

inst.eecs.berkeley.edu/~cs61c/su05  
**CS61C : Machine Structures**

## Lecture #20: Caches 2

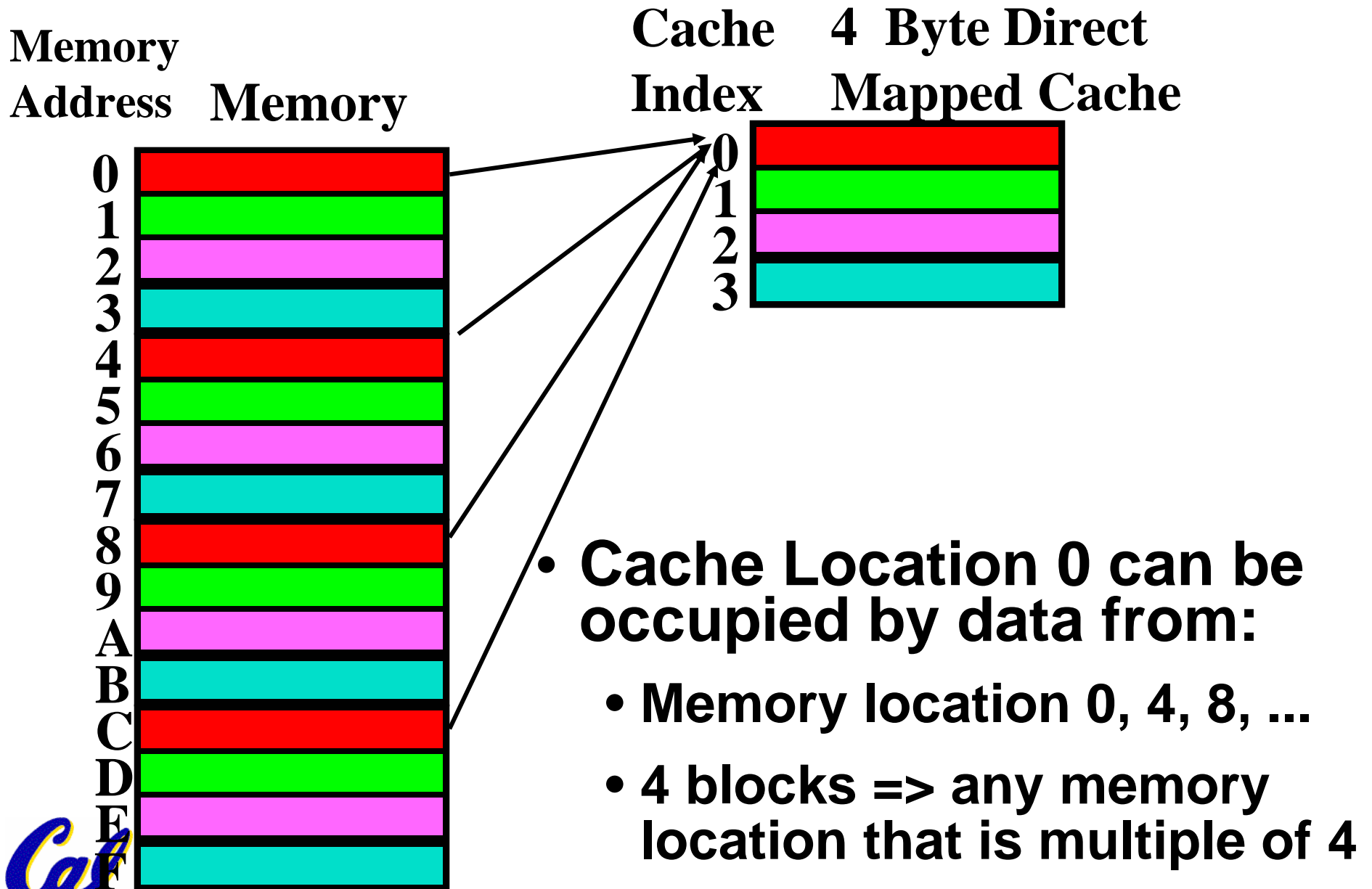


**2005-07-26**

**Andy Carle**



# Review: Direct-Mapped Cache



# Issues with Direct-Mapped



- Since multiple memory addresses map to same cache index, how do we tell which one is in there?
- What if we have a block size > 1 byte?
- Answer: divide memory address into three fields

**HEIGHT**      **WIDTH**



tag  
to check  
if have  
correct block

index  
to  
select  
block

byte  
offset  
within  
block



# Direct-Mapped Cache Terminology

- All fields are read as unsigned integers.
- **Index**: specifies the cache index (which “row” of the cache we should look in)
- **Offset**: once we’ve found correct block, specifies which byte within the block we want -- i.e., which “column”
- **Tag**: the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location



## Direct-Mapped Cache Example (1/3)

- Suppose we have a 16KB of data in a direct-mapped cache with 4 word blocks
- Determine the size of the tag, index and offset fields if we're using a 32-bit architecture
- Offset
  - need to specify correct byte within a block
  - block contains 4 words
    - = 16 bytes
    - =  $2^4$  bytes
  - need **4 bits** to specify correct byte



## Direct-Mapped Cache Example (2/3)

- Index: (~index into an “array of blocks”)
  - need to specify correct row in cache
  - cache contains 16 KB =  $2^{14}$  bytes
  - block contains  $2^4$  bytes (4 words)
  - # blocks/cache
    - =  $\frac{\text{bytes/cache}}{\text{bytes/block}}$
    - =  $\frac{2^{14} \text{ bytes/cache}}{2^4 \text{ bytes/block}}$
    - =  $2^{10}$  blocks/cache
  - need 10 bits to specify this many rows



## Direct-Mapped Cache Example (3/3)

- **Tag: use remaining bits as tag**
  - tag length = addr length - offset - index  
= 32 - 4 - 10 bits  
= 18 bits
  - so tag is leftmost **18 bits** of memory address
- **Why not full 32 bit address as tag?**
  - All bytes within block need same address (4b)
  - Index must be same for every address within a block, so its redundant in tag check, thus can leave off to save memory (10 bits in this example)



# TIO

AREA (cache size, B)  
= HEIGHT (# of blocks)  
\* WIDTH (size of one block, B/block)

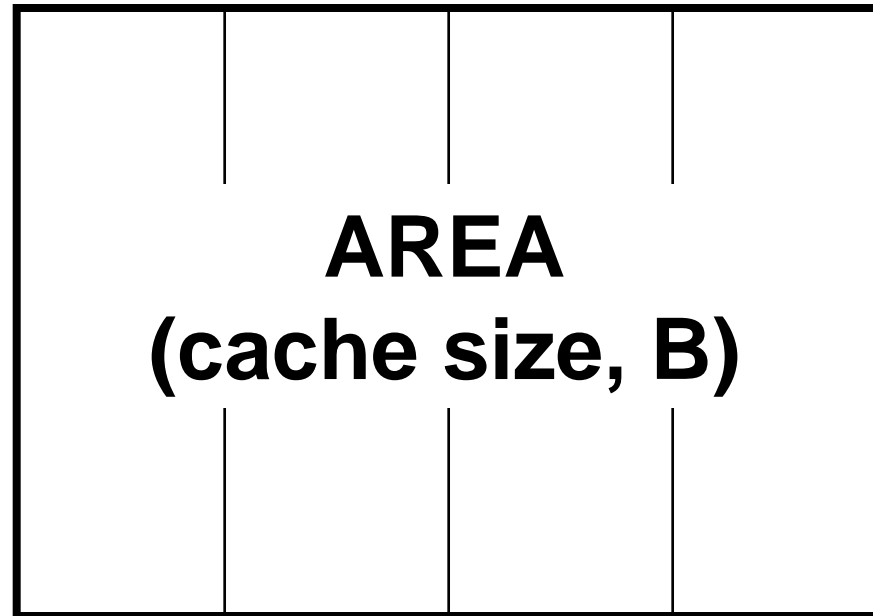
$$2^{(H+W)} = 2^H * 2^W$$



WIDTH  
(size of one block, B/block)



HEIGHT  
(# of blocks)





# Caching Terminology

---

- When we try to read memory, 3 things can happen:
  1. **cache hit**:  
cache block is valid and contains proper address, so read desired word
  2. **cache miss**:  
nothing in cache in appropriate block, so fetch from memory
  3. **cache miss, block replacement**:  
wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)



# Accessing data in a direct mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks
- Read 4 addresses
  1. 0x00000014
  2. 0x0000001C
  3. 0x00000034
  4. 0x00008014
- Memory values on right:
  - only cache/memory level of hierarchy

**Memory**  
Address (hex) Value of Word

...	...
00000010	a
<u>00000014</u>	b
00000018	c
<u>0000001C</u>	d

...	...
00000030	e
<u>00000034</u>	f
00000038	g
0000003C	h

...	...
00008010	i
<u>00008014</u>	j
00008018	k
0000801C	l



# Accessing data in a direct mapped cache

- **4 Addresses:**

- 0x00000014, 0x0000001C,  
0x00000034, 0x00008014

- **4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields**

000000000000000000000000 0000000001 0100

000000000000000000000000 0000000001 1100

000000000000000000000000 0000000011 0100

000000000000000000000010 0000000001 0100

**Tag**

**Index**

**Offset**



# 16 KB Direct Mapped Cache, 16B blocks

- **Valid bit:** determines whether anything is stored in that row (when computer initially turned on, all entries invalid)

**Valid**

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# 1. Read 0x00000014

- 000000000000000000000000 0000000001 0100
- Tag field
Index field
Offset

Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
Index					
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# So we read block 1 (0000000001)

- 00000000000000000000 0000000001 0100  
 Tag field Index field Offset

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
<u>1</u>	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# No valid data

- 000000000000000000000000 0000000001 0100  
 Tag field Index field Offset

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
<u>1</u>	0				
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# So load that data into cache, setting tag, valid

- 00000000000000000000 0000000001 0100
- Tag field
Index field
Offset

Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
Index					
0	0				
1	1	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				





# Read from cache at offset, return word b

- 000000000000000000000000 0000000001 0100  
Tag field
Index field
Offset

Valid	Index	Tag	0x0-3	<u>0x4-7</u>	0x8-b	0xc-f
	0	0				
	<u>1</u>	0	a	<b>b</b>	c	d
	2	0				
	3	0				
	4	0				
	5	0				
	6	0				
	7	0				
	...					
	1022	0				
	1023	0				



## 2. Read 0x0000001C = 0...00 0..001 1100

- 00000000000000000000 0000000001 1100

Tag field
Index field
Offset

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	<b>0</b>	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# Index is Valid

- 000000000000000000000000 0000000001 1100

Tag field

Index field

Offset

Valid

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
<u>1</u>	0	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				



# Index valid, Tag Matches

- 000000000000000000000000 000000000001 1100

Valid Index	Tag	Tag field		Index field		Offset
		0x0-3	0x4-7	0x8-b	0xc-f	
0	0					
1	0	a	b	c	d	
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					



# Index Valid, Tag Matches, return d

- 000000000000000000000000 000000000001 1100

Valid Index	Tag	Tag field		Index field		Offset
		0x0-3	0x4-7	0x8-b	0xc-f	
0	0					
1	0	a	b	c	d	
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...						
1022	0					
1023	0					



### 3. Read 0x00000034 = 0...00 0..011 0100

- 000000000000000000000000 0000000011 0100  
 Valid Tag field Index field Offset

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				
...			...		
1022	0				
1023	0				



# So read block 3

- 000000000000000000000000 0000000011 0100

Valid

Tag field

Index field

Offset

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	0	a	b	c	d
2	0				
3	0				
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				



# No valid data

- 000000000000000000000000 0000000011 0100  
 Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					

...

...

1022	0					
1023	0					





# Load that cache block, return word f

- 000000000000000000000000 0000000011 0100  
 Valid Tag field Index field Offset

Index	Valid	Tag	0x0-3	<u>0x4-7</u>	0x8-b	0xc-f
0	0					
1	1	0	a	b	c	d
2	0					
<u>3</u>	<u>1</u>	<u>0</u>	e	<u>f</u>	g	h
4	0					
5	0					
6	0					
7	0					

...

...

1022	0					
1023	0					



# 4. Read 0x00008014 = 0...10 0..001 0100

- 00000000000000000010 0000000001 0100

Valid

Tag field

Index field

Offset

Index	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0				
1	1	0	a	b	c
2	0				
3	1	0	e	f	g
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				



# So read Cache Block 1, Data is Valid

- 00000000000000000010 0000000001 0100

Valid

Tag field

Index field

Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
<u>1</u>	1	0	a	b	c	d
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					

...

...

1022	0					
1023	0					



# Cache Block 1 Tag does not match (0 != 2)

- 0000000000000000000010 00000000001 0100  
 Tag field Index field Offset

Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0					
<u>1</u>	<u>0</u>	a	b	c	d
2					
3	0	e	f	g	h
4					
5					
6					
7					
...			...		
1022	0				
1023	0				



# Miss, so replace block 1 with new data & tag

- 0000000000000000000010 00000000001 0100

Valid                      Tag field                      Index field                      Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					

...

...

1022	0					
1023	0					



# And return word j

• 00000000000000000010    0000000001    0100  
 Valid            Tag field            Index field    Offset

Index	Tag	0x0-3	<u>0x4-7</u>	0x8-b	0xc-f
0	0				
1	1	2	i	k	l
2	0				
3	1	0	e	f	g
4	0				
5	0				
6	0				
7	0				

...

...

1022	0				
1023	0				



# Peer Instruction #1

- Chose from: Cache: Hit, Miss, Miss w. replace  
Values returned: a ,b, c, d, e, ..., k, l
- Read address **0x00000030** ?  
000000000000000000000000 0000000011 0000
- Read address **0x0000001c** ?  
000000000000000000000000 0000000001 1100

Cache

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	i	j	k	l
2	0					
3	1	0	e	f	g	h
4	0					
5	0					
6	0					
7	0					

...



# Answers

- **0x00000030** a hit

Index = 3, Tag matches,  
Offset = 0, value = **e**

- **0x0000001c** a miss

Index = 1, Tag mismatch,  
so replace from memory,  
Offset = 0xc, value = **d**

- Since reads, values must = memory values whether or not cached:

- **0x00000030** = **e**

- **0x0000001c** = **d**

## Memory

Address Value of Word

Address	Value of Word
...	...
00000010	<b>a</b>
00000014	<b>b</b>
00000018	<b>c</b>
<u>0000001c</u>	<b>d</b>

Address	Value of Word
...	...
<u>00000030</u>	<b>e</b>
00000034	<b>f</b>
00000038	<b>g</b>
0000003c	<b>h</b>

Address	Value of Word
...	...
00008010	<b>i</b>
00008014	<b>j</b>
00008018	<b>k</b>
0000801c	<b>l</b>

...





## Pre-Exam Exercise #2

---

**We are now going to stop for ~5 minutes. During this time, your goal is to (by yourself) come up with a potential exam exercise covering the topic of Digital Logic, State Machines, and Combinational Logic Blocks. Make it as much like a real exam question as possible.**

**After this five minutes, you will explain your question to a small group and work through how you would go about solving it. I'll call on some random samples for the full class.**



# Administrivia

---

- **HW6 Due Tonight**
- **HW7 Out Today**
  
- **Midterm 2:**
  - **Friday, 11:00am – 2:00pm**
  - **306 Soda (HP Auditorium)**
  - **Conflicts, DSP, &&|| terrified about the drop deadline: Contact Andy ASAP**



# Big Endian vs. Little Endian

Big-endian and little-endian derive from Jonathan Swift's *Gulliver's Travels* in which the Big Endians were a political faction that broke their eggs at the large end ("the primitive way") and rebelled against the Lilliputian King who required his subjects (the Little Endians) to break their eggs at the small end.

- The order in which BYTES are stored in memory
- Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)

Consider the number 1025 as we normally write it:

**BYTE3**    **BYTE2**    **BYTE1**    **BYTE0**  
**00000000** **00000000** **00000100** **00000001**

## Big Endian

- ADDR3 ADDR2 ADDR1 ADDR0  
  **BYTE0** **BYTE1** **BYTE2** **BYTE3**  
  00000001 00000100 00000000 00000000
- ADDR0 ADDR1 ADDR2 ADDR3  
  **BYTE3** **BYTE2** **BYTE1** **BYTE0**  
  00000000 00000000 00000100 00000001

## Little Endian

- ADDR3 ADDR2 ADDR1 ADDR0  
  **BYTE3** **BYTE2** **BYTE1** **BYTE0**  
  00000000 00000000 00000100 00000001
- ADDR0 ADDR1 ADDR2 ADDR3  
  **BYTE0** **BYTE1** **BYTE2** **BYTE3**  
  00000001 00000100 00000000 00000000

[www.webopedia.com/TERM/b/big\\_endian.html](http://www.webopedia.com/TERM/b/big_endian.html)  
[searchnetworking.techtarget.com/sDefinition/0,,sid7\\_gci211659,00.html](http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211659,00.html)  
[www.noveltheory.com/TechPapers/endian.asp](http://www.noveltheory.com/TechPapers/endian.asp)  
[en.wikipedia.org/wiki/Big\\_endian](http://en.wikipedia.org/wiki/Big_endian)

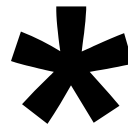
# Memorized this table yet?



• Blah blah **Cache size 16KB** blah blah  **$2^{23}$  blocks** blah blah how many bits?

• **Answer!  $2^{XY}$  means...**

$X=0 \Rightarrow$	no suffix		$Y=0 \Rightarrow$	1
$X=1 \Rightarrow$	kibi ~ Kilo	$10^3$	$Y=1 \Rightarrow$	2
$X=2 \Rightarrow$	mebi ~ Mega	$10^6$	$Y=2 \Rightarrow$	4
$X=3 \Rightarrow$	gibi ~ Giga	$10^9$	$Y=3 \Rightarrow$	8
$X=4 \Rightarrow$	tebi ~ Tera	$10^{12}$	$Y=4 \Rightarrow$	16
$X=5 \Rightarrow$	pebi ~ Peta	$10^{15}$	$Y=5 \Rightarrow$	32
$X=6 \Rightarrow$	exbi ~ Exa	$10^{18}$	$Y=6 \Rightarrow$	64
$X=7 \Rightarrow$	zebi ~ Zetta	$10^{21}$	$Y=7 \Rightarrow$	128
$X=8 \Rightarrow$	yobi ~ Yotta	$10^{24}$	$Y=8 \Rightarrow$	256
			$Y=9 \Rightarrow$	512



# How Much Information IS that?

[www.sims.berkeley.edu/research/projects/how-much-info-2003/](http://www.sims.berkeley.edu/research/projects/how-much-info-2003/)

- Print, film, magnetic, and optical storage media produced about **5 exabytes** of new information in 2002. 92% of the new information stored on magnetic media, mostly in hard disks.
- Amt of new information stored on paper, film, magnetic, & optical media **~doubled in last 3 yrs**
- Information flows through electronic channels -- telephone, radio, TV, and the Internet -- contained **~18 exabytes** of new information in 2002, 3.5x more than is recorded in storage media. **98% of this total is the information sent & received in telephone calls** - incl. voice & data on fixed lines & wireless.
  - WWW  $\Rightarrow$  170 Tb of information on its surface; in volume 17x the size of the Lib. of Congress print collections.
  - *Instant messaging*  $\Rightarrow$   $5 \times 10^9$  msgs/day (750GB), 274 TB/yr.
  - *Email*  $\Rightarrow$  ~400 PB of new information/year worldwide.



# Block Size Tradeoff (1/3)

---

- **Benefits of Larger Block Size**
  - **Spatial Locality**: if we access a given word, we're likely to access other nearby words soon
  - **Very applicable with Stored-Program Concept**: if we execute a given instruction, it's likely that we'll execute the next few as well
  - **Works nicely in sequential array accesses too**



## Block Size Tradeoff (2/3)

---

- Drawbacks of Larger Block Size
  - Larger block size means **larger miss penalty**
    - on a miss, takes longer time to load a new block from next level
  - If block size is too big relative to cache size, then there are too few blocks
    - Result: miss rate goes up
- In general, minimize **Average Memory Access Time (AMAT)**
  - = Hit Time
  - + Miss Penalty x Miss Rate



## Block Size Tradeoff (3/3)

---

- **Hit Time** = time to find and retrieve data from current level cache
- **Miss Penalty** = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
- **Hit Rate** = % of requests that are found in current level cache
- **Miss Rate** =  $1 - \text{Hit Rate}$





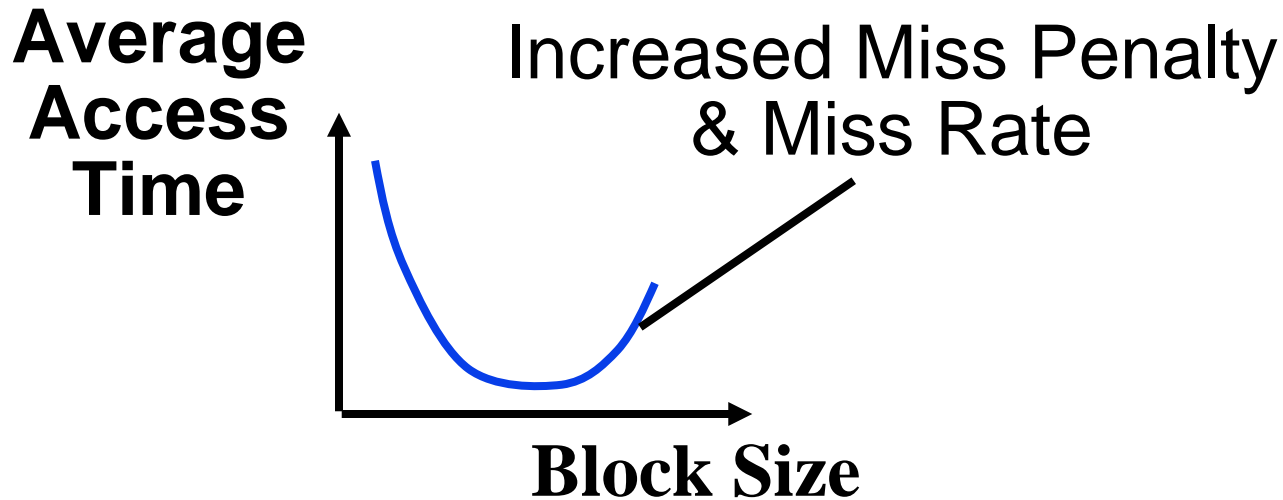
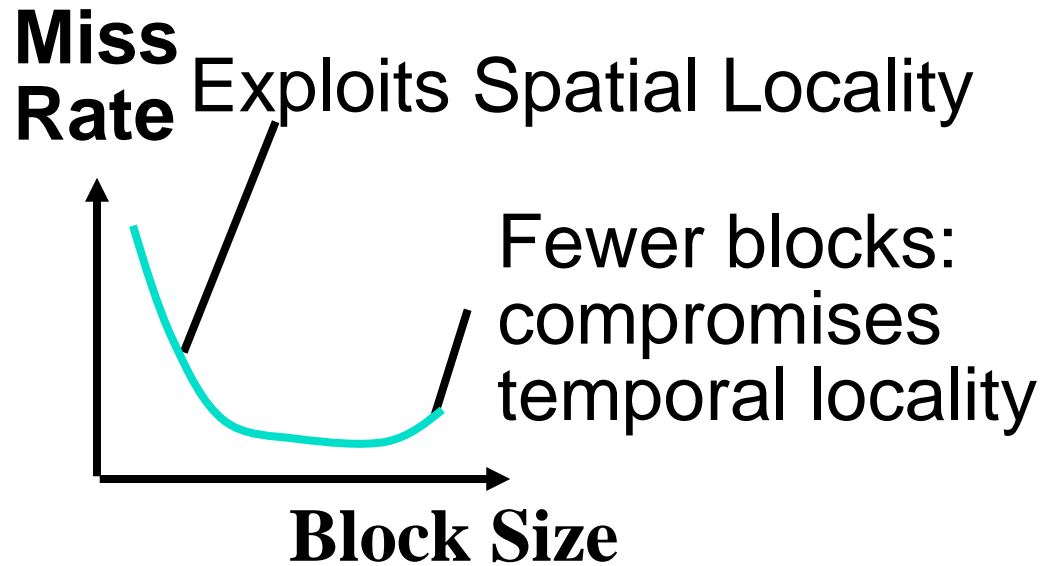
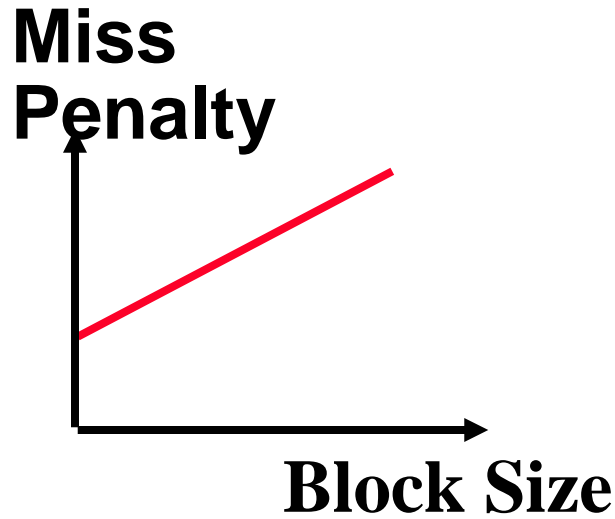
## Extreme Example: One Big Block



- Cache Size = 4 bytes    Block Size = 4 bytes
  - Only **ONE** entry in the cache!
- If item accessed, likely accessed again soon
  - But unlikely will be accessed again immediately!
- The next access will likely to be a miss again
  - Continually loading data into the cache but discard data (force out) before use it again
  - Nightmare for cache designer: **Ping Pong Effect**



# Block Size Tradeoff Conclusions



# Peer Instructions

---

1. **All caches take advantage of spatial locality.**
2. **All caches take advantage of temporal locality.**
3. **On a read, the return value will depend on what is in the cache.**



# Peer Instruction Answer

---

1. All caches take advantage of spatial locality. **FALSE**
2. All caches take advantage of temporal locality. **TRUE**
3. On a read, the return value will depend on what is in the cache. **FALSE**

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

1. Block size = 1, no spatial!
2. That's the idea of caches; We'll need it again soon.
3. It better not! If it's there, use it. Or, get from mem



# And in Conclusion...

- Mechanism for transparent movement of data among levels of a storage hierarchy
  - set of address/value bindings
  - address  $\Rightarrow$  index to set of candidates
  - compare desired address with tag
  - service hit or miss
    - load new block and binding on miss

