



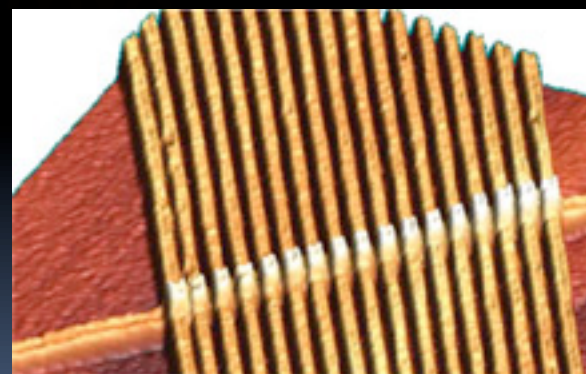
Lecturer SOE  
Dan Garcia

## Lecture 13 – Caches II

2013-02-22

### MEMRISTOR MEMORY ON ITS WAY...

HP has begun testing research prototypes of a novel non-volatile memory element, the memristor. They have double the storage density of flash, and has 10x more read-write cycles than flash ( $10^6$  vs  $10^5$ ). Memristors described in Nature are also capable of being memory and logic, how cool is that?



[www.technologyreview.com/computing/25018](http://www.technologyreview.com/computing/25018)

# Review: New-School Machine Structures

*Software*

*Hardware*

- Parallel Requests  
Assigned to computer  
e.g., Search “Katz”

- Parallel Threads  
Assigned to core  
e.g., Lookup, Ads

- Parallel Instructions  
>1 instruction @ one time  
e.g., 5 pipelined instructions

- Parallel Data  
>1 data item @ one time  
e.g., Add of 4 pairs of words

- Hardware descriptions  
All gates @ one time

- Programming Languages

2/21/13

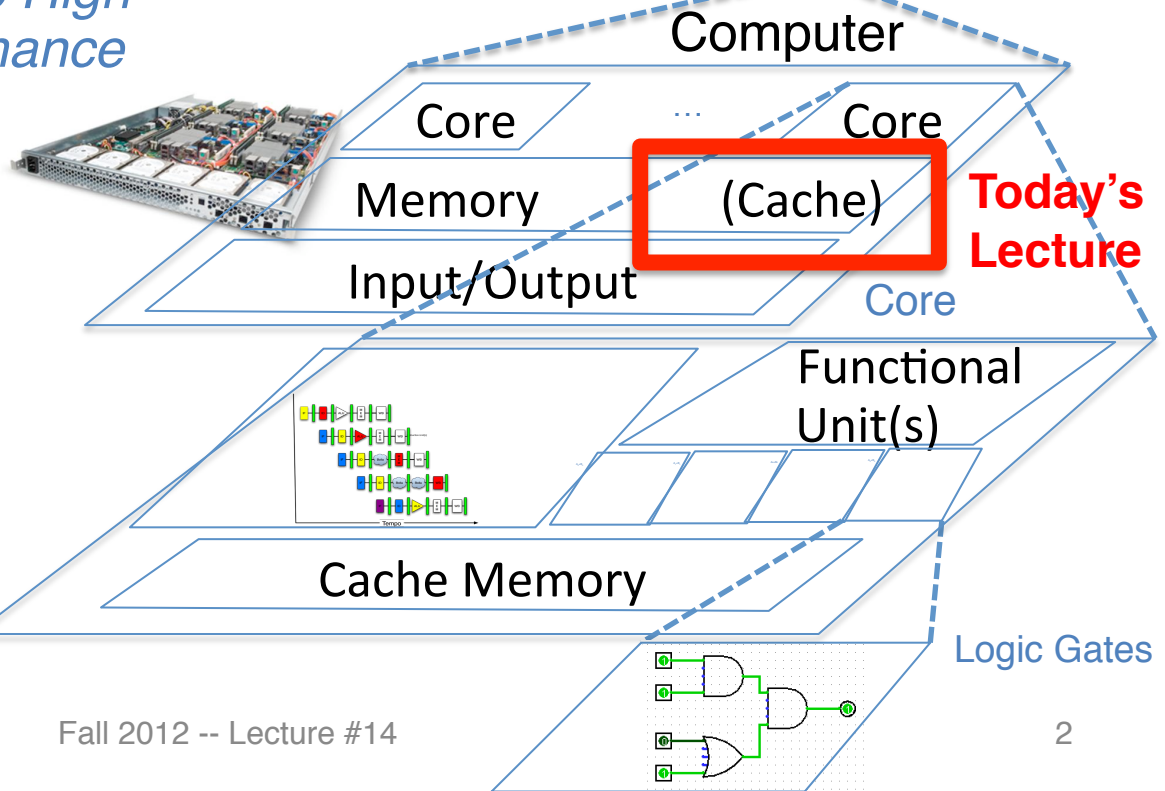
Warehouse  
Scale  
Computer



Smart  
Phone



*Harness  
Parallelism &  
Achieve High  
Performance*



Fall 2012 -- Lecture #14

2



# TIO Dan's great cache mnemonic

AREA (cache size, B)

= HEIGHT (# of blocks)

\* WIDTH (size of one block, B/block)

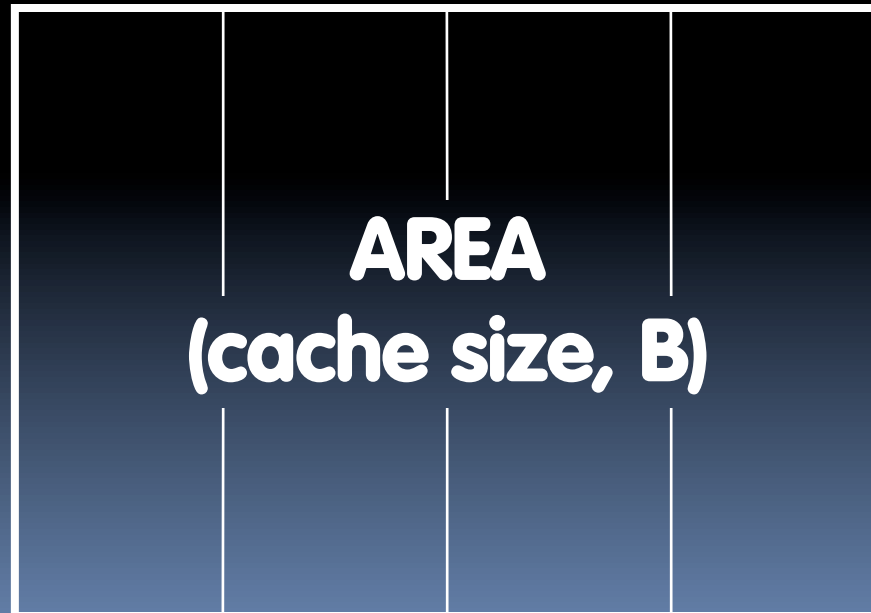
$$2^{(H+W)} = 2^H * 2^W$$



Addr size  
(usu 32 bits)

HEIGHT  
(# of blocks)

WIDTH  
(size of one block, B/block)



# Memory Access without Cache

- **Load word instruction: `lw $t0, 0($t1)`**
- **`$t1` contains  $1022_{\text{ten}}$ , `Memory[1022] = 99`**
  1. **Processor issues address  $1022_{\text{ten}}$  to Memory**
  2. **Memory reads word at address  $1022_{\text{ten}}$  (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  $1022_{\text{ten}}$ , `Memory[1022] = 99`
- With cache (similar to a hash)
  1. Processor issues address  $1022_{\text{ten}}$  to Cache
  2. Cache checks to see if has copy of data at address  $1022_{\text{ten}}$ 
    - 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  $1022_{\text{ten}}$
      - 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
- **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 (A Berkeley innovation!)





# Accessing data in a direct mapped cache

- Ex.: 16KB 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:

**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
<b>Tag</b>	<b>Index</b>	<b>Offset</b>



# 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	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 0000000001 0100  
 Tag field Index field Offset

Valid

Index	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 0000000001 0100  
 Tag field Index field Offset

Valid

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3
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	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 00000000001 0100  
Tag field Index field Offset

Valid	Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3
	0	0				
	1	0	d	c	b	a
	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 00000000001 0100  
 Tag field Index field Offset

Valid

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

...

...

1022	0				
1023	0				





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

- 000000000000000000000000 0000000001 1100  
                                   Tag field                                  Index field                  Offset

Valid

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0				
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 000000000001 1100  
 Tag field Index field Offset

Valid

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

...

...

1022	0				
1023	0				



# Index valid, Tag Matches

- 000000000000000000000000
000000000000000000000000
1
1100  
Tag field
Index field
Offset

Valid	Index	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

▪ 00000000000000000000000000000000 000000000001 1100  
 Tag field Index field Offset

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

...

1022	0					
1023	0					



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

- 000000000000000000000000 0000000011 0100  
 Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0				
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 0000000011 0100  
 Valid Tag field Index field Offset

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

...

...

1022	0				
1023	0				





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

- 0000000000000000000010 0000000001 0100  
 Valid Tag field Index field Offset

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

...

...

1022	0				
1023	0				



# So read Cache Block 1, Data is Valid

0000000000000000000010 0000000001 0100  
 Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	0x4-7	0x0-3
0	0				
<u>1</u>	0	d	c	b	a
2	0				
3	1	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 00000000001 0100  
 Valid Tag field Index field Offset

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

- 000000000000000000000010 000000000001 0100  
 Valid Tag field Index field Offset

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

...

...

1022	0				
1023	0				



# And return word J

- 0000000000000000000010 0000000001 0100  
 Valid Tag field Index field Offset

Index	Tag	0xc-f	0x8-b	<u>0x4-7</u>	0x0-3
0	0				
1	2	l	k	j	i
2	0				
3	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

## Cache

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					

...



# 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)	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
...	...
00080010	i
<u>00080014</u>	j
00080018	k
0008001c	l
...	...



# Administrivia

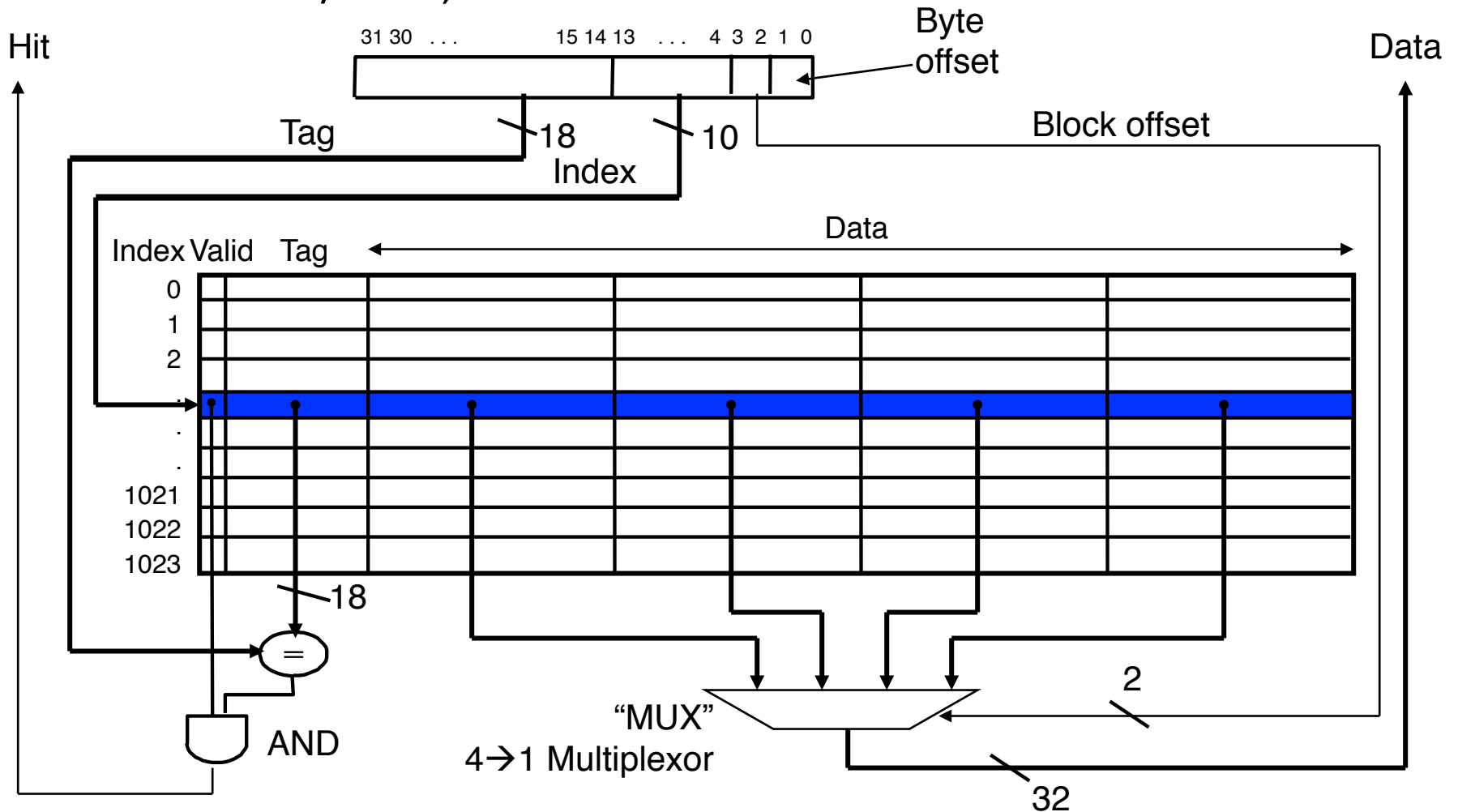
- **Midterm in 10 days**
  - **Taking it will be complicated, involving 4 rooms.**
  - **We'll assign you by last name to your room.**
- **Please watch all last week's videos by Monday so I can ask you about them**





# Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 4K words



*What kind of locality are we taking advantage of?*

# 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 Answer

1) "We are...forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less accessible." – von Neumann, 1946

2) Block size = 1, no spatial!

3) That's the idea of caches;  
We'll need it again soon.

1) Mem hierarchies were invented before 1950. UNIVAC I wasn't delivered 'til 1951)  
**TRUE**

2) All caches take advantage of spatial locality.  
**FALSE**

3) All caches take advantage of temporal locality.  
**TRUE**

- |    | 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 |



# 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

address: tag index offset  
000000000000000000000000 000000000001 1100

Valid

	Tag	<span style="color: red;">0xc-f</span>	<span style="color: red;">0x8-b</span>	<span style="color: red;">0x4-7</span>	<span style="color: red;">0x0-3</span>
0					
1	0	d	c	b	a
2					
3					

