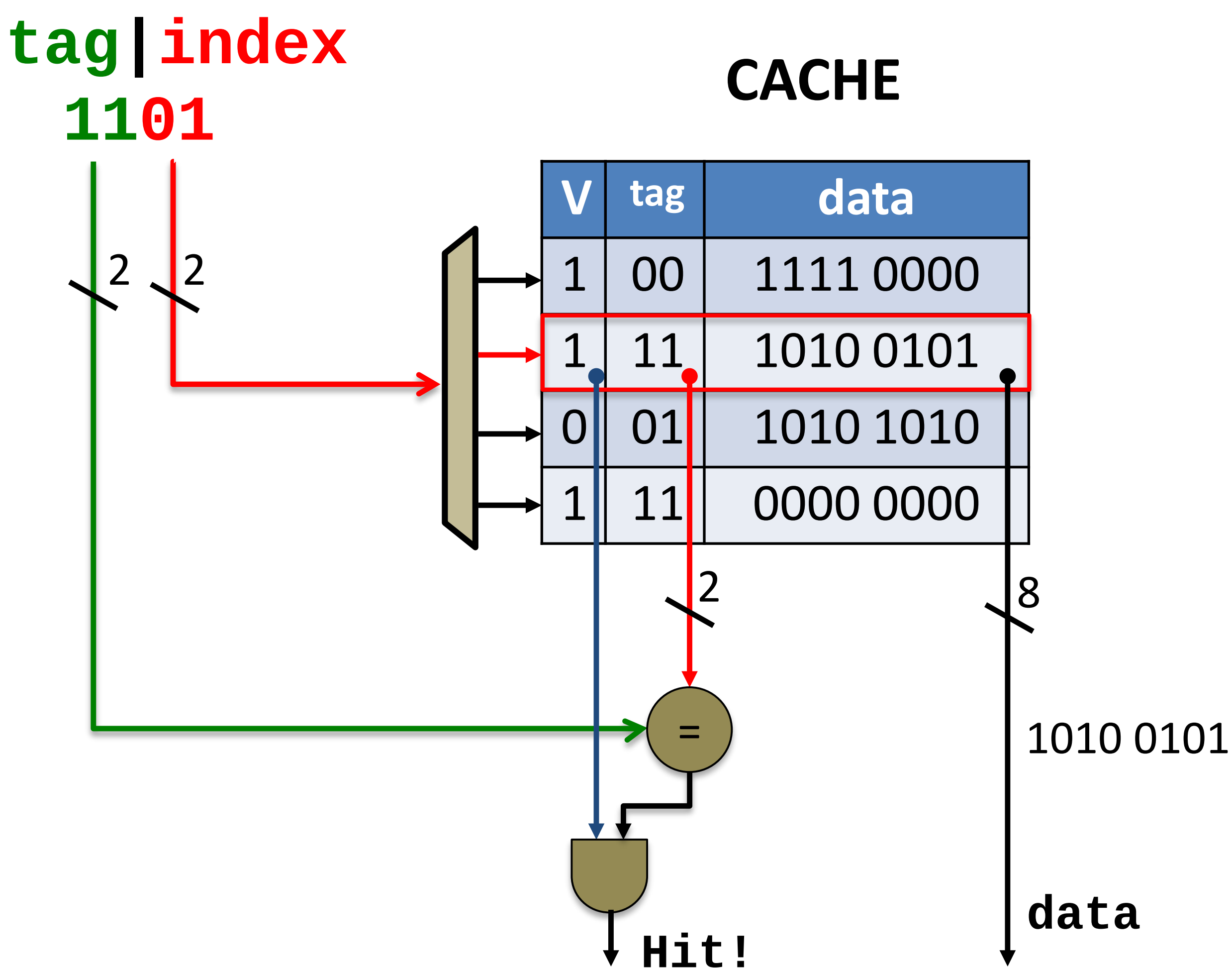
```
load    1100
...
load    1100
```

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

Block Diagram
4-entry, direct mapped Cache



Simulation #2:
4-byte, DM Cache

CACHE

index	V	tag	data
00	0	11	X
01	0	11	X
10	0	11	X
11	0	11	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
load  1101
load  0100
load  1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

Reducing Cold Misses by Increasing Block Size

Leveraging Spatial Locality

Increasing Block Size

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

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

- **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

tag | index | offset
XXXX

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	0	x	X | X
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
load    1101
load    0100
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

Removing Conflict Misses with Fully-Associative Caches

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
```

Lookup:

- ~~Index into $~~
- Check tags
- Check valid bits

LRU Pointer

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

Pros and Cons of Full Associativity

+ No more conflicts!

+ Excellent utilization!

But either:

Parallel Reads

– lots of reading!

Serial Reads

– lots of waiting

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

$= 4 + 5\% \times 100$

$= 9$ cycles

$= 6 + 3\% \times 100$

$= 9$ 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	V	tag	data
0	0	xx	E | F	0	xx	N | O
1	0	xx	C | D	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?

8 byte, 2-way set associative Cache

XXXX
tag|index|offset

CACHE

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

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

Lookup:

- Index into $
- Check tag
- Check valid bit

LRU Pointer

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

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

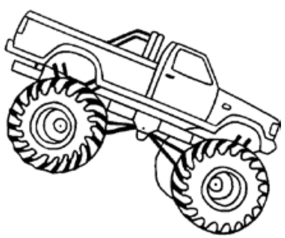
Misses: the Three C's

Cold (compulsory) Miss:

never seen this address before

Conflict Miss:

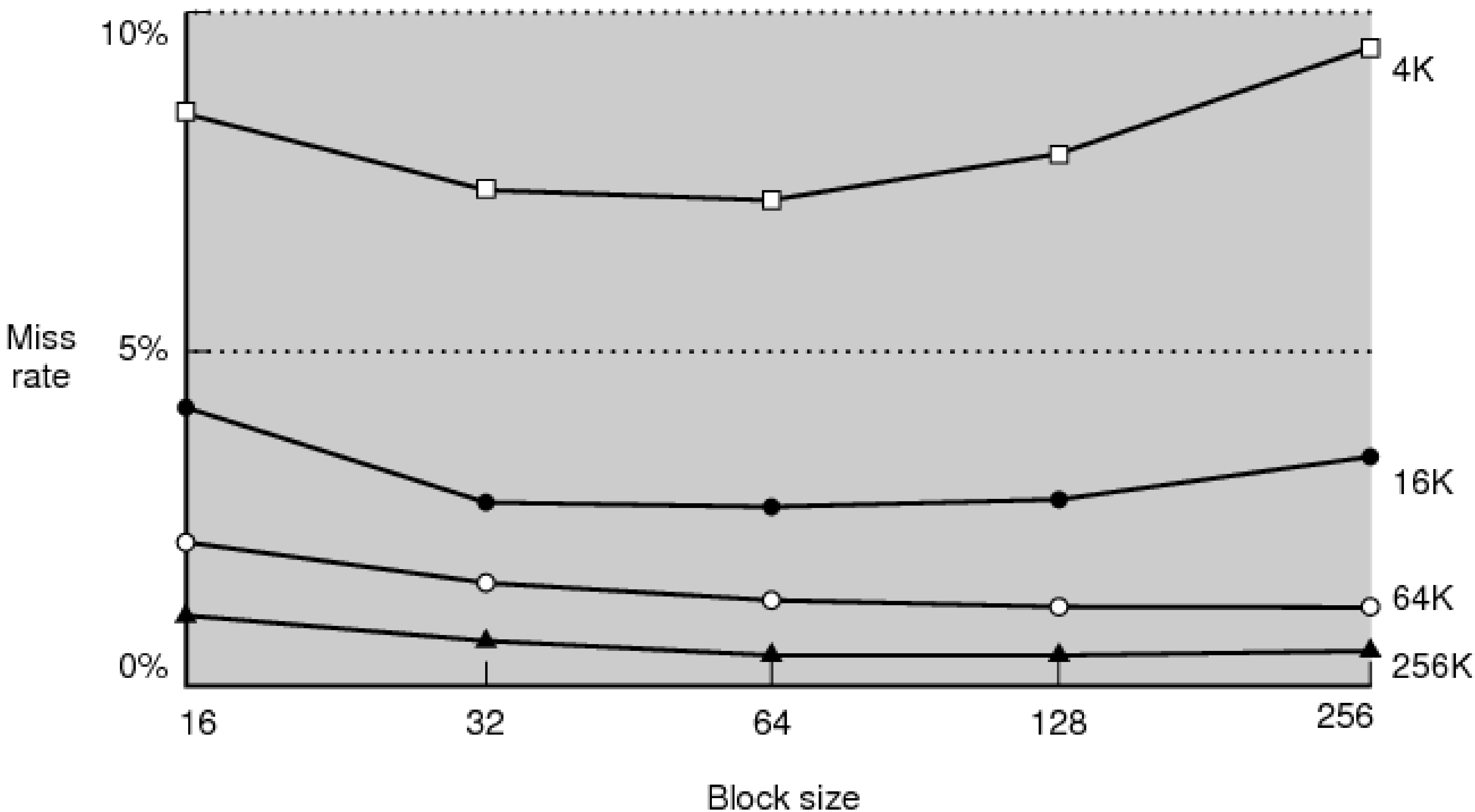
cache associativity is too low



Capacity Miss:

cache is too small

Miss Rate vs. Block Size

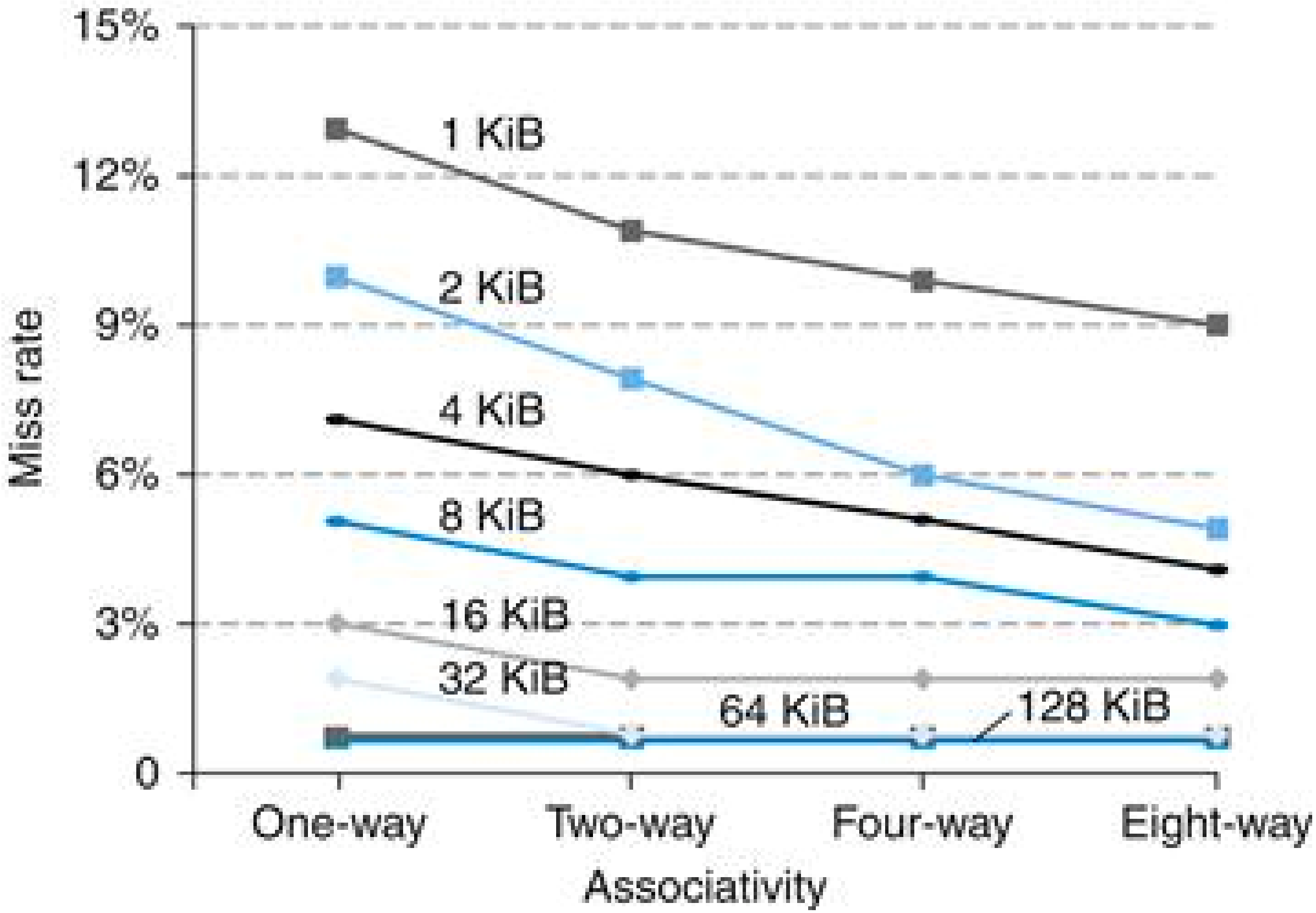


Block Size Tradeoffs

- 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



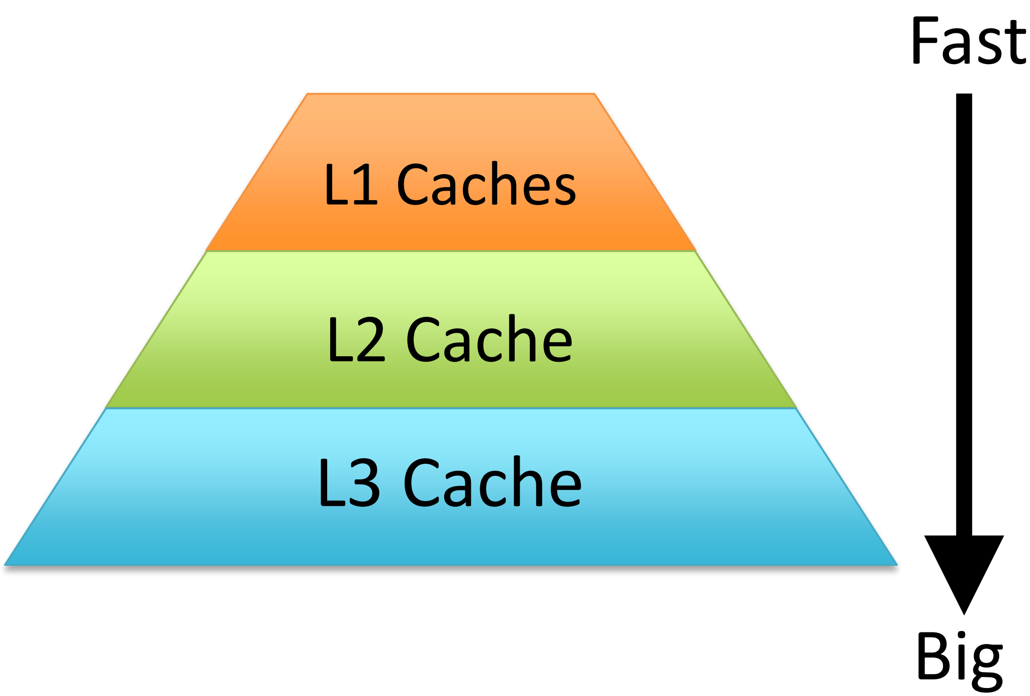
ABCs of Caches

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

+ Associativity:
  - ↓conflict misses ☺
  - ↑hit time ☹
+ Block Size:
  - ↓cold misses ☺
  - ↑conflict misses ☹
+ Capacity:
  - ↓capacity misses ☺
  - ↑hit time ☹

Which caches get what properties?

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



Design with speed in mind

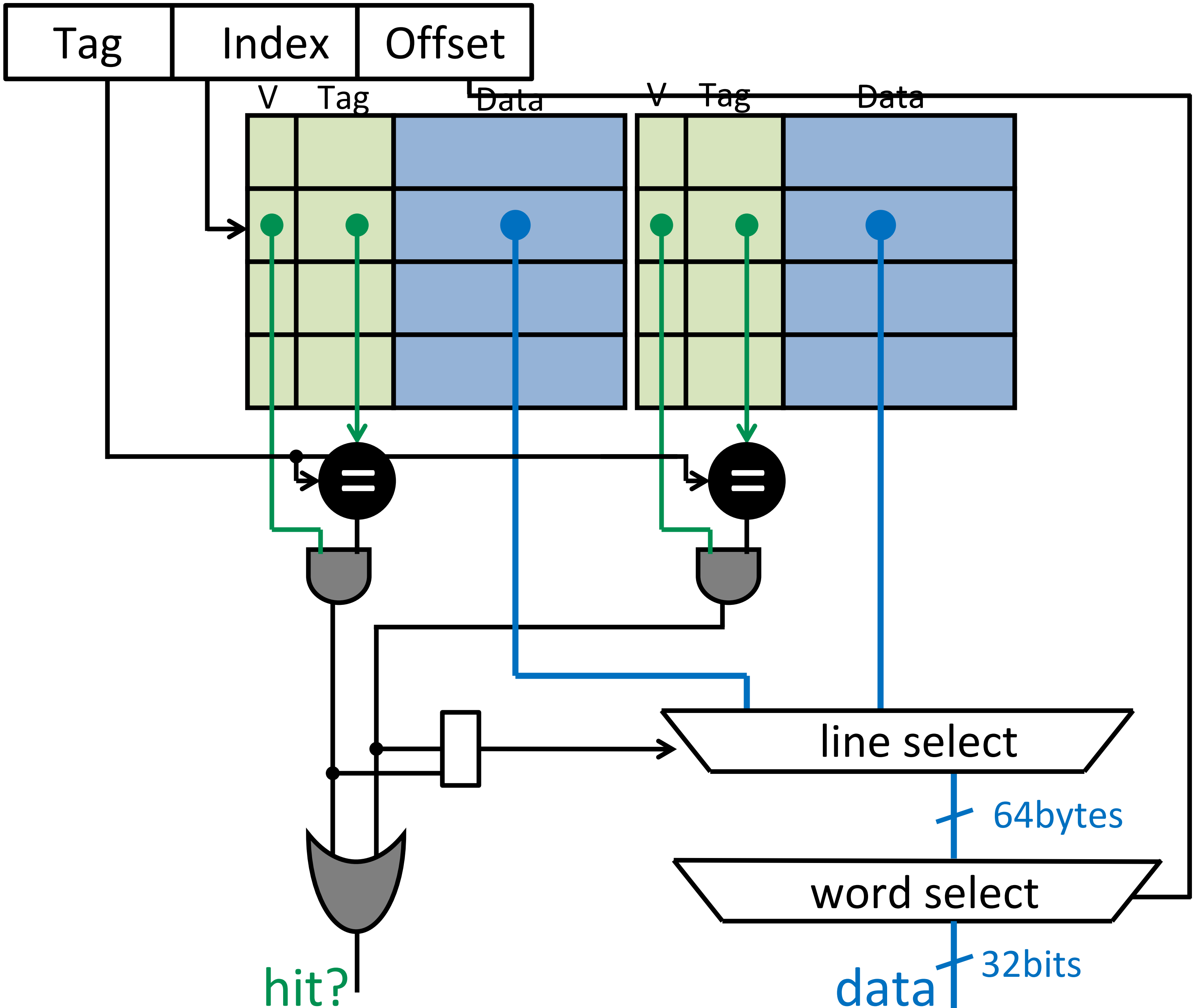
More Associative
Bigger Block Sizes
Larger Capacity

Design with miss rate in mind

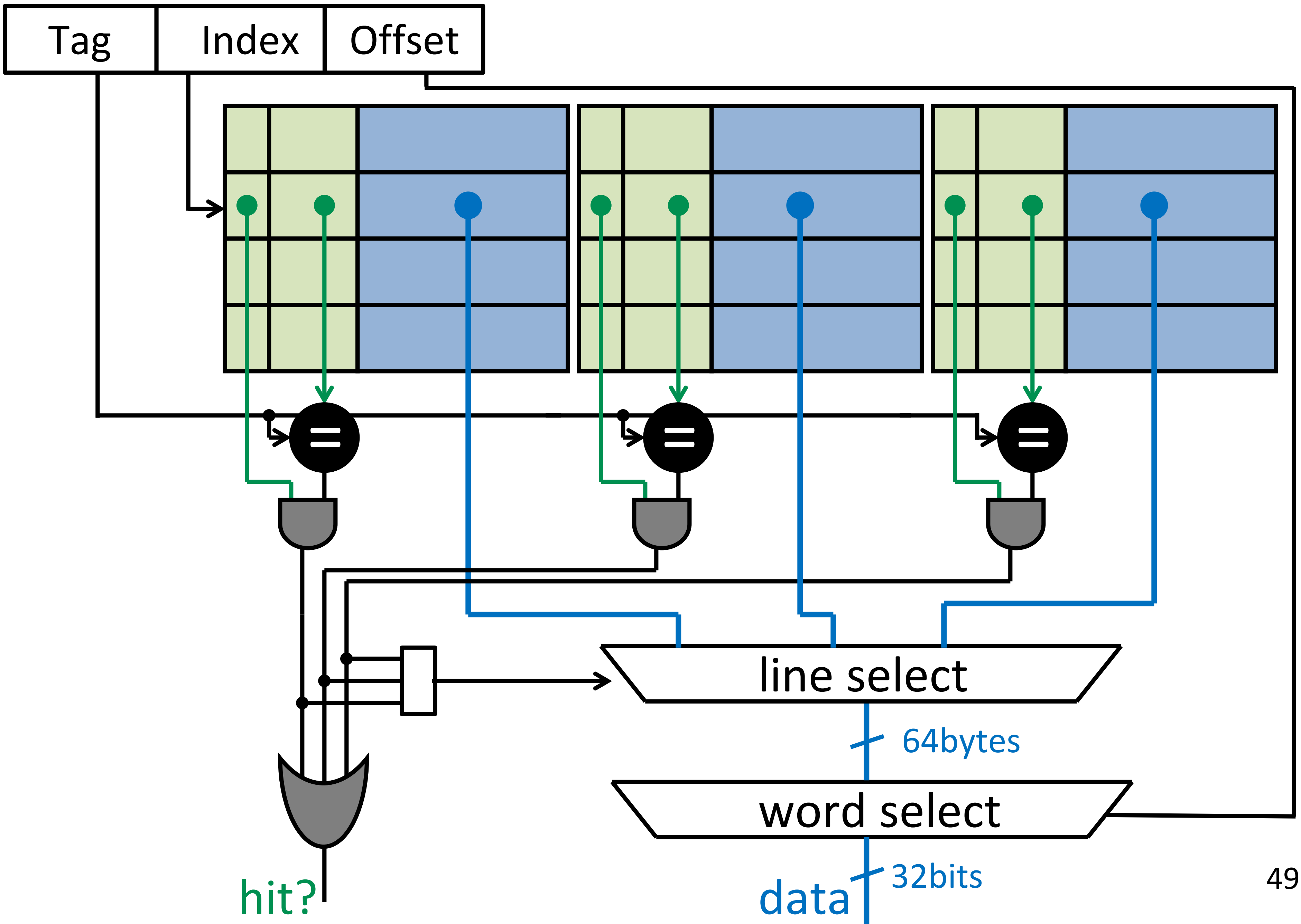
Roadmap

- 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

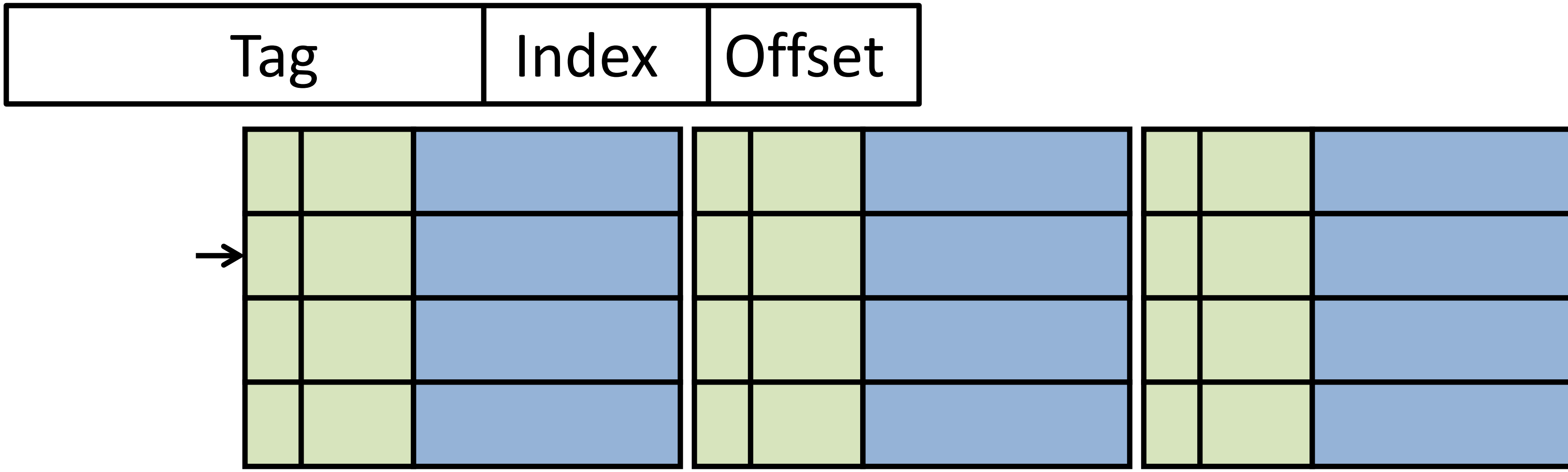
2-Way Set Associative Cache (Reading)



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

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

- **Parameters**
  - 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?**
  - $t_{missD\$}$ =
  - $t_{avgD\$}$ =
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ =
  - t_{avgL2} =
  - $t_{avgD\$}$ =

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, ∞)
- 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 is **working set** size cannot fit in cache.

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

What about Stores?

We want to write to the cache.

If the data is not in the cache?

Bring it in. (Write allocate policy)

Should we also update memory?

- Yes: write-through policy
- No: write-back policy

Write-Through Cache

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]

16 byte, byte-addressed memory
4 btye, fully-associative cache:
2-byte blocks, write-allocate
4 bit addresses:
3 bit tag, 1 bit offset

Cache

lru	V	tag	data
1	0		
0	0		

Register File

$0	
$1	
$2	
$3	

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

Cache

lru	V	tag	data
1	0		
0	0		

Register File

$0	
$1	
$2	
$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

Misses: 0

Hits: 0

Reads: 0

Writes: 0

Summary: Write Through

Write-through policy with write allocate

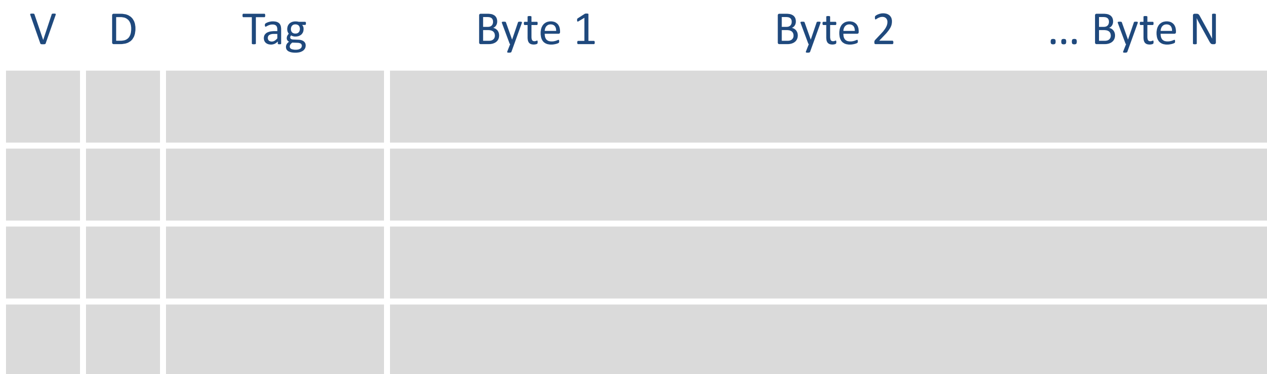
- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory

Next Goal: Write-Through vs. Write-Back

What if we DON'T to write 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 btye, 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]

Cache

lru	V	d	tag	data
1	0			
0	0			

Register File

$0	
$1	
$2	
$3	

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

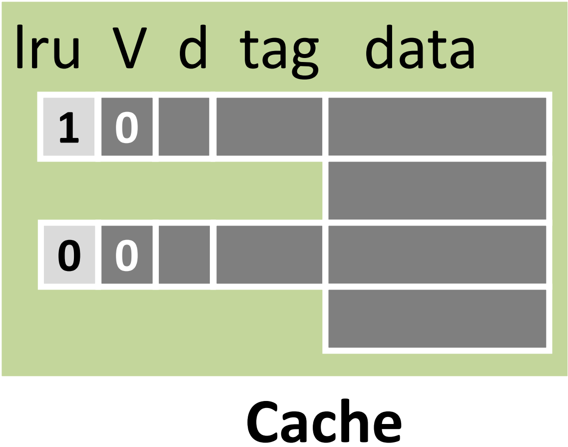
Memory

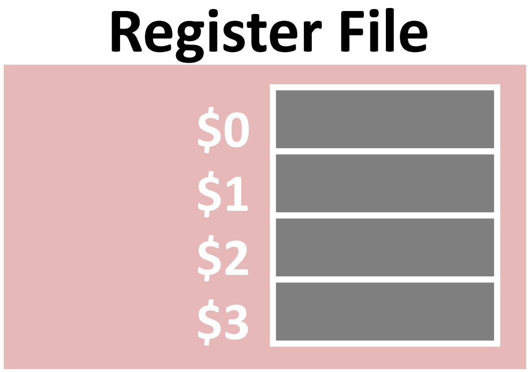
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-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]





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

How Many Memory References?

Write-back performance

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

Write-back vs. Write-through Example

Assume: large associative cache, 16-byte lines

N 4-byte words

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

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

So is write back just better?

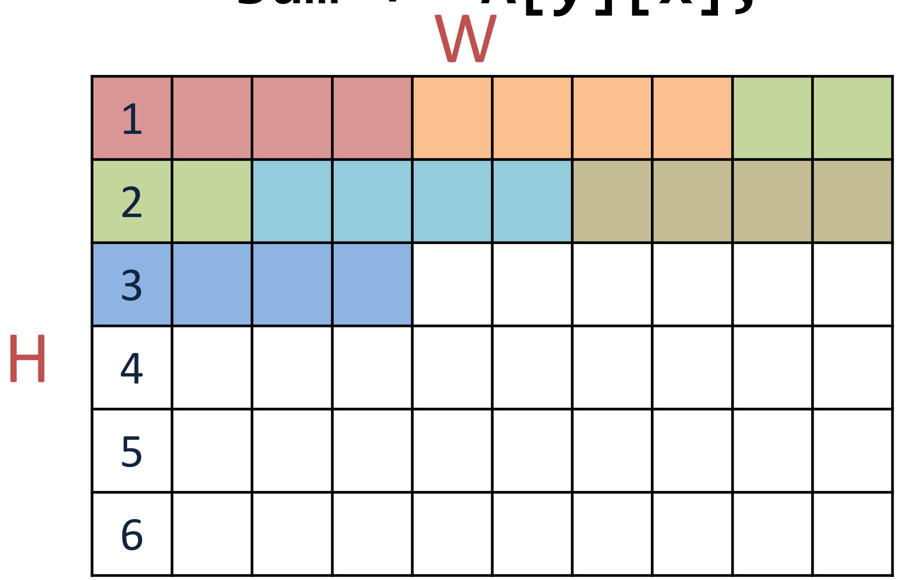
Short Answer: Yes (fewer writes is a good thing)

Long Answer: It’s complicated.

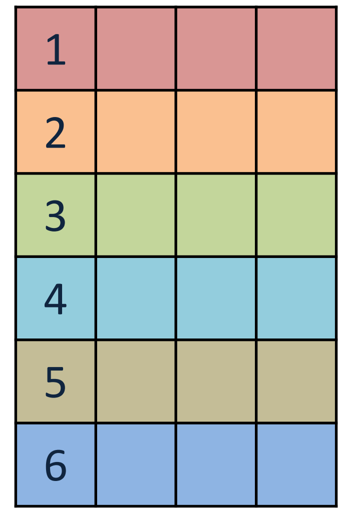
- Evictions require entire line be written back to memory (vs. just the data that was written)
- Write-back can lead to incoherent caches on multi-core processors (later lecture)

Cache Conscious Programming

```
// H = 6, W = 10
int A[H][W];
for(x=0; x < W; x++)
   for(y=0; y < H; y++)
      sum += A[y][x];
```



YOUR MIND



CACHE

MEMORY

Every access a cache miss!

(unless *entire* matrix fits in cache)

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.

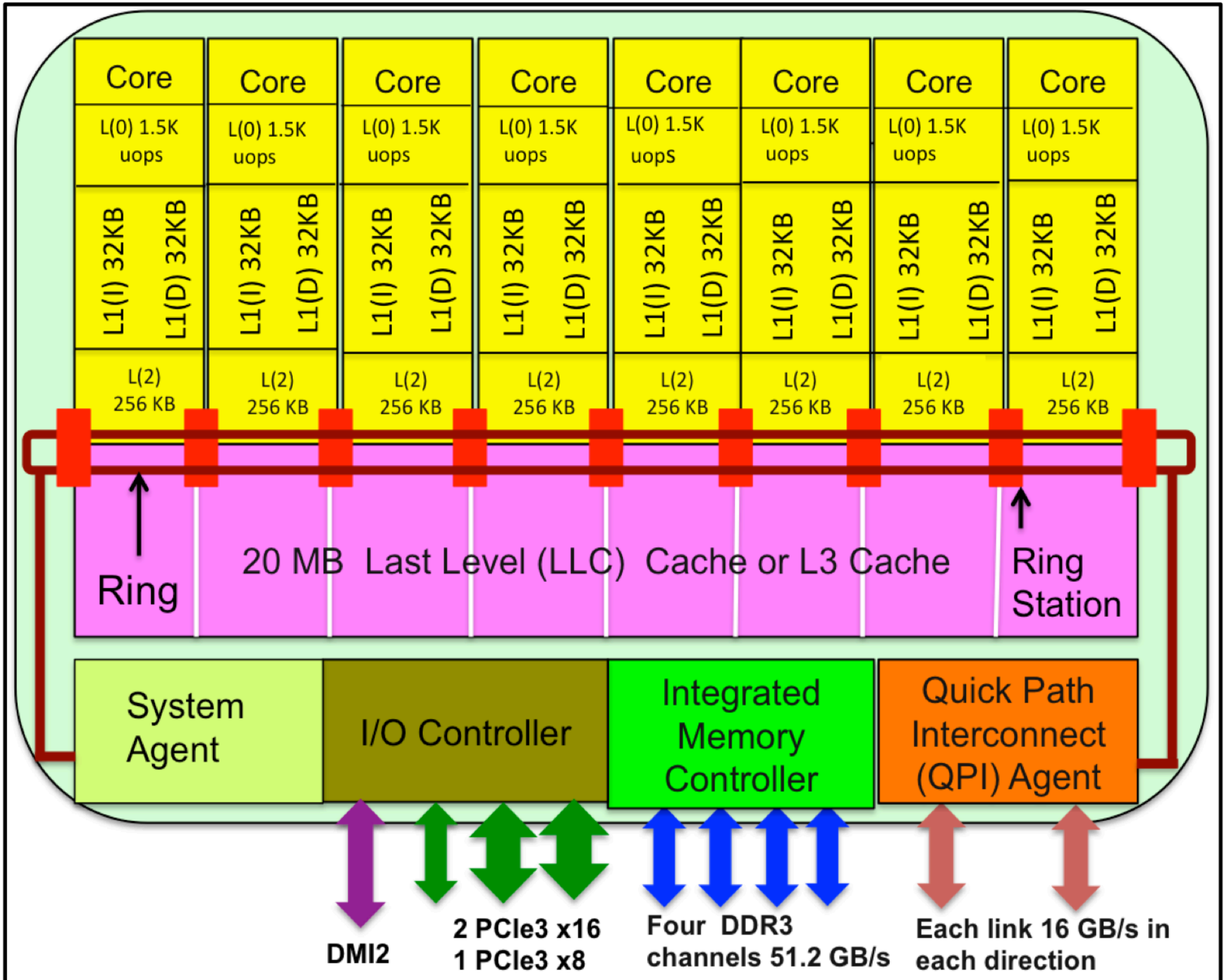


Figure 1. Schematic diagram of a Sandy Bridge processor.

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

Have a great Spring Break!!