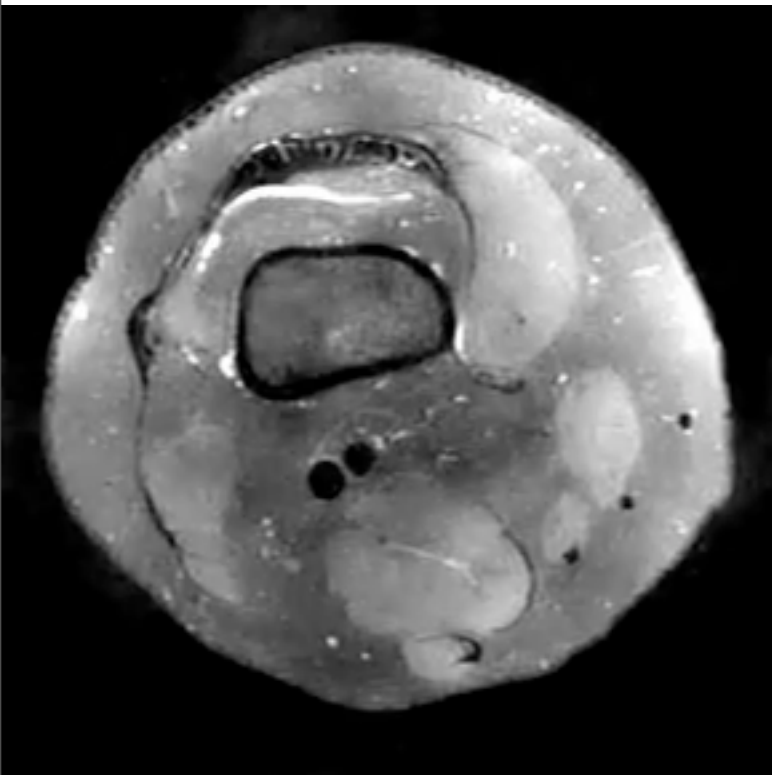


inst.eecs.berkeley.edu/~cs61c
CS61C : Machine Structures

Lecture 13 – Caches II

2014-09-29

**Instructor:
Miki Lustig**



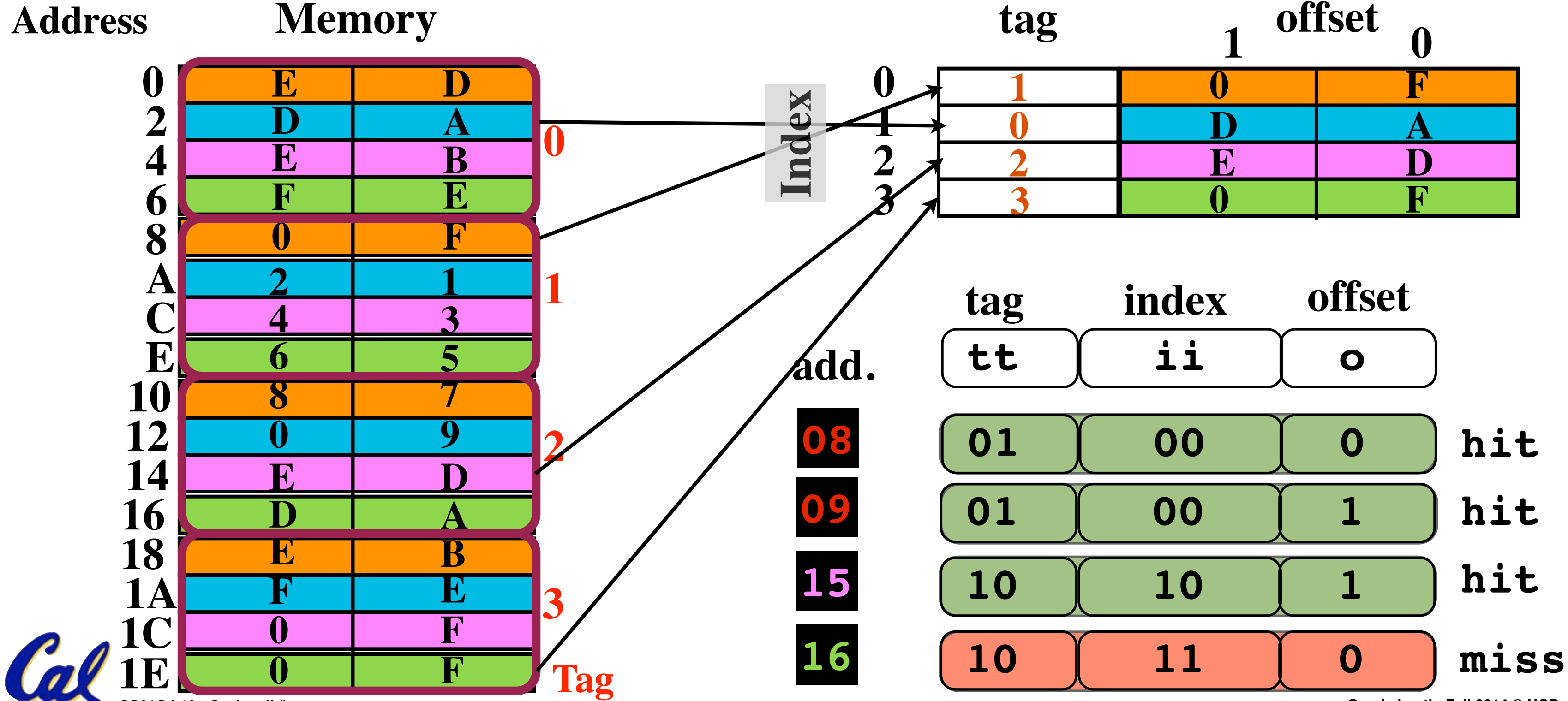
August 2014: IBM Unveils a ‘Brain-Like’ Chip With 4,000 Processor Cores

TrueNorth comes packed with 4,096 processor cores, and it mimics one million human neurons and 256 million synapses. The chip is meant to run neural-net processing for recognition.



Review: Direct-Mapped Cache

5bit Memory



Review: TIO Dan's great cache mnemonic

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

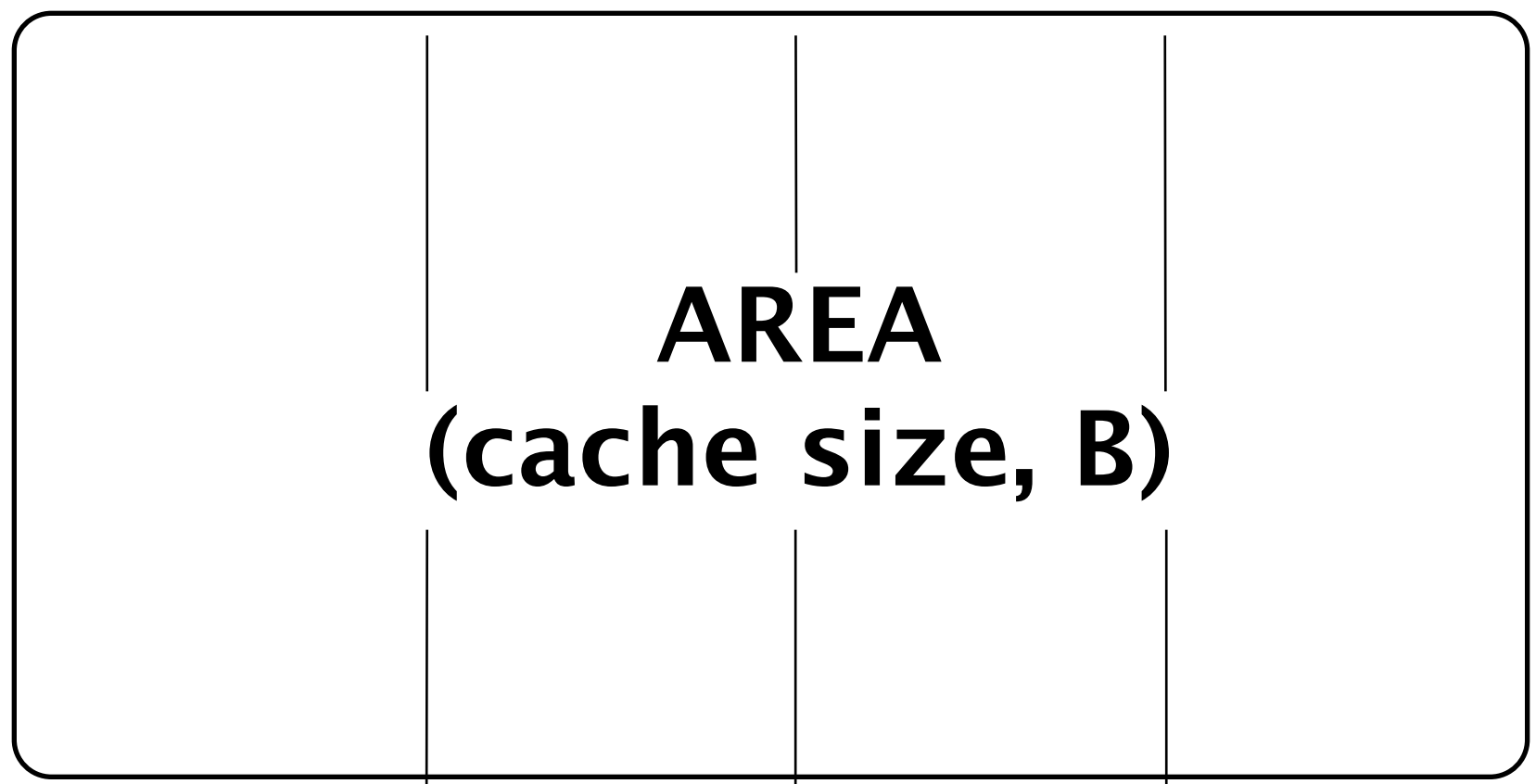
$$2^{(H+W)} = 2^H * 2^W$$



WIDTH
 (size of one block, B/block)

tag	1	offset	0
1	0	F	
0	D	A	
2	E	D	
3	0	F	

HEIGHT
 (# of blocks)

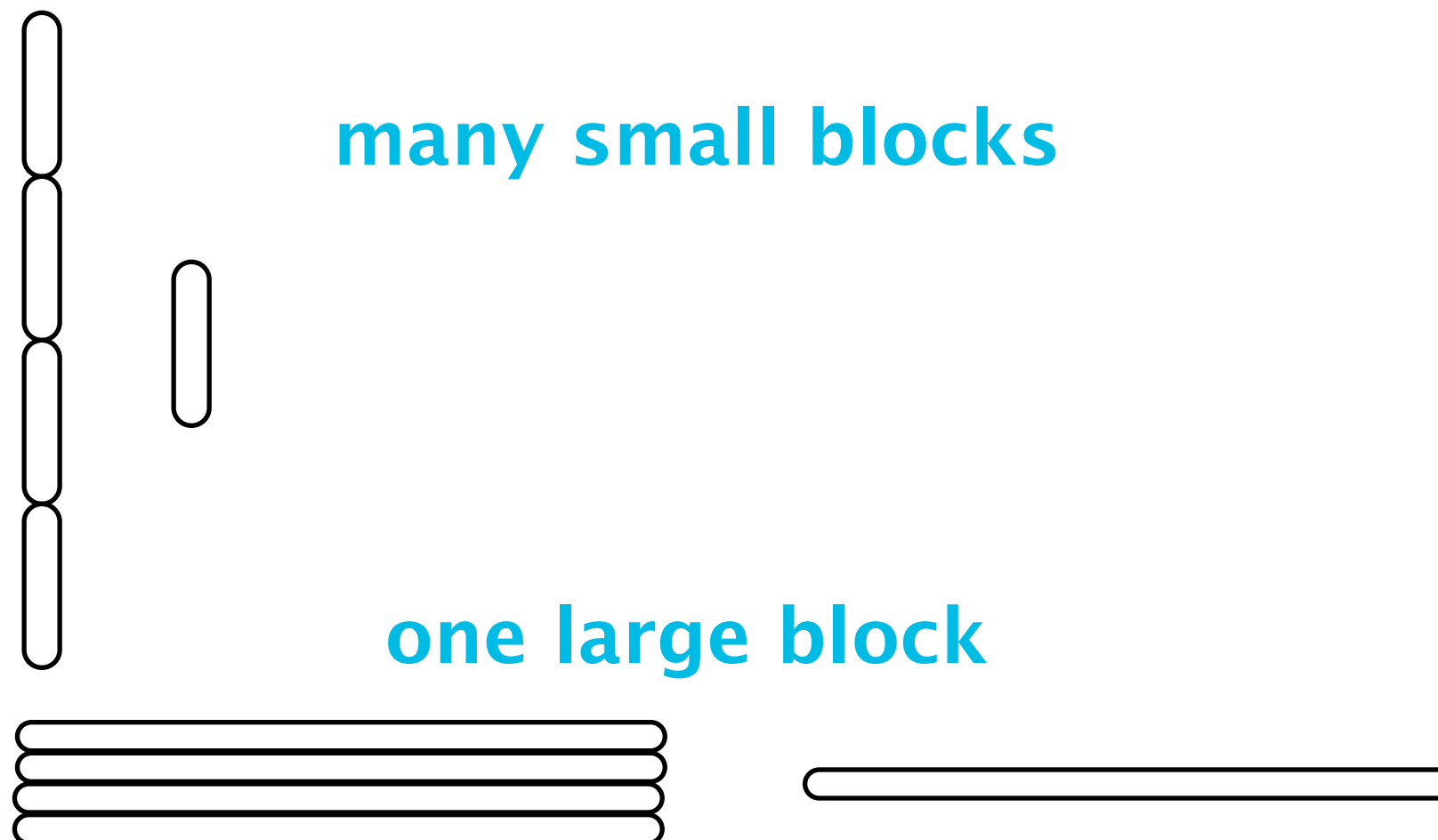


How to Split Cache?

- **Small or large block sizes?**

- **Large**

- **Better spatial locality**
- **Too large, misses increase with large penalty**



Memory Access without Cache

- Load word instruction: `lw $t0, 0($t1)`
- `$t1` contains 1022_{ten} , `Memory[1022] = 99`

1. Processor issues address 1022_{ten} to Memory
2. Memory reads word at address 1022_{ten} (99)
3. Memory sends 99 to Processor
4. Processor loads 99 into register `$t1`



Memory Access with Cache

- **Load word instruction: `lw $t0, 0($t1)`**
- **`$t1` contains 1022_{ten} , `Memory[1022] = 99`**
- **With cache (similar to a hash)**
 - 1. Processor issues address 1022_{ten} to Cache**
 - 2. Cache checks to see if has copy of data at address 1022_{ten}**
 - 2a. If finds a match (Hit): cache reads 99, sends to processor**
 - 2b. No match (Miss): cache sends address 1022 to Memory**
 - I. Memory reads 99 at address 1022_{ten}**
 - II. Memory sends 99 to Cache**
 - III. Cache replaces word with new 99**
 - IV. Cache sends 99 to processor**
 - 3. Processor loads 99 into register `$t1`**



Caching Terminology

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



Cache Terms

- **Hit rate**: fraction of access that hit in the cache [0.0–1.0]
- **Miss rate**: $1 - \text{Hit rate}$
- **Miss penalty**: time to replace a block from lower level in memory hierarchy to cache
- **Hit time**: time to access cache memory (including tag comparison)

- **Abbreviation**: “\$” = cache



Accessing data in a direct mapped cache

- Ex.: 16KiB of data, direct-mapped, 4 word blocks
- Can you work out height, width, area?

- Read 4 addresses

1. $0x00000014$

2. $0x0000001c$

3. $0x00000034$

4. $0x00008014$

- Memory vals here:

Address (hex)	Memory 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

00000000000000000000000000000000	000000000001	0100
00000000000000000000000000000000	000000000001	1100
00000000000000000000000000000000	000000000011	0100
00000000000000000000000000000010	000000000001	0100

Tag

Index

Offset



16 KiB Direct Mapped Cache, 16B blocks

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

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

1. Read 0x00000014

• 000000000000000000000000
Tag

0000000001
Index Field

0100
offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
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)

• 000000000000000000000000

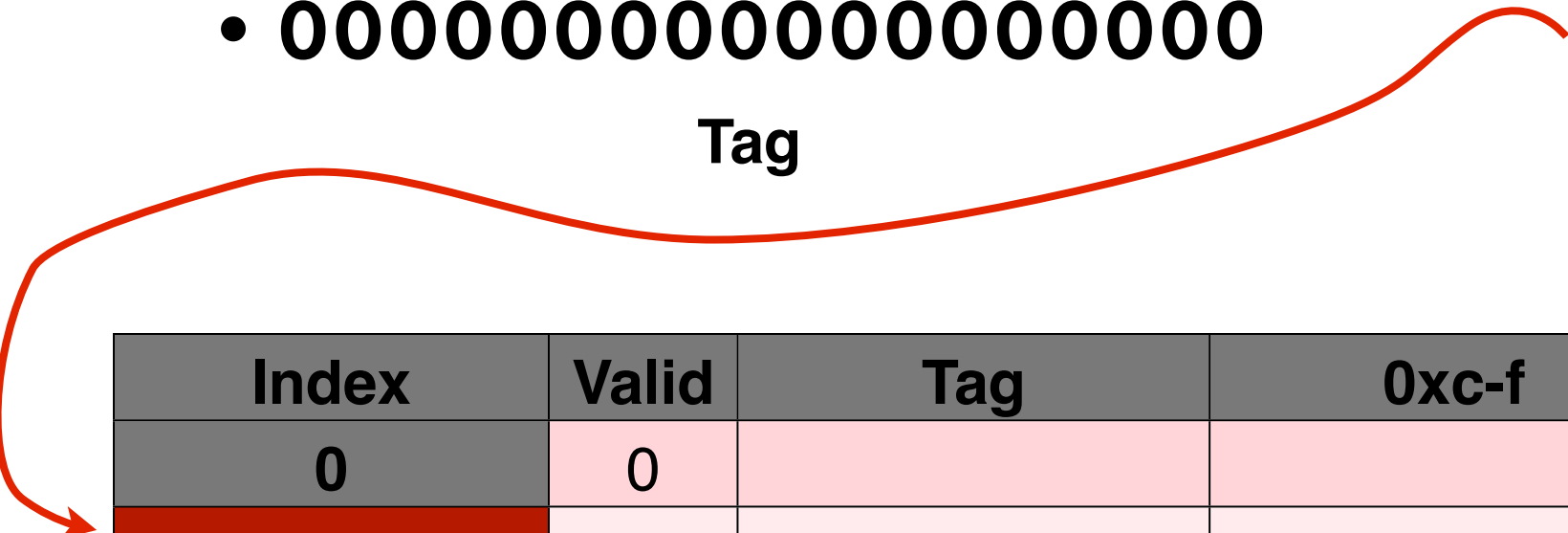
Tag

0000000001

Index Field

0100

offset



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



No valid data

• 000000000000000000000000

Tag

0000000001

Index Field

0100

offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	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

• 000000000000000000000000

Tag

0000000001

Index Field

0100

offset

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



Index is Valid

• 000000000000000000000000

Tag

0000000001

Index Field

1100

offset

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



Index Valid, Tag Matches, return d

• 000000000000000000000000

Tag

00000000001

Index Field

1100

offset

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



So read block 3

• 000000000000000000000000

Tag

0000000011

Index Field

0100

offset

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



No valid data

• 000000000000000000000000

Tag

0000000011

Index Field

0100

offset

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



So read Cache Block 1, Data is Valid

• 0000000000000000000010

Tag

0000000001

Index Field

0100

offset

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



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

• 0000000000000000000010

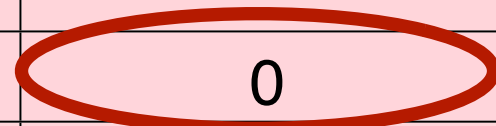
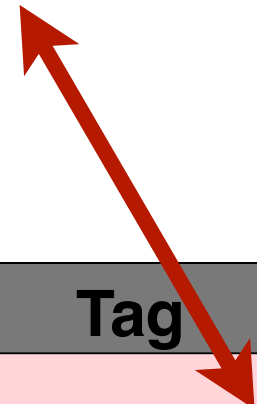
Tag

0000000001

Index Field

0100

offset



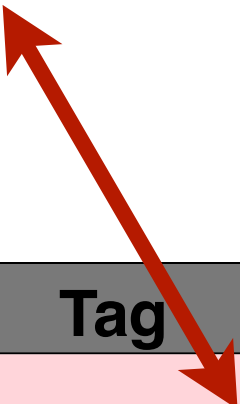
Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...
1022	0					
1023	0					



Miss, so replace block 1 with new data & tag

• 0000000000000000000010

Tag



0000000001

Index Field

0100

offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...
1022	0					
1023	0					



And return word J

• 00000000000000000010
Tag

0000000001
Index Field

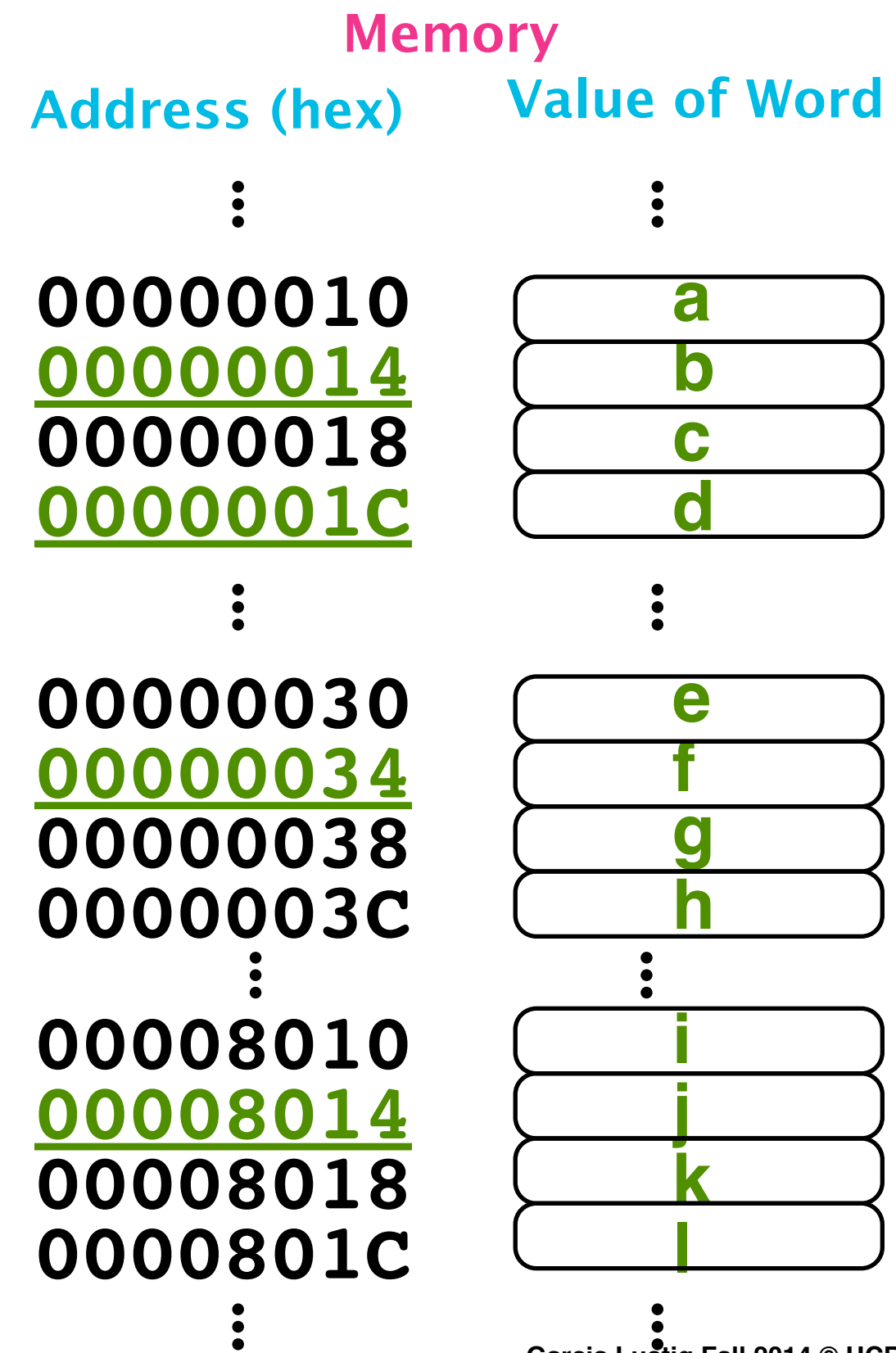
0100
offset

Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...
1022	0					
1023	0					



Do an example yourself. What happens?

- Chose from: Cache: Hit, Miss, Miss w. replace
Values returned: a ,b, c, d, e, ..., k, l
- Read address **0x00000030** ?
00000000000000000000 0000000011 0000
- Read address **0x0000001c** ?
00000000000000000000 0000000001 1100



Index	Valid	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...
1022	0					
1023	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

Address (hex)	Memory 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
⋮	⋮

Peer Instruction

- 1) Mem hierarchies **were invented before 1950.** (UNIVAC I wasn't delivered 'til 1951)
- 2) **All caches take advantage of spatial locality.**
- 3) **All caches take advantage of temporal locality.**

	1	2	3
a)	F	F	F
a)	F	F	T
b)	F	T	F
b)	F	T	T
c)	T	F	F
d)	T	F	T
e)	T	T	F
e)	T	T	T



Peer Instruction

- A. For a given cache size: a larger block size can cause a lower hit rate than a smaller one.
- B. If you know your computer's cache size, you can often **make your code run faster**.
- C. Memory hierarchies take advantage of **spatial locality** by keeping the most recent data items **closer** to the processor.

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



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

