Caches (1)

CS 3410: Computer System Organization and Programming

Spring 2025

Today's Goals

- What are caches?
- DRAM vs. SRAM
- Principle of Locality
- Memory Hierarchy Revisited
- Direct Mapped Caches

Our Final Exam is on
Saturday 5/17/2025 at 2:00pm

Programs Are Constantly Accessing Memory

C Code

```c
int n = 100;
int main (int argc, char* argv[]) {
  int i, m = n, sum = 0;
  for (i = 1; i ≤ m; i++) {
    sum += i;
  }
}
```

Load/Store Architectures:

- Read data from memory into registers
- Manipulate it
- Store it back to memory

RISC-V Assembly Code

```
main:
  addi   sp,sp,-48
  sd     ra,40(sp)
  sd     s0,32(sp)
  addi   s0,sp,48
  mv     a5,a0
  sd     a1,-48(s0)
  sw     a5,-36(s0)
  lui    a5,0x12
  lw     a5,16(a5)
  sw     a5,-28(s0)
  sw     zero,-24(s0)
  li     a5,1
  sw     a5,-20(s0)
  j      check
loop:
  lw     a5,-24(s0)
  mv     a4,a5
  lw     a5,-20(s0)
  addw   a5,a4,a5
  sw     a5,-24(s0)
  lw     a5,-20(s0)
  addiw  a5,a5,1
  sw     a5,-20(s0)

check:
  lw     a5,-20(s0)
  mv     a4,a5
  lw     a5,-28(s0)
  sext.w a4,a4
  sext.w a5,a5
  bge    a5,a4,loop
  li     a5,0
  mv     a0,a5
  ld     ra,40(sp)
  ld     s0,32(sp)
  addi   sp,sp,48
  ret
```

(rv gcc –std=c23 sum.c –o sum)

Programs Are Constantly Accessing Memory

Half of the instructions transfer memory!

C Code

```
int n = 100;
int main (int argc, char* argv[]) {
  int i, m = n, sum = 0;
  for (i = 1; i ≤ m; i++) {
    sum += i;
  }
}
```

Load/Store Architectures:

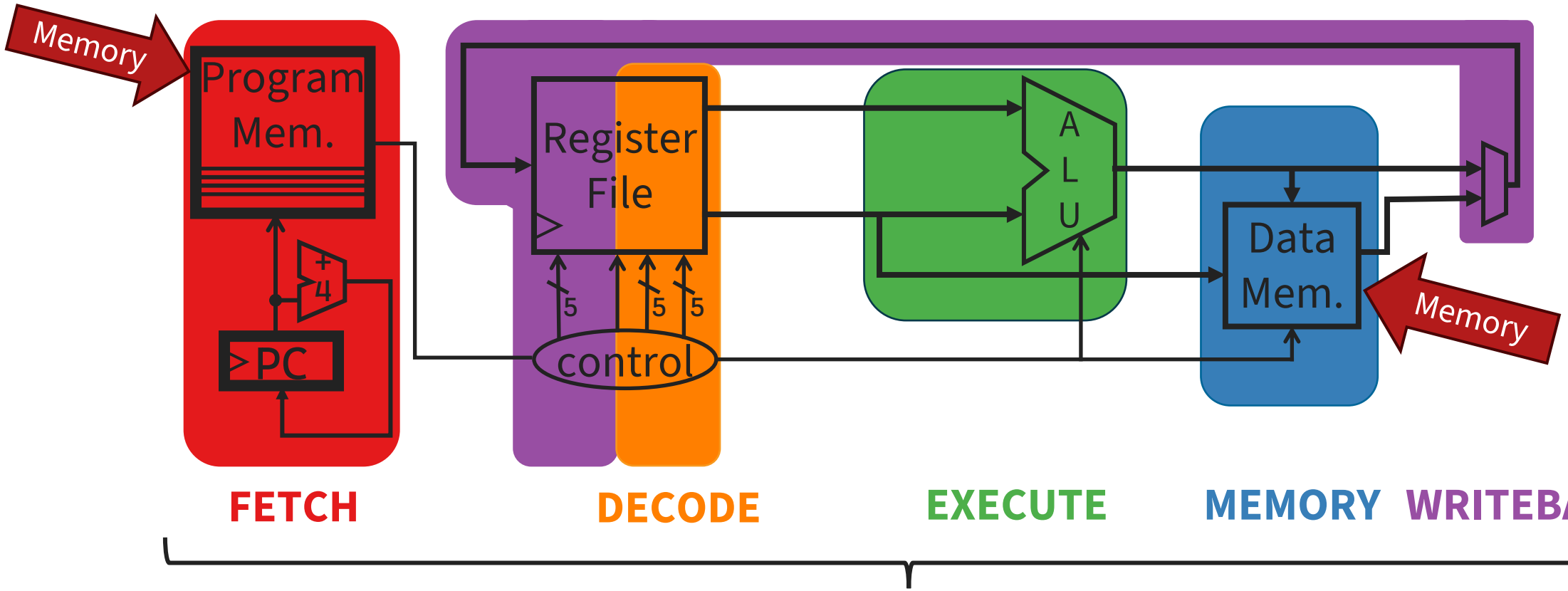
- Read data from memory into registers
- Manipulate it
- Store it back to memory

RISC-V Assembly Code

```
main:
  addi   sp,sp,-48
  sd     ra,40(sp)
  sd     s0,32(sp)
  addi   s0,sp,48
  mv     a5,a0
  sd     a1,-48(s0)
  sw     a5,-36(s0)
  lui    a5,0x12
  lw     a5,16(a5)
  sw     a5,-28(s0)
  sw     zero,-24(s0)
  li     a5,1
  sw     a5,-20(s0)
  j      check
loop:
  lw     a5,-24(s0)
  mv     a4,a5
  lw     a5,-20(s0)
  addw   a5,a4,a5
  sw     a5,-24(s0)
  lw     a5,-20(s0)
  addiw  a5,a5,1
  sw     a5,-20(s0)
check:
  lw     a5,-20(s0)
  mv     a4,a5
  lw     a5,-28(s0)
  sext.w a4,a4
  sext.w a5,a5
  bge    a5,a4,loop
  li     a5,0
  mv     a0,a5
  ld     ra,40(sp)
  ld     s0,32(sp)
  addi   sp,sp,48
  ret
```

(rv gcc –std=c23 sum.c –o sum)

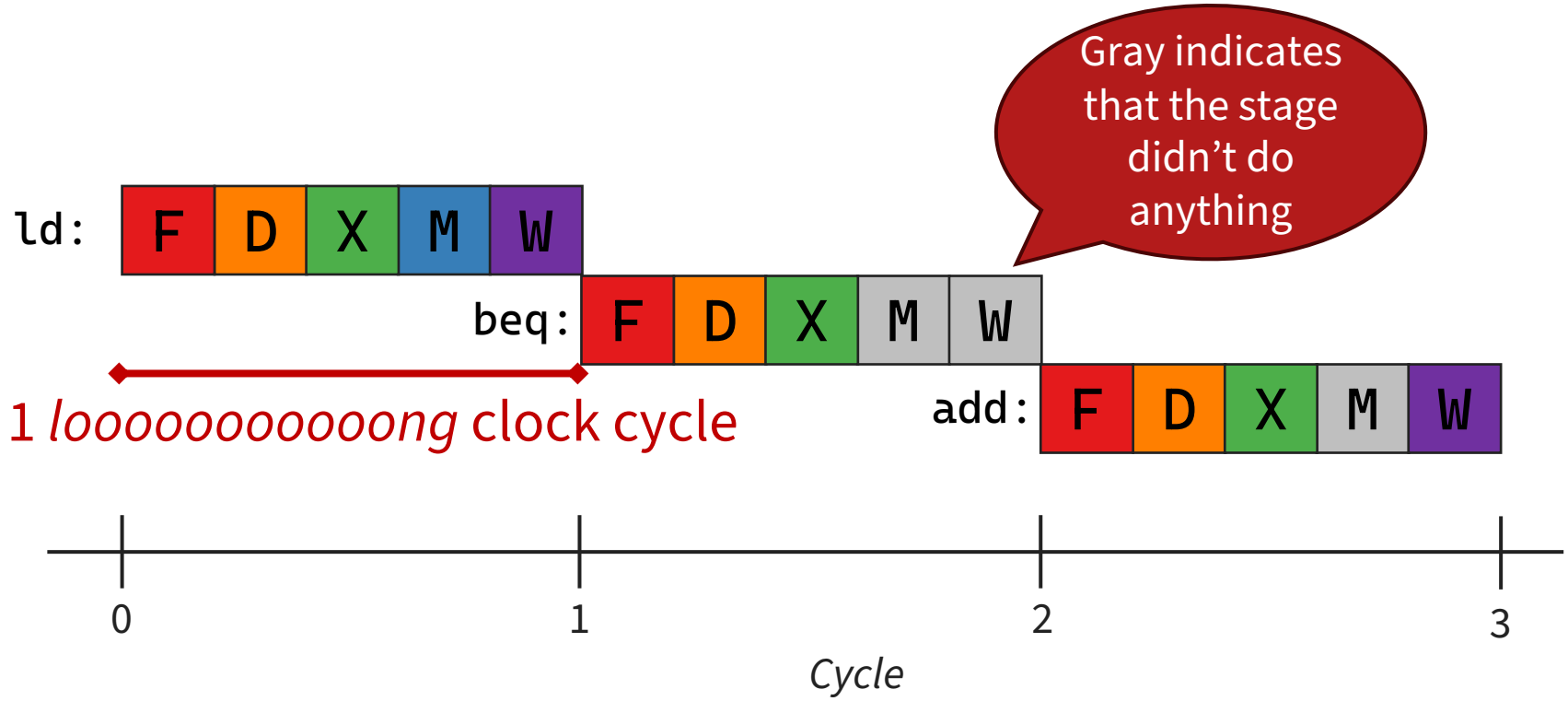
Single-Cycle RISC-V Datapath



Clock frequency must be slow enough for the very **slowest** instruction to complete in **1 cycle**

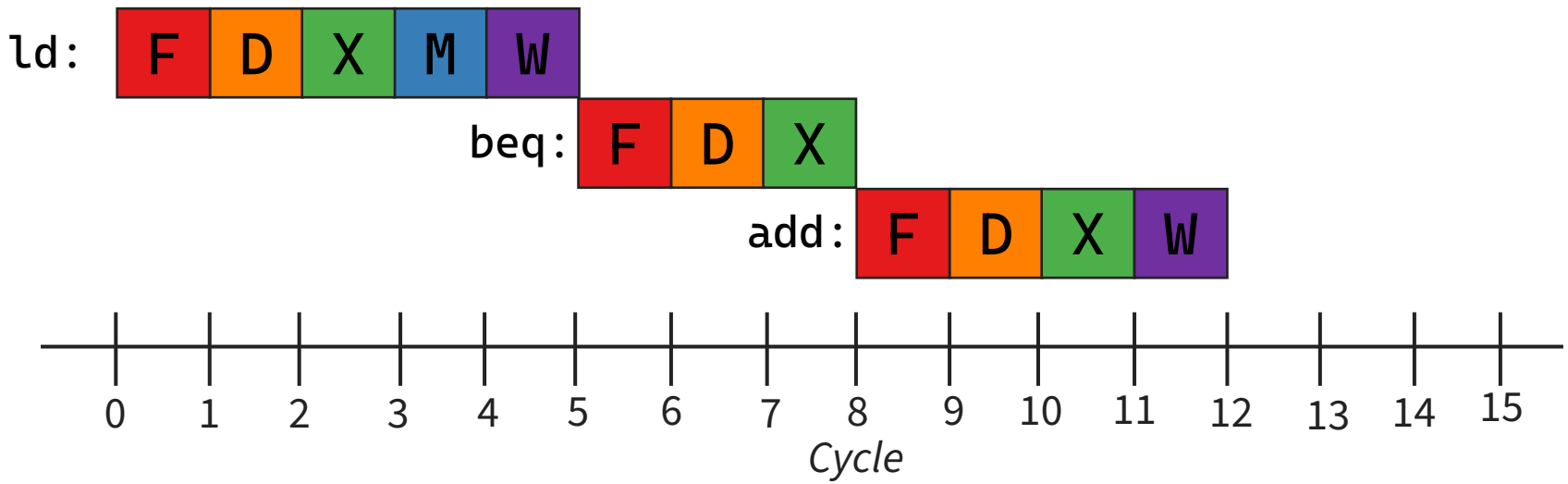
1 Stage ≠ 1 Cycle

Single-cycle



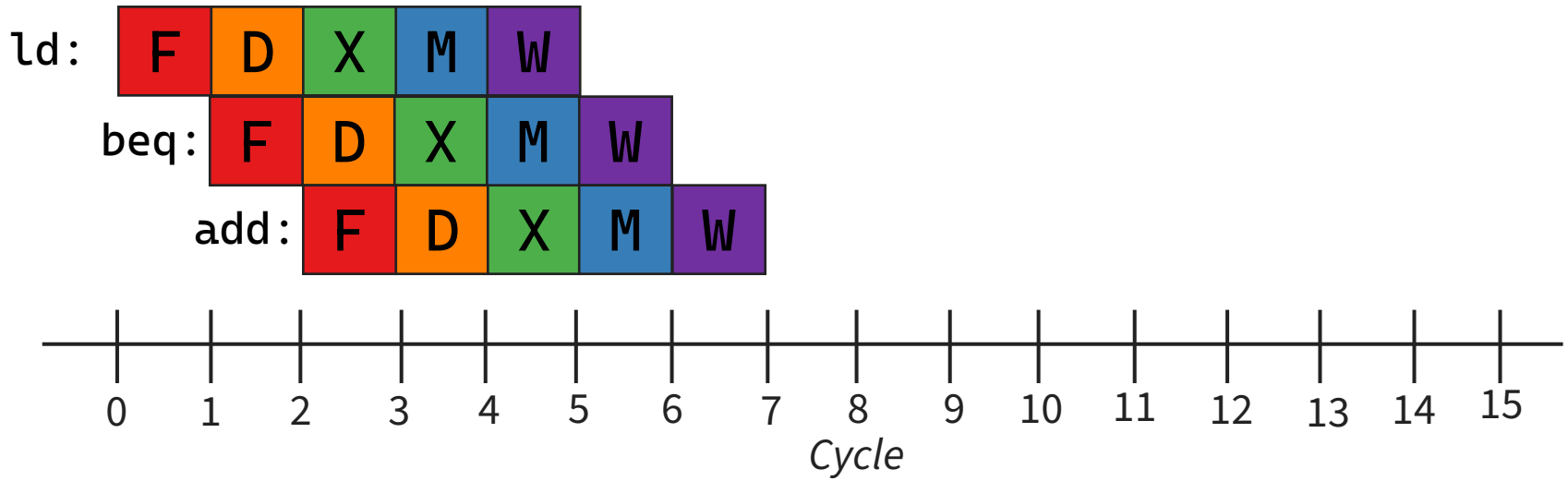
1 Stage ≠ 1 Cycle

Multi-cycle

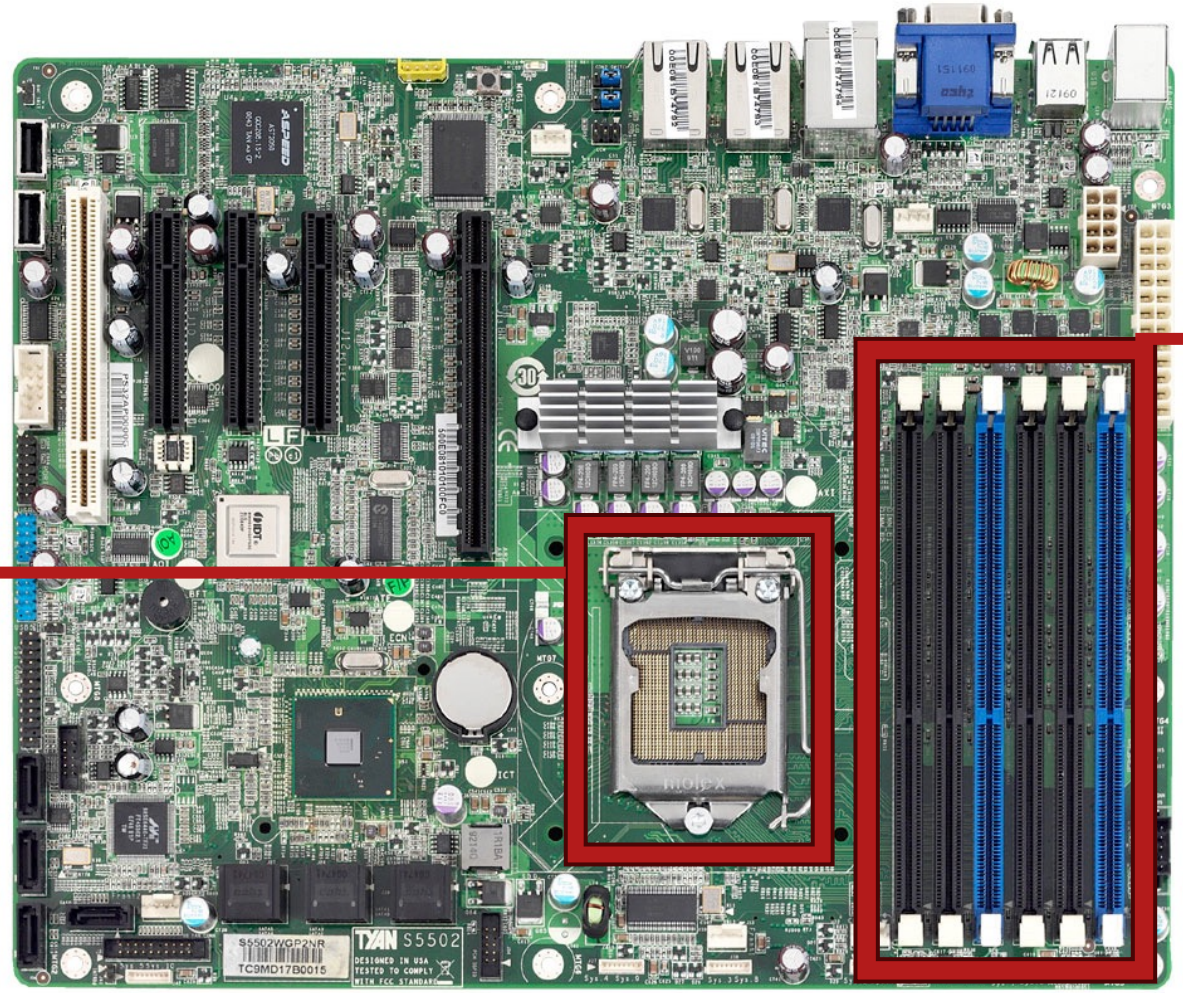


1 Stage ≠ 1 Cycle

Pipelined



Memory is far away from the CPU



CPU

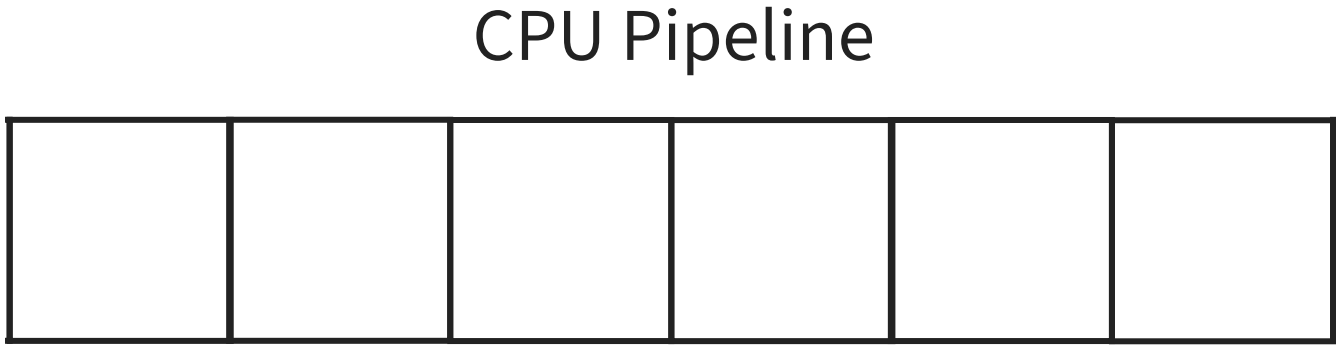
- Small
- Expensive
+ Fast
+ Close

Main Memory

+ Big
+ Cheap
- Slow
- Far away

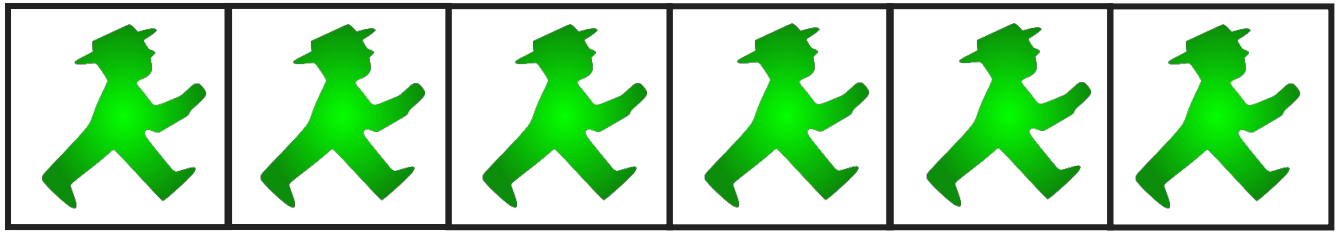
SandyBridge Motherboard, 2011 http://news.softpedia.com

The Need for Speed



The Need for Speed

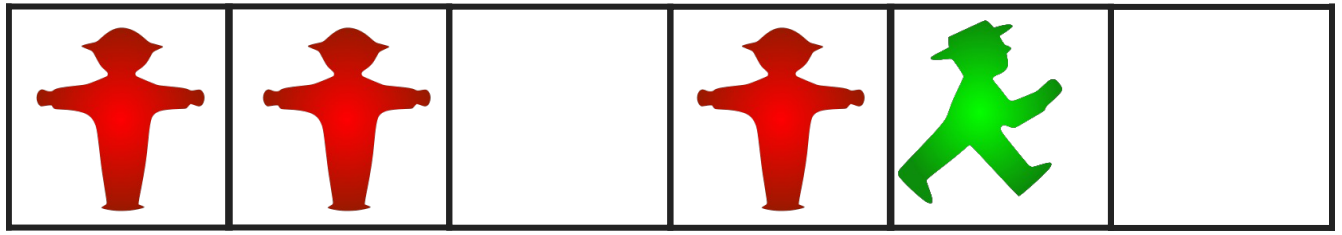
CPU Pipeline



- Many stages take ~1 cycle…
  - Integer addition
  - Shifting
- But not all do:
  - Integer multiplication & division (3-15 cycles)
  - **Accessing memory (Hundreds of cycles)**

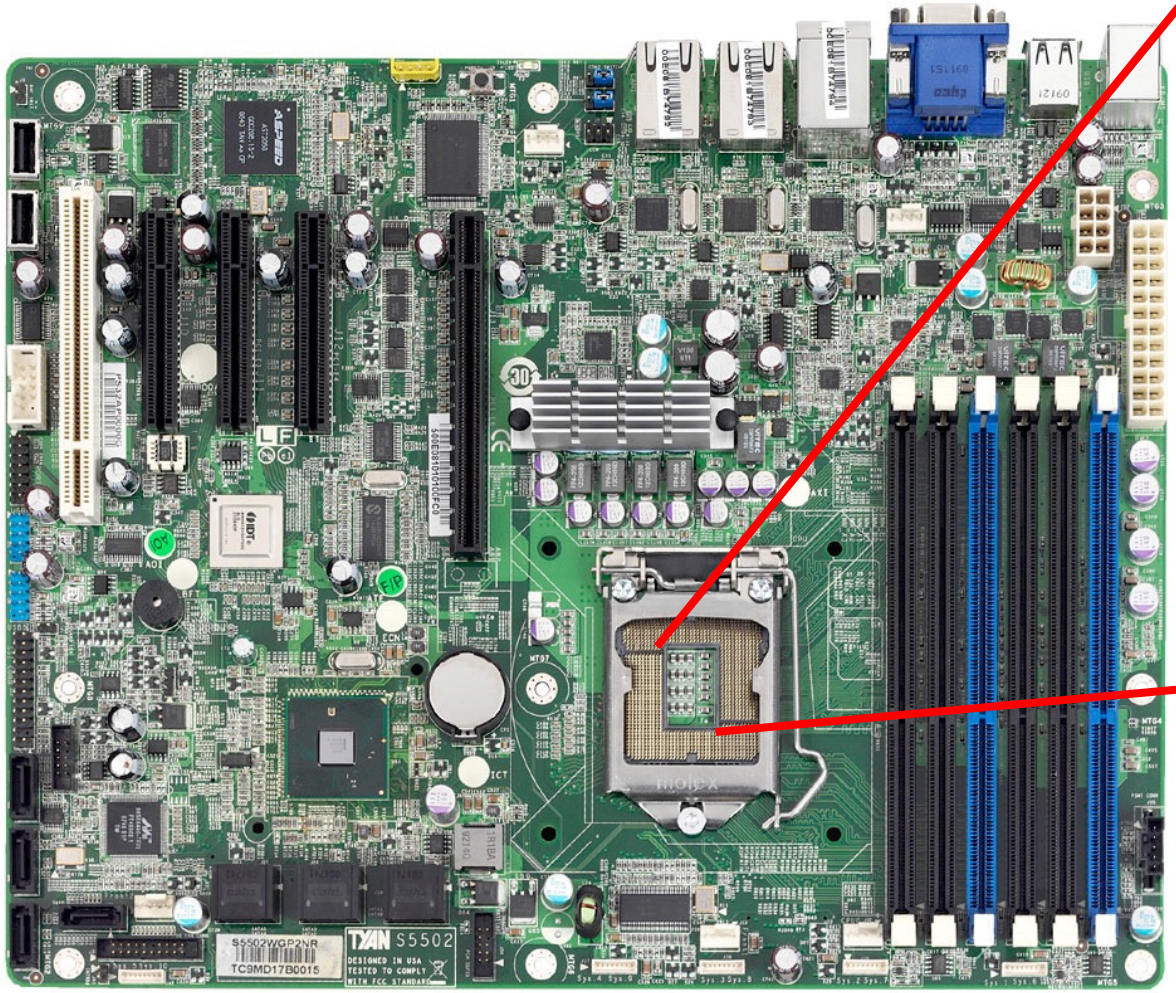
The Need for Speed

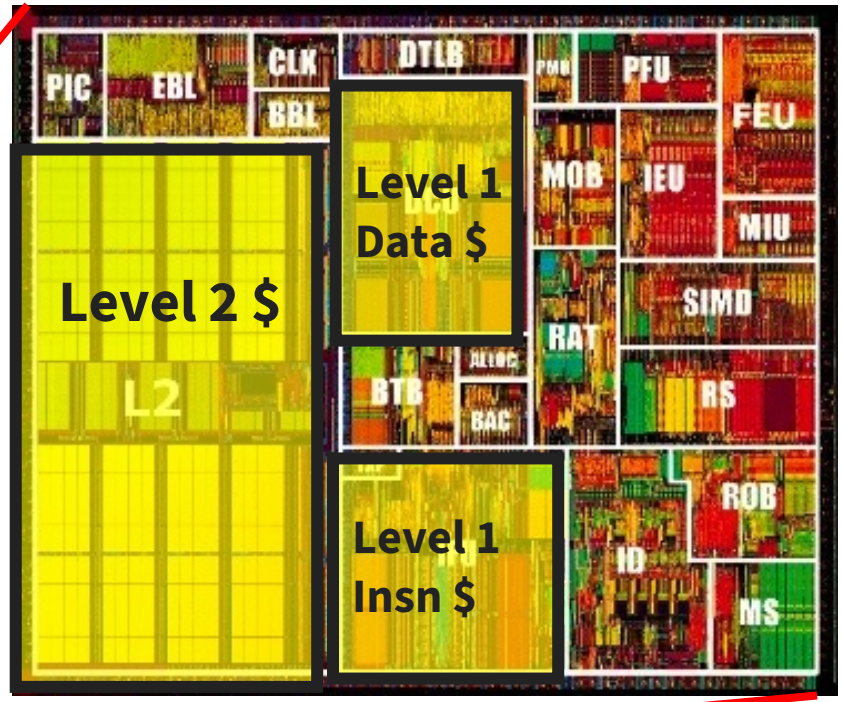
CPU Pipeline



- Many stages take ~1 cycle…
  - Integer addition
  - Shifting
- But not all do:
  - Integer multiplication & division (3-15 cycles)
  - **Accessing memory (Hundreds of cycles)**

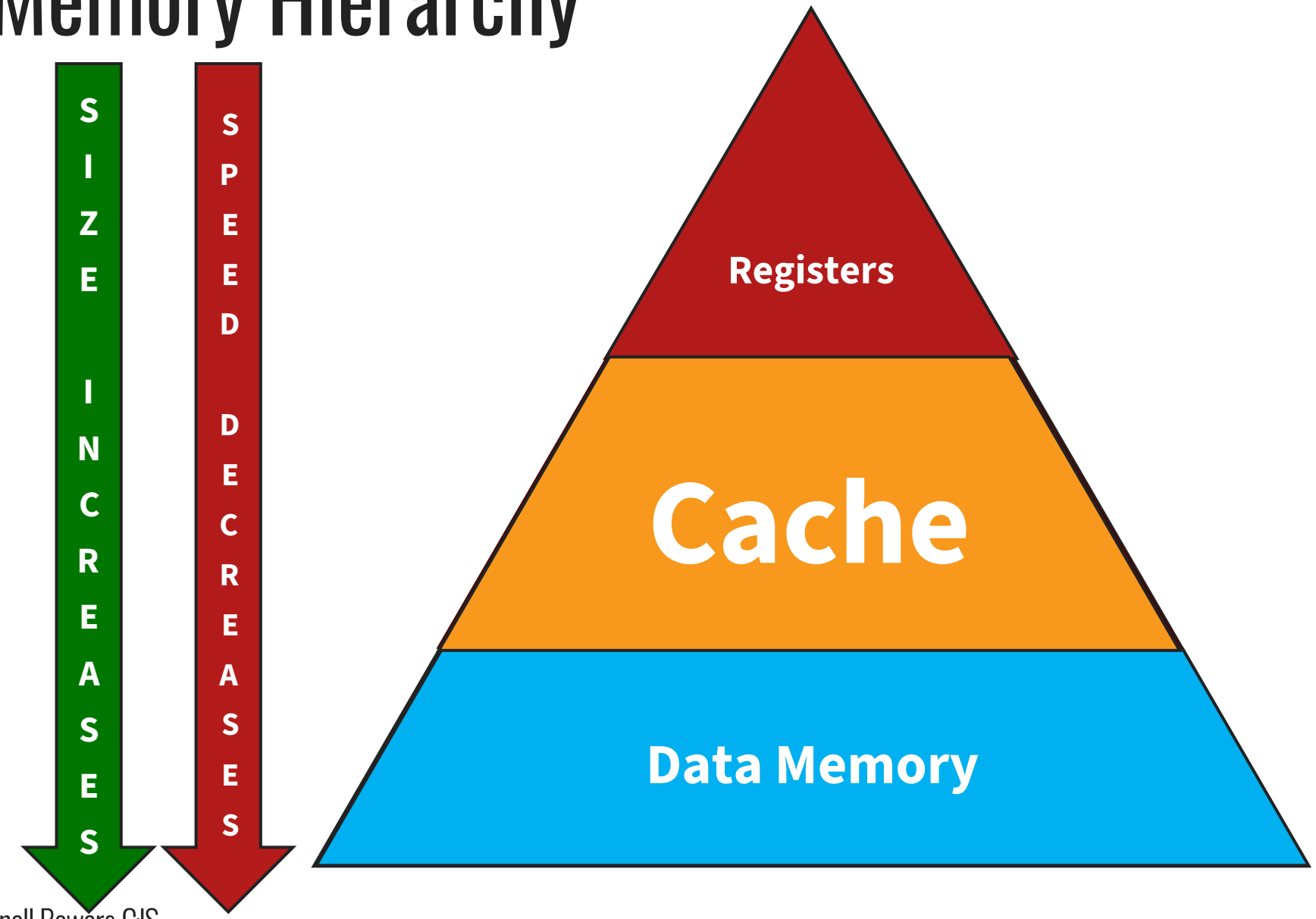
Caches!





SandyBridge Motherboard, 2011 http://news.softpedia.com
Intel Pentium 3, 1999

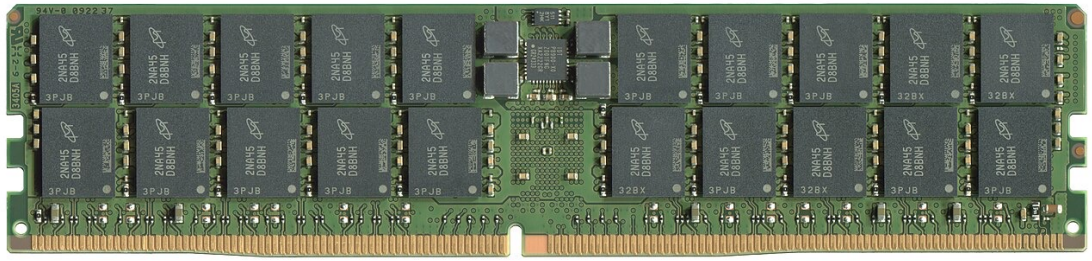
Memory Hierarchy



DRAM vs. SRAM

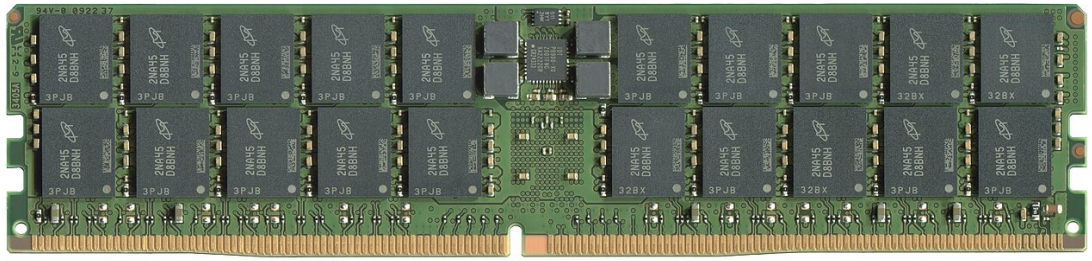
Dynamic Random-Access Memory (DRAM)

- **Main Memory** is DRAM
- **Latency**:
  - First Word: 10 ns (~30-40 cycles)
  - Successive: 0.5ns – 1ns
- **Cost:** $3/GiB
  - 1 Gi**bi**byte (GiB) = $2^{\mathbf{30}m}$ bytes
  - 1 Gi**ga**byte (GB) = $10^{\mathbf{9}v}$ bytes
- **Size:** Measured in GB (e.g., 16 GB)



Dynamic Random-Access Memory (DRAM)

- Each cell has a transistor and a capacitor to store **one bit**
  - Capacitor charged = 1
  - Capacitor empty = 0
- Capacitors lose charge over time…
  - Periodically **refreshes**
  - Data gets rewritten every few milliseconds
- DRAM is **volatile**
  - Data loss occurs when power is lost



Static Random-Access Memory (SRAM)

- Registers and **caches** are SRAM
  - Uses logic (flip-flops!) to store data
- **Latency:** ~0.5ns
- **Cost:** $$$$/GB
- **Size:** Measured in MB
- SRAM is **volatile**
  - Data loss occurs when power is lost

Satyamfifa, Public domain, via Wikimedia Commons

DRAM vs. SRAM

Dynamic Random-Access Memory (DRAM)

- Used for main memory
- Dense and cheap ($3/GB)
- DRAM is **dynamic**
  - Uses **capacitors** to store bits
  - Chip needs to internally refresh data periodically
- **Slow** and **energy-inefficient**

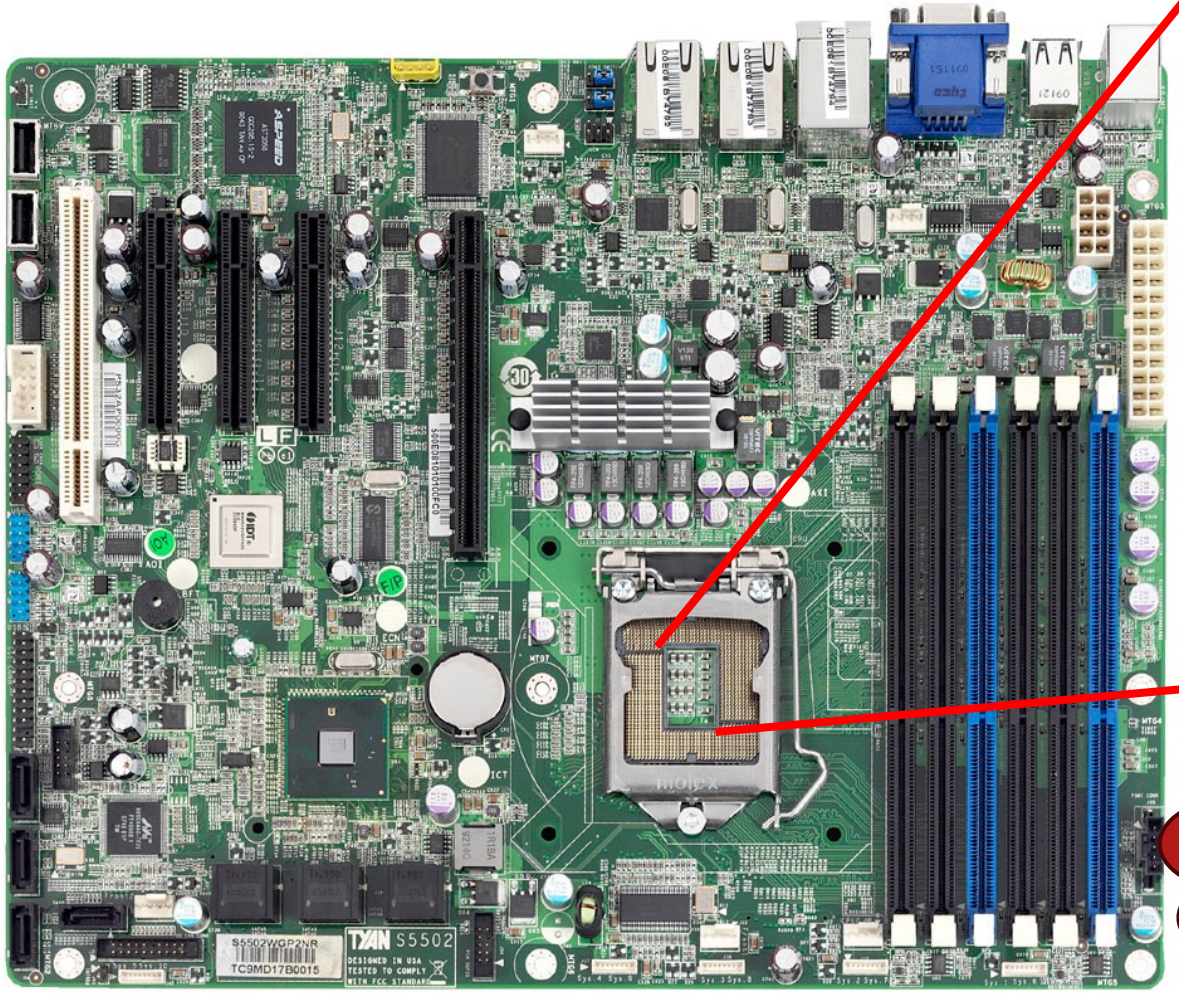
Static Random-Access Memory (SRAM)

- Used for registers and caches
- Small and expensive ($1000s/GB)
- SRAM is **static**
  - Uses logic gates (flip-flops!)
- **Fast** and **energy-efficient**

Storage / Disk / Secondary Memory

- Attached as a **peripheral I/O device: non-volatile**
- **Solid-State Device (SSD)**
  - 40-100µs (~100k processor cycles)
  - $0.05-0.5/GB
  - Usually flash memory
- **Hard-Disk Drive (HDD)**
  - Time access: < 5-10ms (10-20M processor cycles)
  - $0.01-0.1/GB
  - Usually mechanical

Caches!



What data should go in cache?

SandyBridge Motherboard, 2011 http://news.softpedia.com
Intel Pentium 3, 1999

Principle of Locality

Locality Locality Locality

Principle of Locality

Principle of Locality

Temporal Locality

If you need something, you'll likely need it ***again***, soon.

Spatial Locality

If you need something, you'll likely need something else located ***nearby***, soon.

Locality of Data

```
1  int total = 0;
2  for (int i = 1; i < n; i++) {
3    total += a[i];
4  }
5  return total;
```

Locality of Data

```
1   int total = 0;
2   for (int i = 1; i < n; i++) {
3     total += a[i];
4   }
5   return total;
```

`total` has good **temporal** locality because it will be accessed again and again

Locality of Data

```
1  int total = 0;
2  for (int i = 1; i < n; i++) {
3    total += a[i];
4  }
5  return total;
```

`total` has good **temporal** locality because it will be accessed again and again

Accesses to **`a`** have good **spatial** locality because **`a[i+1]`** is accessed soon after **`a[i]`** is

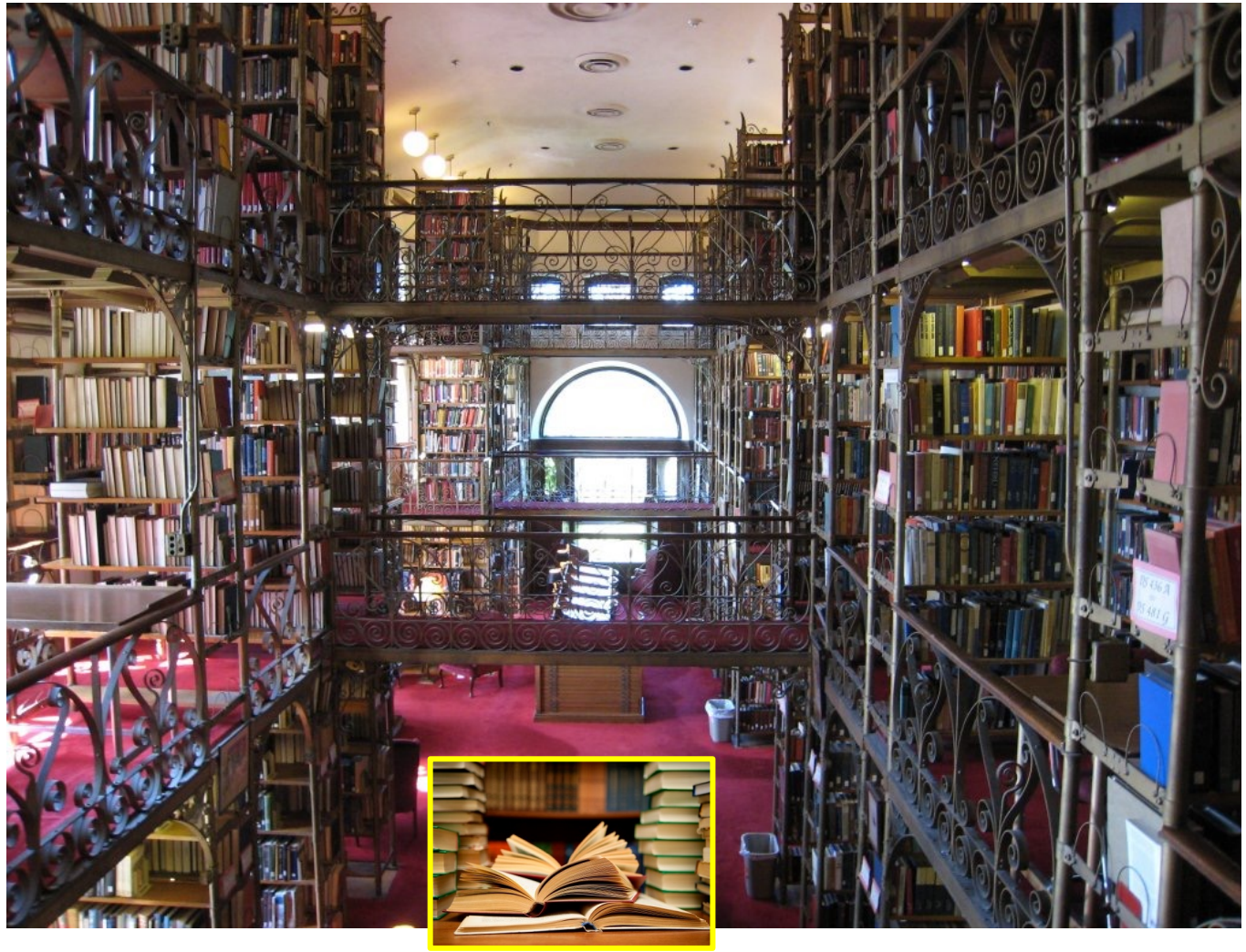
Locality of Code

```
1  total = 0;
2  for (i = 1; i < n; i++) {
3    n--;
4    total += a[i];
5  }
6  return total;
```

Q1: Which line of **code** has the best **temporal** locality?

Q2: Which line of **code** has the best **spatial** locality?

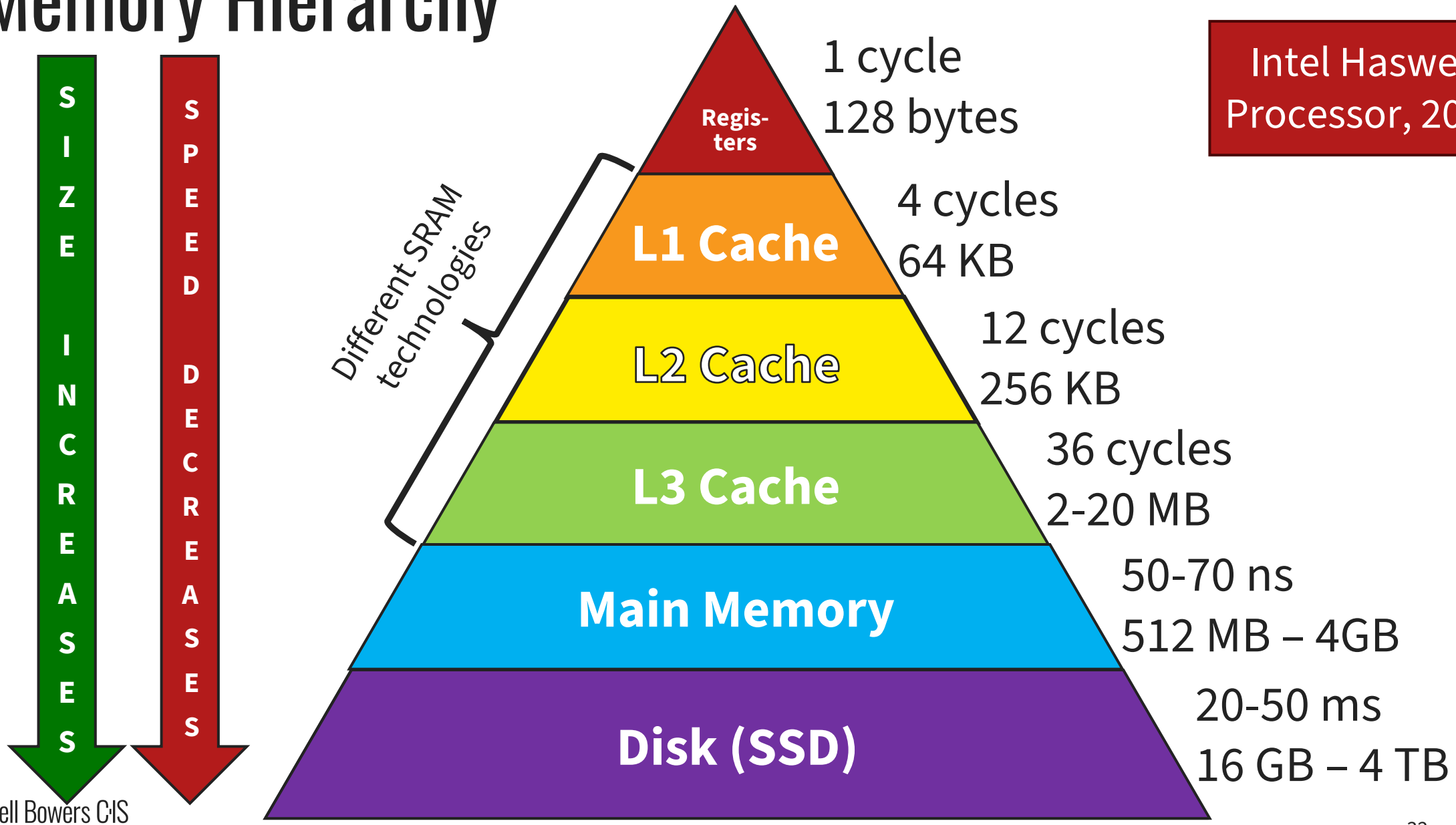
Your Life is Full of Locality



Please take **one minute** to think about an example of **locality** in everyday life, **one minute** to discuss with your neighbor, and **30 seconds** to add to the word cloud.

Memory Hierarchy

Memory Hierarchy



Intel Haswell Processor, 2013

Terminology

Cache hit

- data is in the cache
- **Hit Time:** time it takes to access the cache
- Hit Rate (%hit): # cache hits / # cache accesses

Cache miss

- data is **not** in the cache
- **Miss Penalty:** extra time it takes to retrieve data from lower memory
  - **Miss Time:** time it takes to access the cache (Hit Time) and retrieve data from lower memory (Miss Penalty)
- Miss rate (%miss): # cache misses / # cache accesses

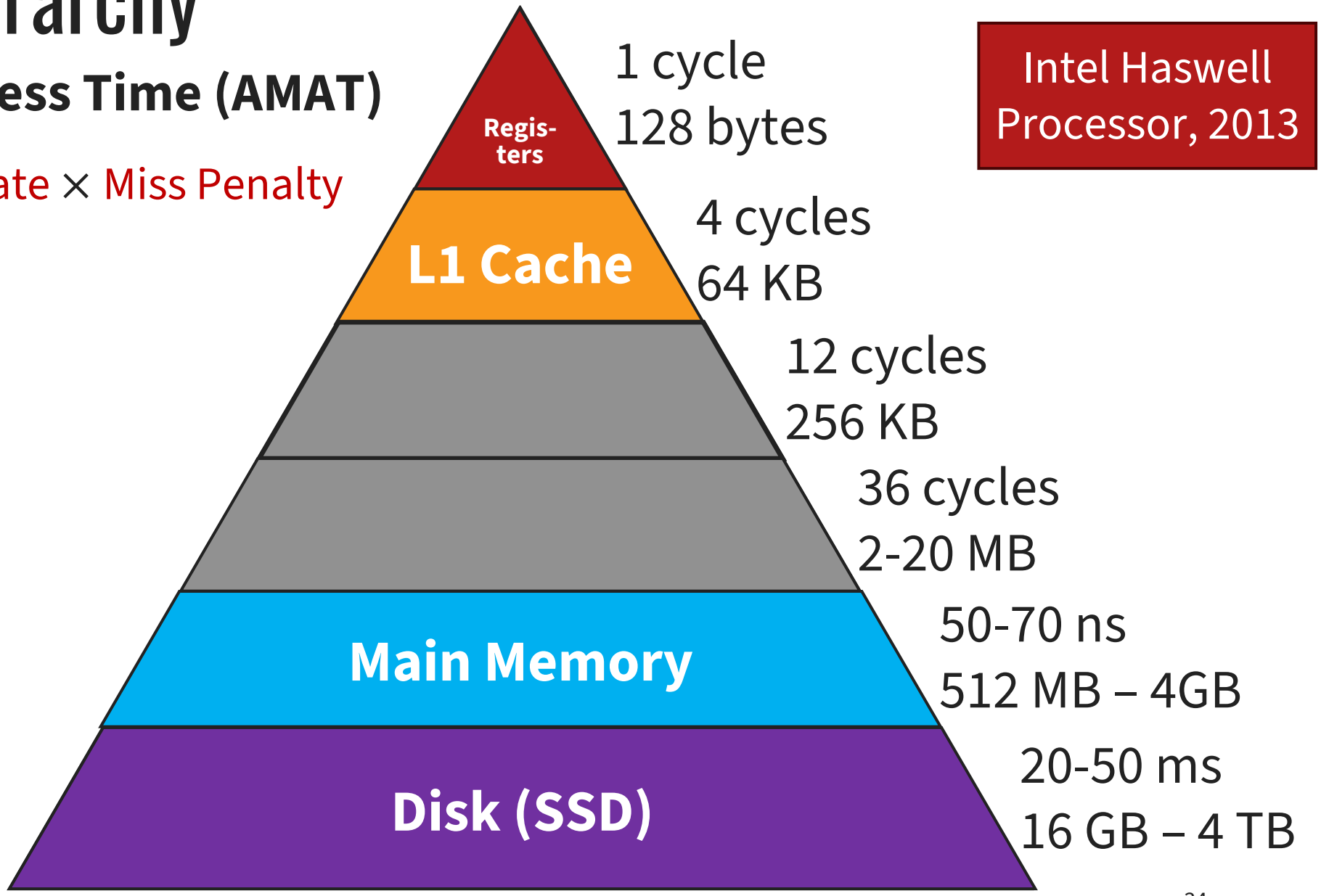
Memory Hierarchy

Average Memory Access Time (AMAT)

AMAT = Hit Time + Miss Rate × Miss Penalty

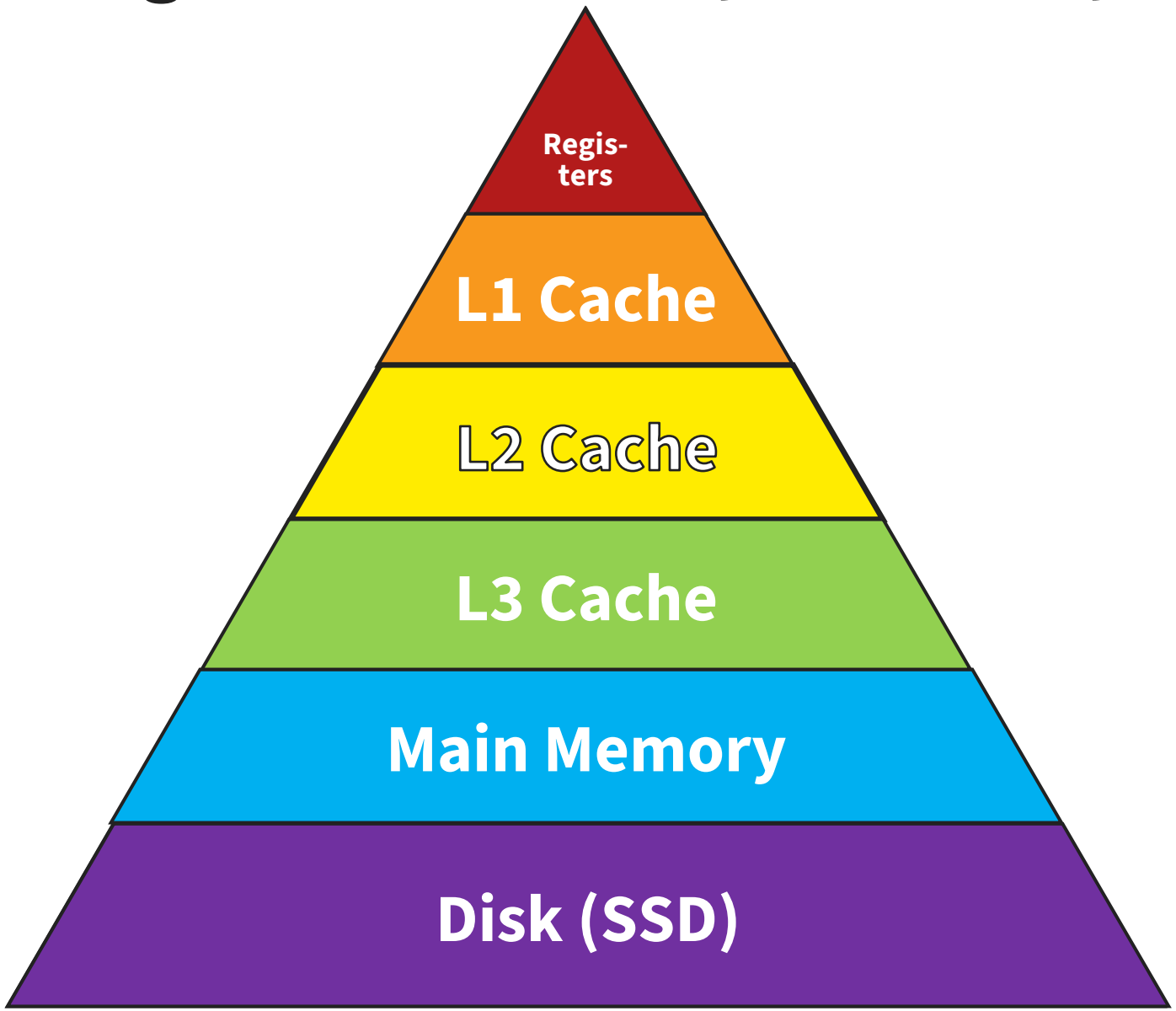
AMAT = 4 + 5% × 100

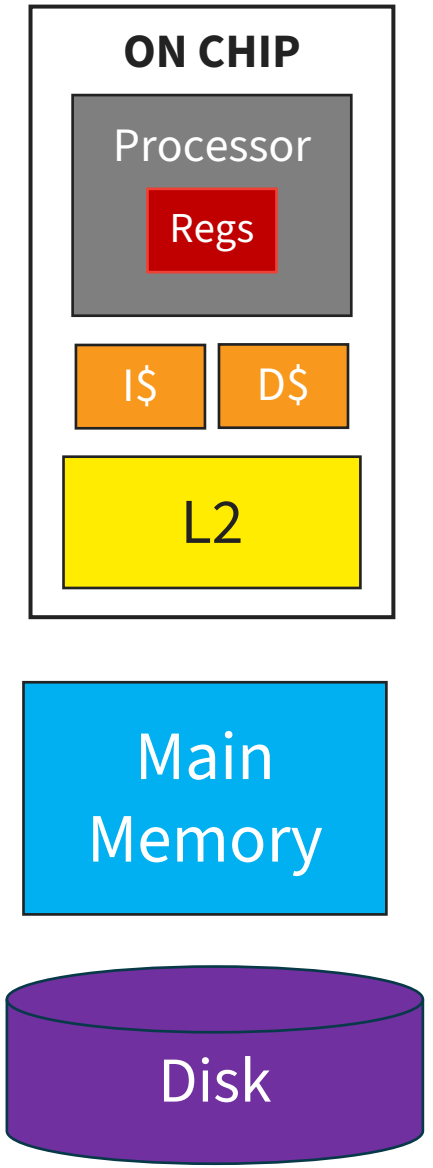
AMAT = **9 cycles**



Intel Haswell Processor, 2013

Single-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	x10-x20	8 GB
SSD (Flash)		5k-50k ns	Tens of thousands	x0.75-x1	512 GB
Disk		5M-20M ns	Millions	x0.05-x0.1	4 TB

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

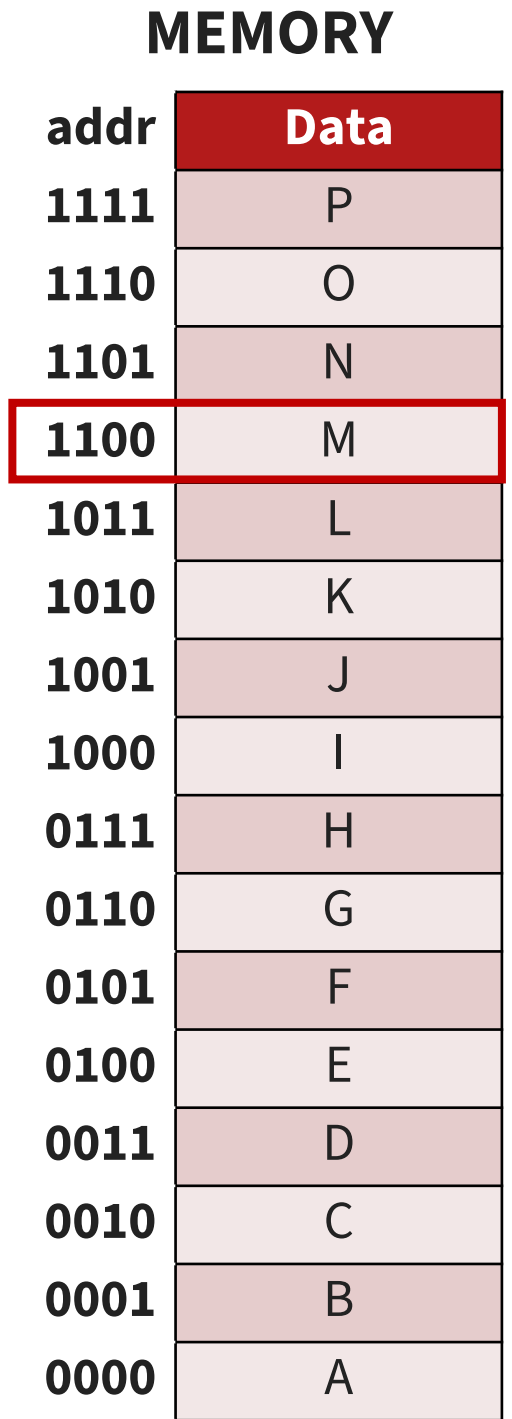
Direct-Mapped Caches

Your first cache!

16 Byte Memory

- Byte-addressable memory
- 4 bit memory addresses

```
load 1100 ⟶ x1
```



4-Byte, Direct-Mapped Cache

CACHE

index XXXX

index	Data
11	D
10	C
01	B
00	A

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

Direct mapped:

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

Indexed with LSB:

- Supports spatial locality

MEMORY

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

Analogy to a Spice Rack

Spice Rack

(Cache)

Index	Spice
A	
B	
C	
D	
E	
F	
…	
Z	

Compared to spice wall:

- Smaller
- Faster
- More costly (per oz.)

Spice Wall

(Memory)

Analogy to a Spice Rack

Spice Rack

(Cache)

Index	Tag	Spice
A		
B		
C	innamon	
D		
E		
F		
…		
Z		

Compared to spice wall:

- Smaller
- Faster
- More costly (per oz.)

Spice Wall

(Memory)

4-Byte, Direct-Mapped Cache

CACHE

index	Tag	Data
11	00	D
10	00	C
01	00	B
00	00	A

```
tag|index
  1101
```

Tag: minimalist label/address

```
address = tag + index
```

MEMORY

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

Cornell Bowers CIS
Computer Science

4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	00	D
10	0	00	C
01	0	00	B
00	0	00	A

```
tag|index
  1101
```

One last tweak: **valid bit**

MEMORY

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

Let's Do Some Examples!

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	0	11	X

```
tag|index
  1101
```

Lookup:

1. Check **index**
2. Check **tag**
3. Check **valid bit**

```
load 1100
```

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	1	11	M

`tag|index`
`1101`

Lookup:

1. Check **index**
2. Check **tag**
3. Check **valid bit**

`load 1100` **MISS**

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	xx	X
10	0	xx	X
01	0	xx	X
00	1	11	M

```
tag|index
  1101
```

Lookup:

1. Check **index**
2. Check **tag**
3. Check **valid bit**

```
load 1100
```

MISS

```
load 1100
```

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	xx	X
10	0	xx	X
01	0	xx	X
00	1	11	M

`tag|index`
`1101`

Lookup:

1. Check **index**
2. Check **tag**
3. Check **valid bit**

`load 1100` **MISS**

`load 1100` **HIT**

MEMORY

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

Caches (2)

CS 3410: Computer System Organization and Programming

Spring 2025

Today's Goals

- Direct Mapped Caches
  - Larger Blocks
- Fully Associative Caches
  - Replacement Policies
- Set-Associative Caches
- Cache Characteristics

4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11			
10			
01			
00			

`tag|index`

`address = tag + index`

Index: index of cache block

Tag: minimalist label/address of block

Valid Bit: whether block is meaningful

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	0	11	X

```
tag|index
  1100
```

Lookup:

1. Check **index**
2. Check **valid bit**
3. Check **tag**

```
load 1100
```

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	1	11	M

`tag|index`
`1100`

Lookup:

1. Check **index**
2. Check **valid bit**
3. Check **tag**

`load 1100` **MISS**

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	1	11	M

`tag|index`
`1100`

Lookup:

1. Check **index**
2. Check **valid bit**
3. Check **tag**

`load 1100` **MISS**

`load 1100`

MEMORY

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

Ex #1: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0	XX	X
10	0	XX	X
01	0	XX	X
00	1	11	M

```
tag|index
  1100
```

Lookup:

1. Check **index**
2. Check **valid bit**
3. Check **tag**

```
load 1100
```
MISS

```
load 1100
```
HIT

MEMORY

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

Ex #2: 4-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0		
10	0		
01	0		
00	0		

`tag|index`
`1100`

Lookup:

1. Check **index**
2. Check **valid bit**
3. Check **tag**

Hit or Miss?

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

MEMORY

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

Reducing Cold Misses by Increasing Block Size

Leveraging Spatial Locality

Increasing Block Size

CACHE

index	V	Tag	Data
11	0	0	G | H
10	0	0	E | F
01	0	0	C | D
00	0	0	A | B

`tag|index|offset`

`1101`

Block size: from 1 byte to 2 bytes

Block offset: least significant bits indicate where you live *in* the block

MEMORY

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

Ex #3: 8-Byte, Direct-Mapped Cache

CACHE

index	V	Tag	Data
11	0		
10	0		
01	0		
00	0		

`tag|index|offset`

`1101`

Hit or Miss?

Lookup:

1. Check **index**
2. Check **tag**
3. Check **valid bit**

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

MEMORY

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

Fully Associative Cache

Any address, any entry

8-Byte, Fully Associative Cache

CACHE

V	Tag	Data
0	010	E | F
0	000	A | B
0	001	C | D
0	011	G | H

`tag|offset`
`1101`

No more index!

- Any address, any entry
- 4-bit addresses
- 2-byte blocks

MEMORY

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

8-Byte, Fully Associative Cache

CACHE

V	Tag	Data
0		
0		
0		
0		

`tag|offset`
`1101`

Hit or Miss?

For every entry:

1. Check **tag**
2. Check **valid bit**

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

MEMORY

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

Replacement Policy

Which entry should we evict?

- **Least-recently used (LRU):** Evict block that last used longest ago
  - Lots of bookkeeping
- **Not most-recently used (NMRU):** Randomly pick block that isn't the most recently accessed one
  - Worse decisions, but cheaper
- **FIFO**: replace oldest line

Pros & Cons of Full Associativity

Pros:

- No more conflicts!
- Excellent utilization!

Cons:

- Need to search *all* entries on *every* access
- Expensive

Set-Associative Cache

The best of both worlds!

Set-Associative Cache

index	Way 1 V	Way 1 Tag	Way 1 Data	Way 2 V	Way 2 Tag	Way 2 Data
1	0	XX	X | X	0	XX	X | X
0	0	XX	X | X	0	XX	X | X

Divide cache into groups of size 2^k (**ways**)

- $\frac{2^n}{2^k} = 2^{n-k}$ sets
- Ways within each set are **associative**

`tag|index|offset`
`1100`

- 4-bit addresses
- 4-entry cache
- 2-way set associative
- 2 byte blocks

MEMORY

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

8-byte, 2-way Set-Associative Cache

index	Way 1 V	Way 1 Tag	Way 1 Data	Way 2 V	Way 2 Tag	Way 2 Data
1	0			0		
0	0			0		

```
tag|index|offset
    1100
```

1. **Index** into set
2. Search for **tag** across **ways**
3. Check **valid bit**

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

Hit or Miss?

MEMORY

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

Cache Characteristics

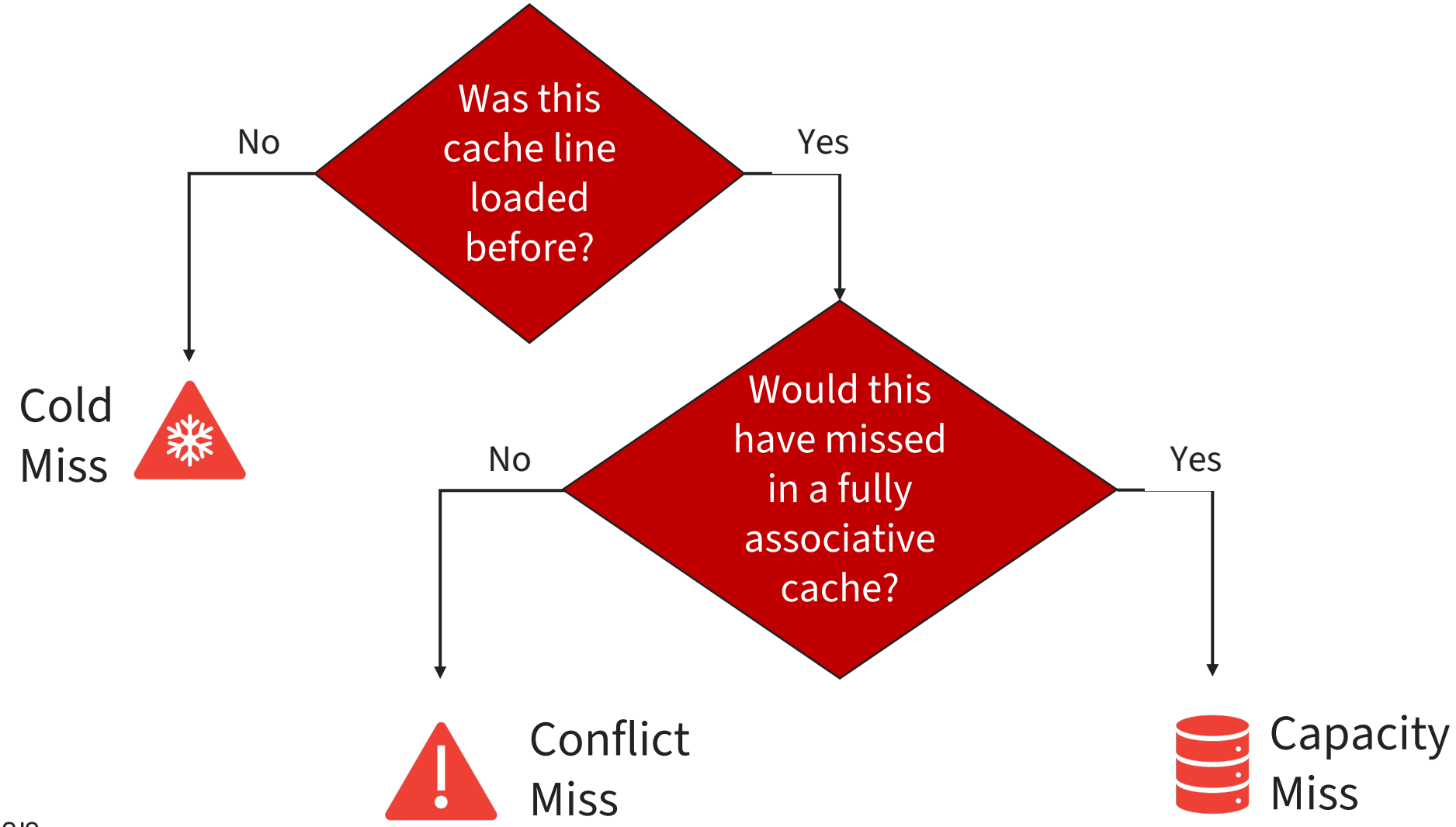
The Three C's of Cache Misses

Cold (Compulsory): never seen this cache line before

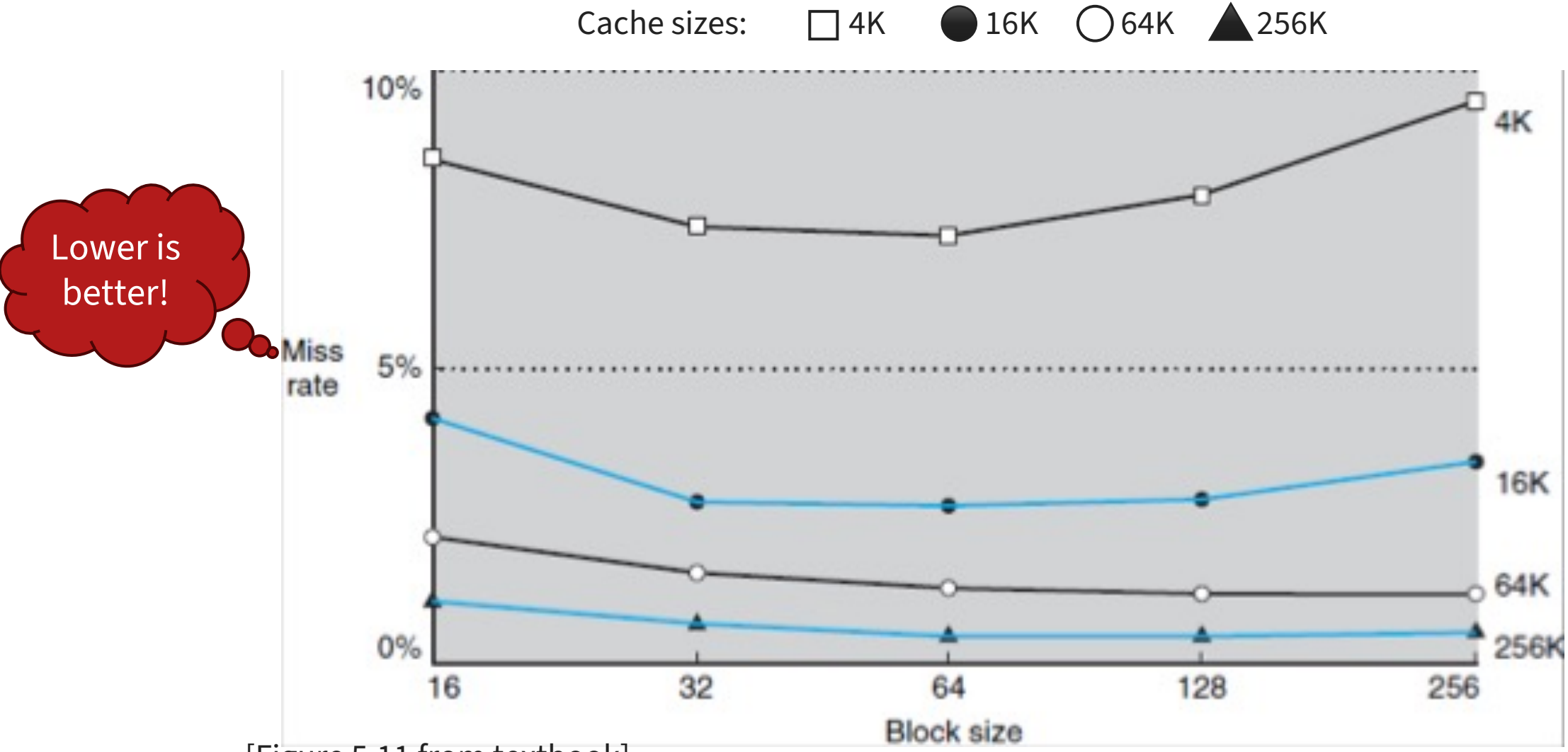
Conflict: cache associativity is too low

Capacity: cache is too small

Characterizing Misses



Miss Rate vs. Block Size



[Figure 5.11 from textbook]

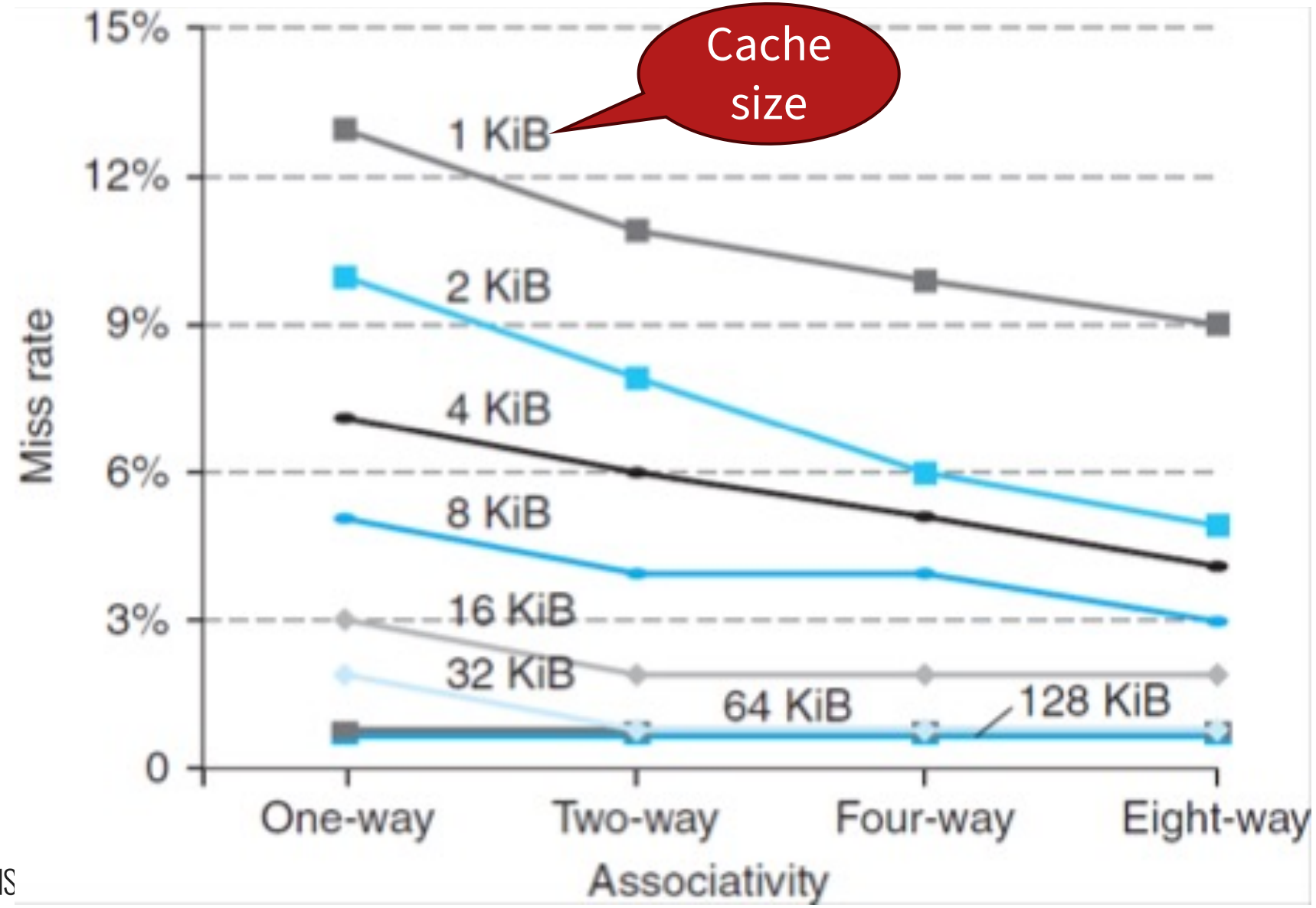
Bigger block size helps at first, but then, not so much.

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 cache
  - and larger miss penalty (time to fetch block)

Miss Rate vs. Associativity



Lower is better!

Today's Goals

- Direct Mapped Caches
  - Larger Blocks
- Fully Associative Caches
  - Replacement Policies
- Set-Associative Caches
- Cache Characteristics

Caches (3)

CS 3410: Computer System Organization and Programming

Spring 2025

Today's Goals

- Review: Set-Associative Caches
- Review: Cache Characteristics
- Cache Performance
- Handling Stores
- Cache Conscious Programming

Set-Associative Cache

	Way 1			Way 2		
index	V	Tag	Data	V	Tag	Data
1	0	XX	X | X	0	XX	X | X
0	0	XX	X | X	0	XX	X | X

Divide 2^n cache entries into *m* **sets**

- Each set has *k* **ways**
- Index is used to map each address to a **set**
- Can store block in any **way** within **set**

`tag|index|offset`
`1100`

- 4-bit addresses
- 4-entry cache
- 2-way set associative
- 2 byte blocks

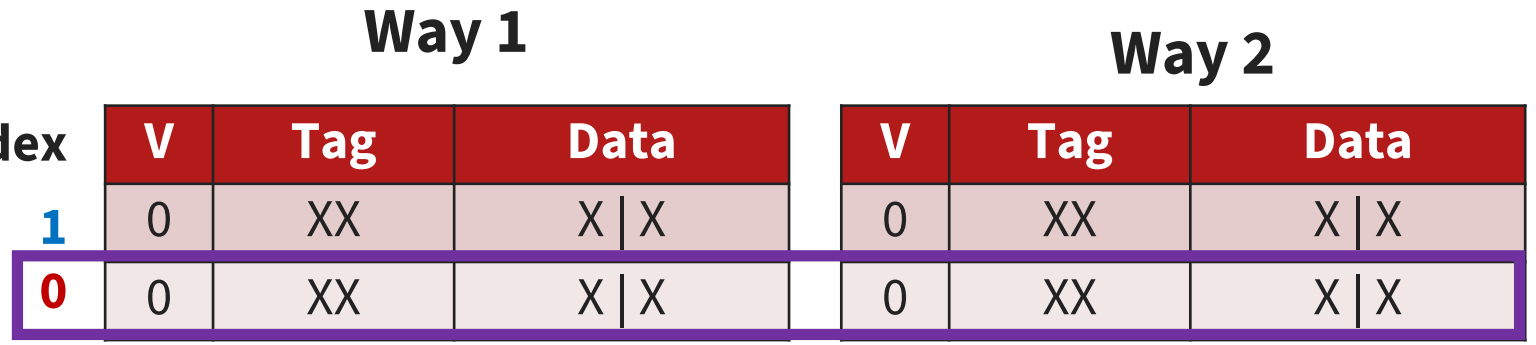
MEMORY

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

Set-Associative vs. Direct-Mapped vs. Fully Associative

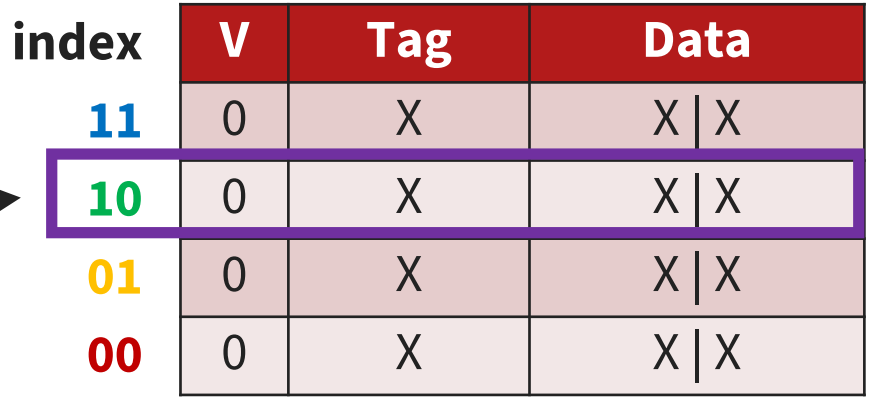
Set-Associative
- 2 sets
- 2 ways

Address
1101



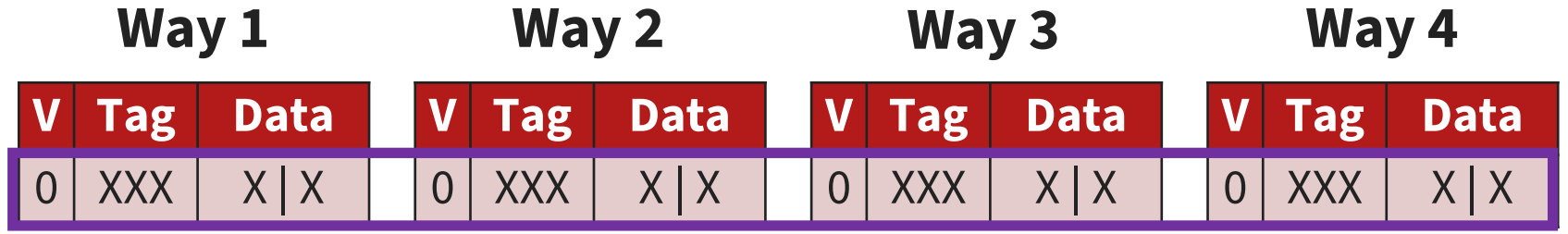
Direct-Mapped
- 4 sets
- **1 way**

Address
1101

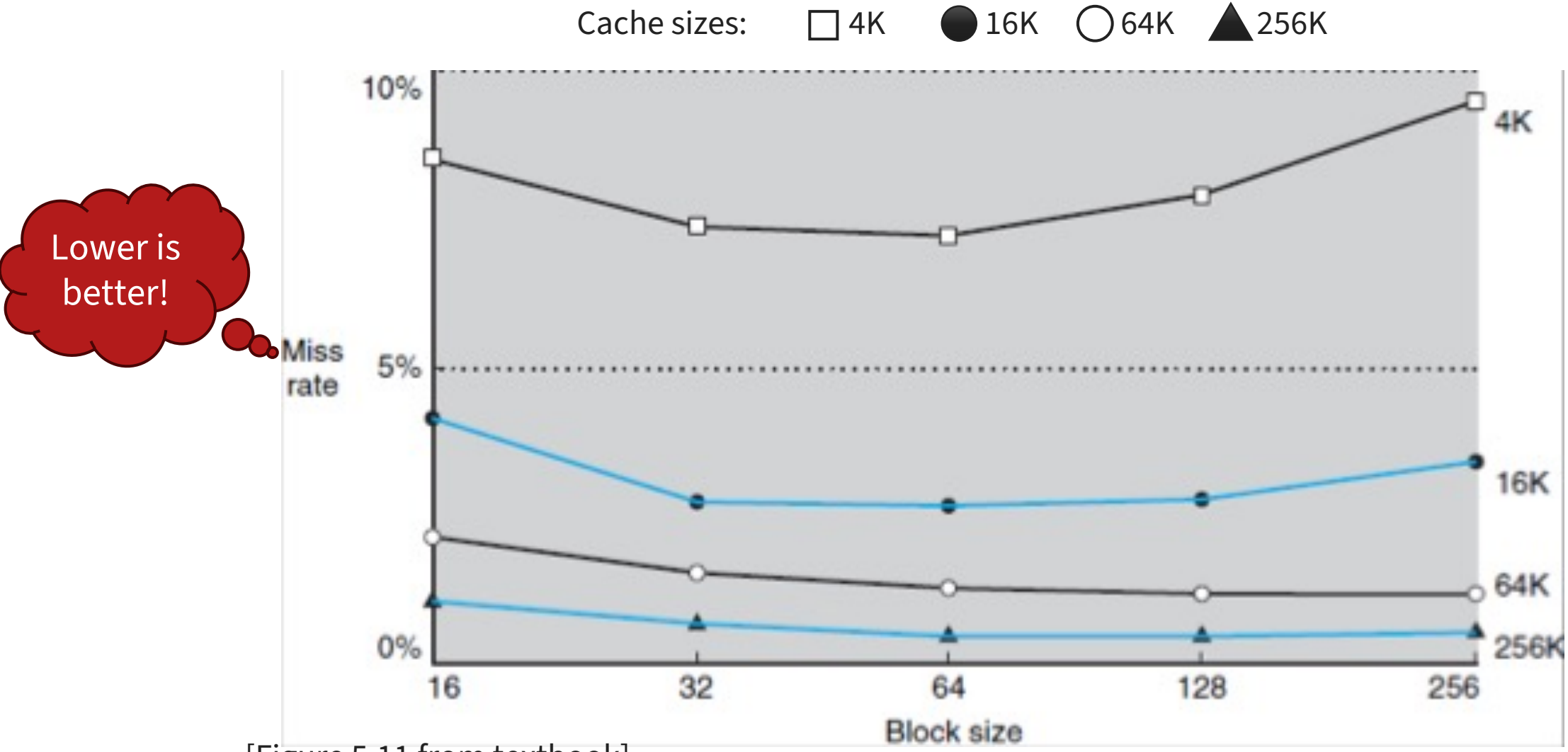


Fully Associative
- **1 set**
- 4 ways

Address
1101



Miss Rate vs. Block Size



[Figure 5.11 from textbook]

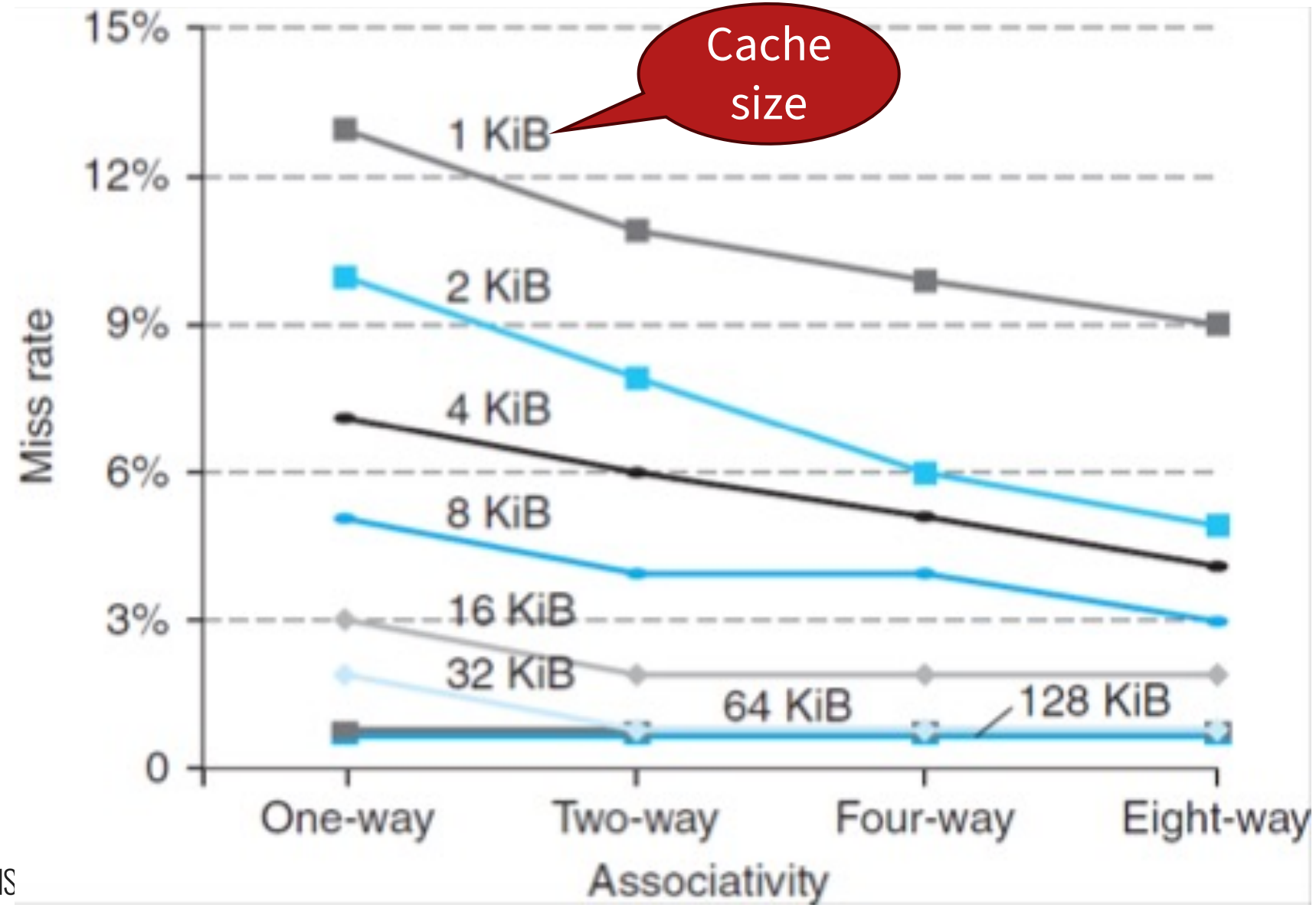
Bigger block size helps at first, but then, not so much.

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 cache
  - and larger miss penalty (time to fetch block)

Miss Rate vs. Associativity



Lower is better!

PollEverywhere

What does **NOT** happen when you increase the associativity of the cache (holding cache size constant)?

A. Conflict misses decreases
B. Tag overhead decreases
C. Hit time increases

ABCs of Caches

+ Associativity:
  - ⬇️conflict misses ☺
  - ⬆️hit time ☹
+ Block Size:
  - ⬇️cold misses ☺
  - ⬆️conflict misses ☹
+ Capacity:
  - ⬇️capacity misses ☺
  - ⬆️hit time ☹

Cache Performance

Average Memory Access Time (AMAT)

$$t_{avg} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

Hit Time: time to access the cache

Miss Rate: fraction of accesses that are misses

Miss Penalty: time it takes to retrieve data from lower memory structure

---

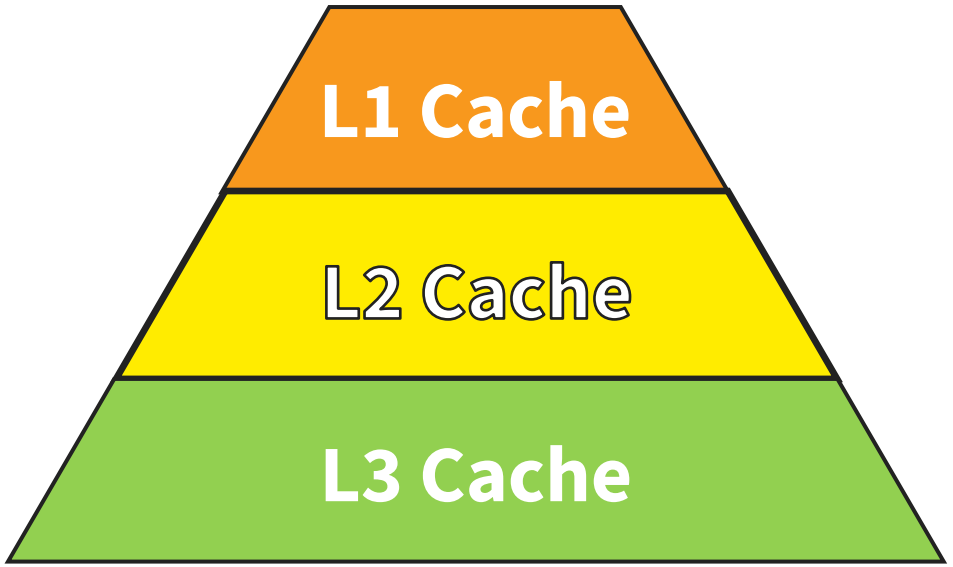
Example:

Cache Access: 1 ns
Main Memory Access: 50 ns
Hit Rate: 95%

$$1 + 0.05 \times 50 = 3.5 \text{ ns}$$

Which caches get what properties?

t_{avg} = Hit Time + Miss Rate × Miss Penalty



Fast

Big

Design with speed in mind

More Associative
Bigger Block Sizes
Larger Capacity

Design with miss rate in mind

Example: AMAT w/ multiple caches

$$t_{avg} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

L1 Cache:

- Latency: 1 ns
- Miss Rate: 5%

L2 Cache:

- Latency: 10 ns
- Miss Rate: 20%

Main Memory:

- Latency: 50 ns

$$t_{avg}^{L1} = t_{hit}^{L1} + \%_{miss}^{L1} \times t_{miss}^{L1}$$

$$t_{avg}^{L2} = t_{hit}^{L2} + \%_{miss}^{L2} \times t_{miss}^{L2}$$

$$t_{avg}^{mem} = 50 \text{ ns}$$

Cornell Bowers CIS
Computer Science

Example: AMAT w/ multiple caches

$$t_{avg} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

L1 Cache:
- Latency: 1 ns
- Miss Rate: 5%

L2 Cache:
- Latency: 10 ns
- Miss Rate: 20%

Main Memory:
- Latency: 50 ns

$$t_{avg}^{L1} = t_{hit}^{L1} + \%_{miss}^{L1} \times t_{miss}^{L1}$$
$$= 1\text{ ns} + 5\% \times 20\text{ ns}$$
$$= 2\text{ ns}$$

$$t_{avg}^{L2} = t_{hit}^{L2} + \%_{miss}^{L2} \times t_{miss}^{L2}$$
$$= 10\text{ ns} + 20\% \times 50\text{ ns}$$
$$= 20\text{ ns}$$

$$t_{avg}^{mem} = 50\text{ ns}$$

PollEverywhere

$$t_{avg} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$$

You have a 1 GHz processor (1 cycle = 1 ns). It has an L1 cache with a 1 cycle access time and a 50% hit rate, and an L2 cache with a 10 cycle access time and a 90% hit rate. Your main memory has a latency of 100 ns.
Q: What is the average memory access time of this processor in ns?

A: 5 B: 7 C: 9 D: 11 E: 13

Performance Summary

Average memory access time (AMAT) depends on:

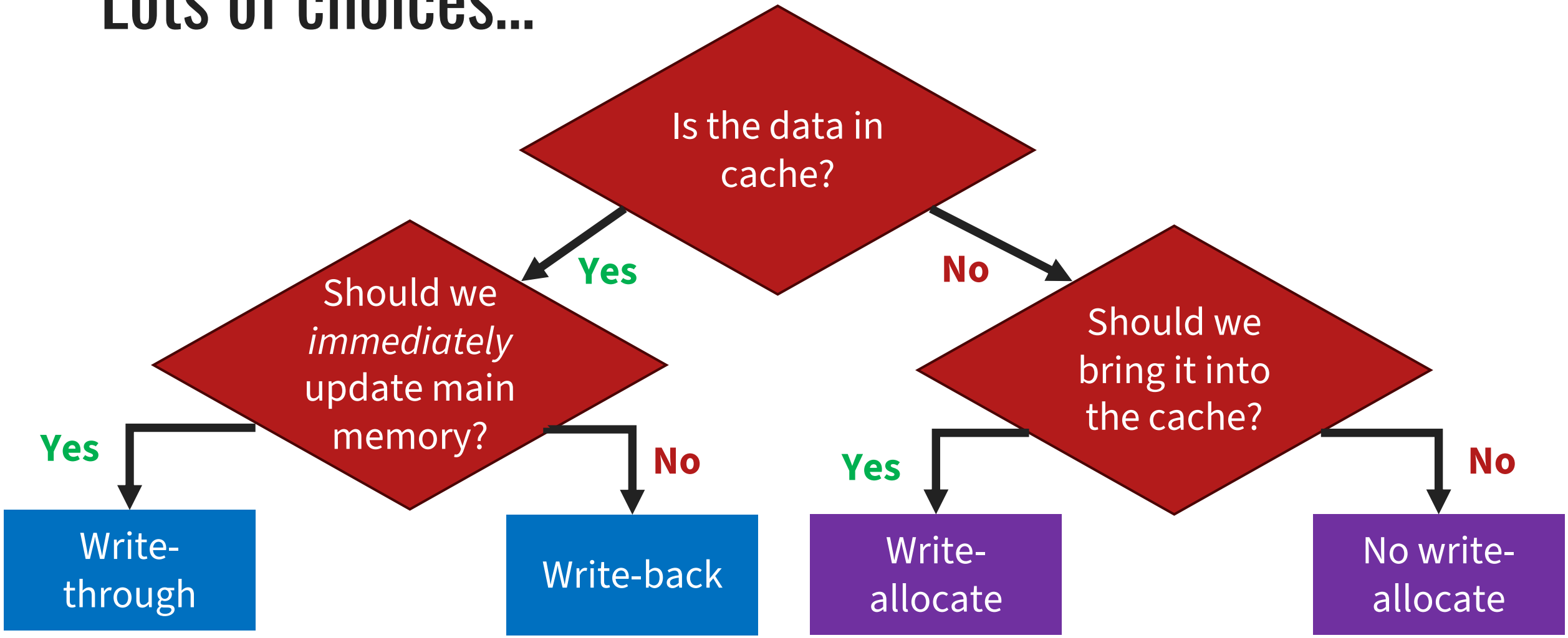
- Cache architecture and size
- Hit and miss rates
- Access times and miss penalty

Cache design is a very complex problem:

- Cache size
- Block size (a.k.a., line size)
- Number of ways of set-associativity (1, n, ∞)
- Replacement policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or unified cache
- Write policy

Handling Stores

Lots of choices…



Write-back Caches

CACHE

index	V	Dirty?	Tag	Data
11				
10				
01				
00				

`tag|index|offset`

`1101`

MEMORY

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

Update main memory only when data is **evicted**

- Minimize costly stores

Dirty Bit

- Clean (0): in sync with main memory
- Dirty (1): out of sync with main memory

Cornell Bowers CIS
Computer Science

Write-back Caches

CACHE

index	V	Dirty?	Tag	Data
11				
10				
01				
00				

`tag|index|offset`

`1101`

MEMORY

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

Algorithm Updates:

- When ***filling*** an entry, set dirty bit to 0
- When ***storing*** to an entry, set dirty bit to 1
- When ***evicting*** an entry, check dirty bit:
  - If clean, then do nothing.
  - If dirty, write data to main memory

Example: Write-back Caches

CACHE

index	V	Dirty?	Tag	Data
11				
10				
01				
00				

`tag|index|offset`

`1101`

```
load 1100

store 'Z', 1101

load 0100

load 1100
```

MEMORY

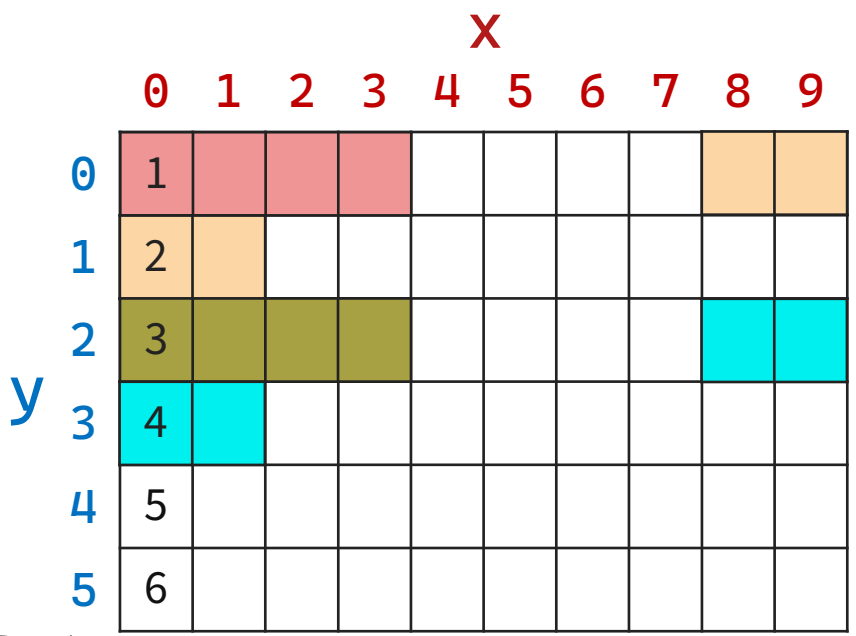
addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

Cache Conscious Programming

Be nice to the cache, and the cache will be nice to you.

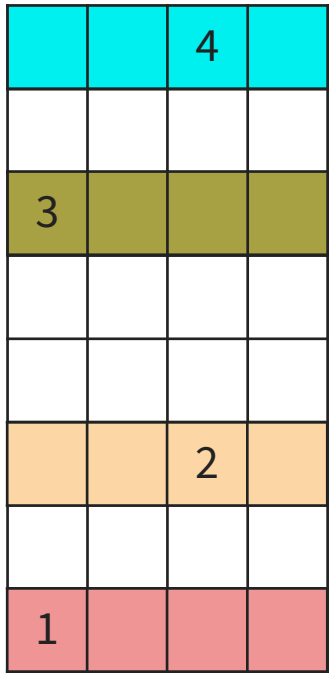
Cache Conscious Programming

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




```
1: A[0][0]
2: A[1][0]
3: A[2][0]
4: A[3][0]
5: A[4][0]
6: A[5][0]
7: A[0][1]
8: A[1][1]
 . . .
```

All cache misses!



4 byte blocks

CACHE

MEMORY

Cache Conscious Programming

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

x

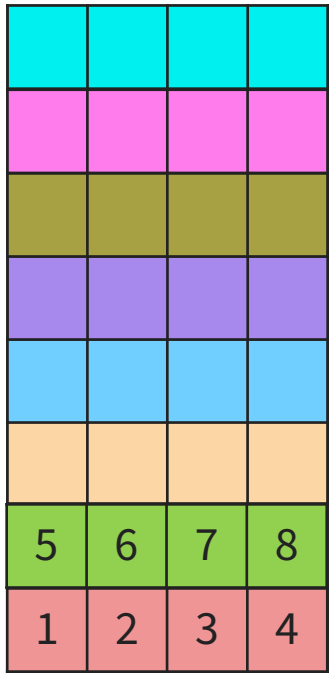
y

	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8		
1										
2										
3										
4										
5										

```
1: A[0][0]
2: A[0][1]
3: A[0][2]
4: A[0][3]
5: A[0][4]
6: A[0][5]
7: A[0][6]
8: A[0][7]
 . . .
```

75% hit rate!

4 byte blocks



CACHE

MEMORY

PollEverywhere

Choose the best block size for your cache among the choices given. Assume that integers and pointers are all 4 bytes each and that the `scores` array is 4-byte aligned.

a) 1 byte

b) 4 bytes

c) 8 bytes

d) 16 bytes

e) 32 bytes

```
int scores[NUM STUDENTS] = 0;
int sum = 0;
for (i = 0; i < NUM STUDENTS; i++) {
  sum += scores[i];
}
```

PollEverywhere

Choose the best block size for your cache among the choices given. Assume integers and pointers are 4 bytes.

a) 1 byte
b) 4 bytes
c) 8 bytes
d) 16 bytes
e) 32 bytes

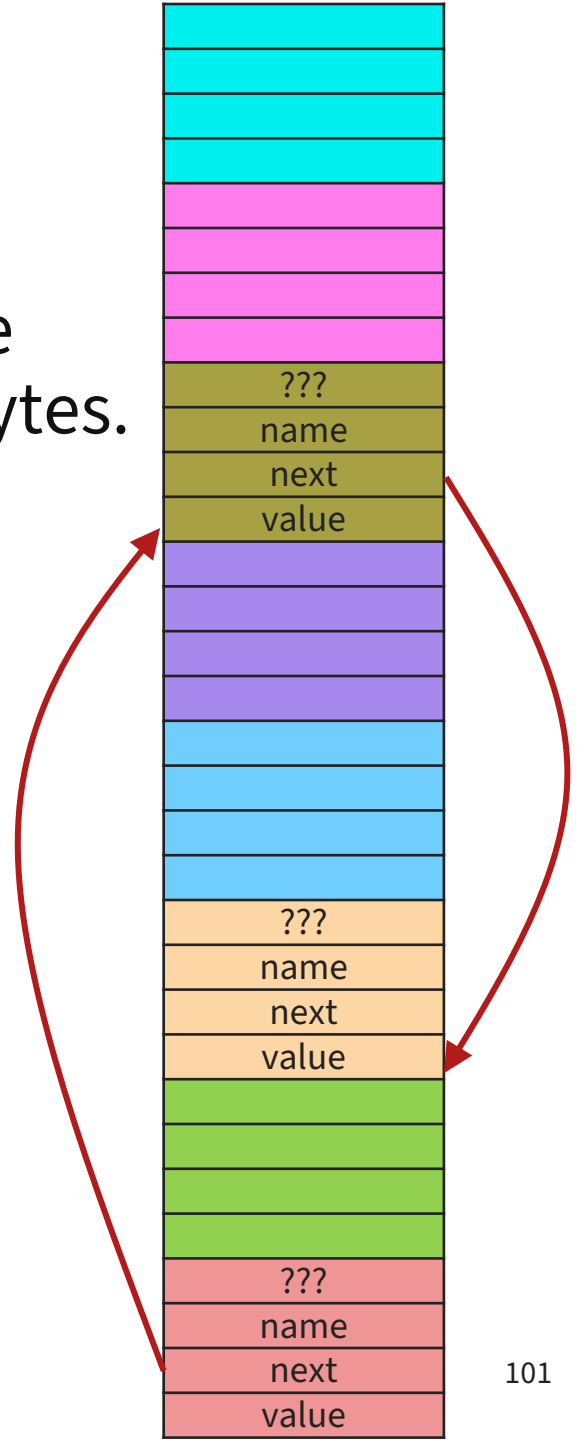
```
typedef struct item_t {
  int value;
  struct item_t *next;
  char *name;
} item_t;
int sum = 0;
item_t *curr = head;
while (curr ≠ NULL) {
  sum += curr→value;
  curr = curr→next;
}
```

PollEverywhere

Choose the best block size for your cache among the choices given. Assume integers and pointers are 4 bytes.

a) 1 byte
b) 4 bytes
c) 8 bytes
d) 16 bytes
e) 32 bytes

```
typedef struct item_t {
  int value;
  struct item_t *next;
  char *name;
} item_t;
int sum = 0;
item_t *curr = head;
while (curr ≠ NULL) {
  sum += curr→value;
  curr = curr→next;
}
```



Today's Goals

- Review: Set-Associative Caches
- Review: Cache Characteristics
- Cache Performance
- Handling Stores
- Cache Conscious Programming