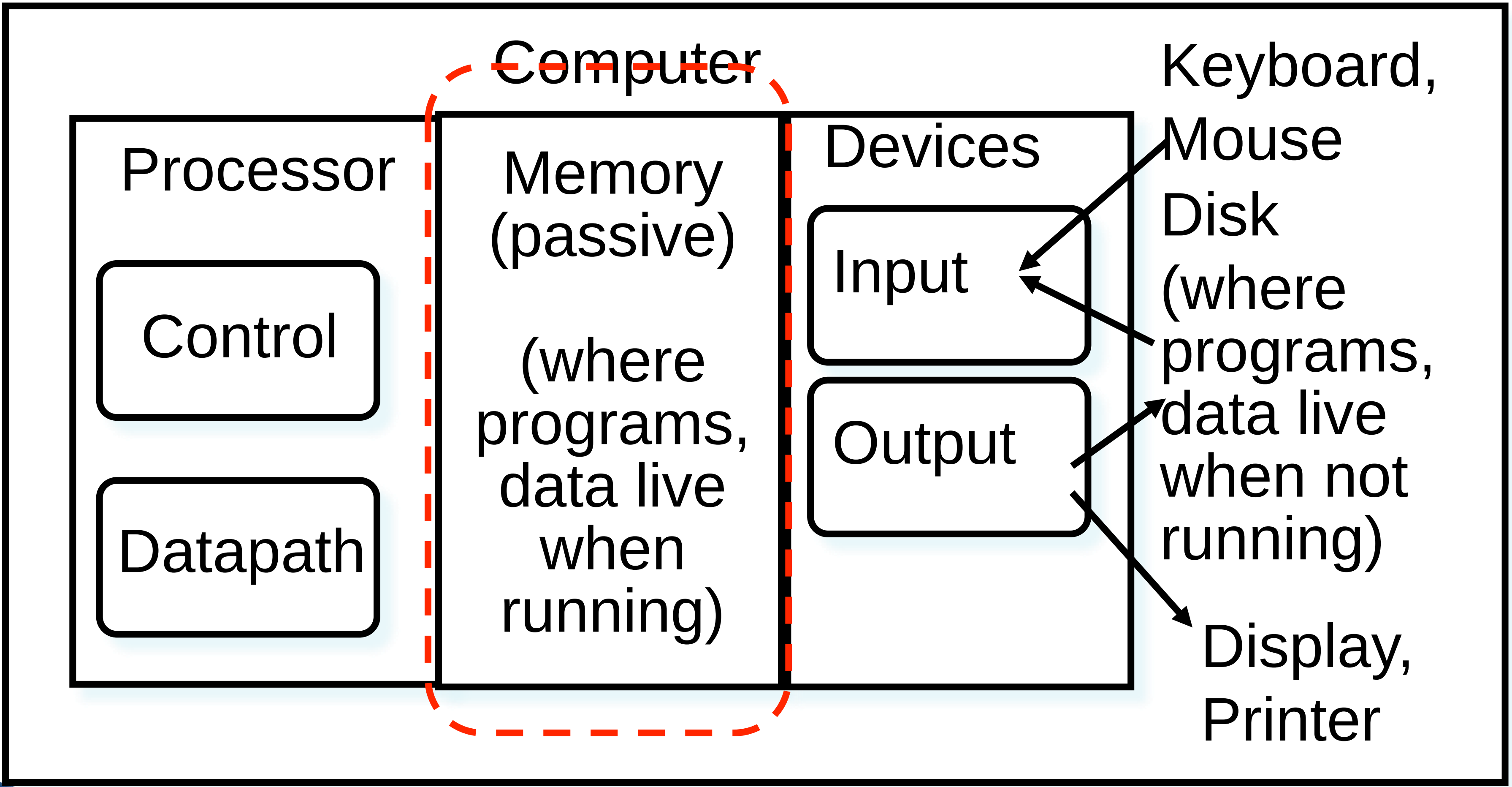
CSE 140
Computer Architecture

Lecture 6 – Memory (1)

Announcement

- Lab #2 this week
  - Due in one week
- Project #1
  - Due 10/11 (Monday) at 11:59pm
    - Start as soon as possible
- Reading assignment #3
  - Chapter 5.1 – 5.4
    - Do all Participation Activities in each section
    - Access through CatCourses
    - Due Thursday (9/19) at 11:59pm

Five Components of a Computer



Storage in Computer System

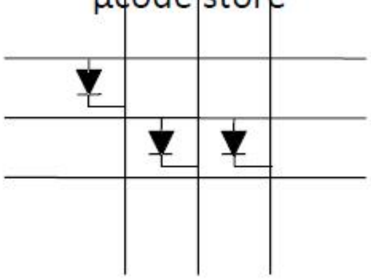
- Processor
  - Holds data in register file (~100 Bytes)
  - Registers accessed on nanosecond timescale
- Memory (we'll call it "main memory")
  - More capacity than registers (~Gbytes)
  - Access time ~50-100 ns
  - Hundreds of clock cycles per memory access?!
- Disk
  - HUGE capacity (virtually limitless)
  - VERY slow: runs ~milliseconds
    - Samsung 960 Pro NVME M.2 ($1,299, 2T)
      - 3,500MB/s ->1.143 ms

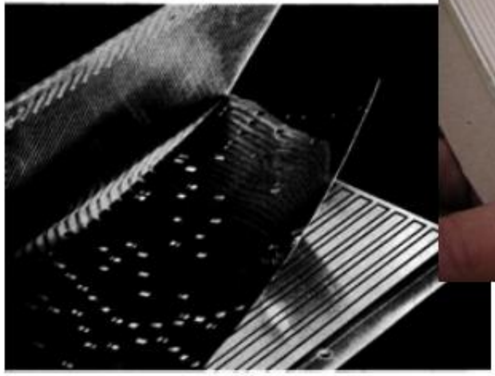
Early Read-Only Memory Technologies

Punched cards, From early 1700s through Jaquard Loom, Babbage, and then IBM

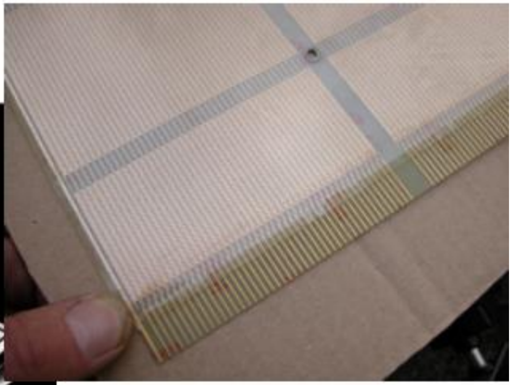
Punched paper tape, instruction stream in Harvard Mk 1

Diode Matrix, EDSAC-2 µcode store





IBM Card Capacitor ROS



IBM Balanced Capacitor ROS

ROS – Read-only Storage

Early Read/Write Main Memory Technologies

Babbage, 1800s: Digits stored on mechanical wheels

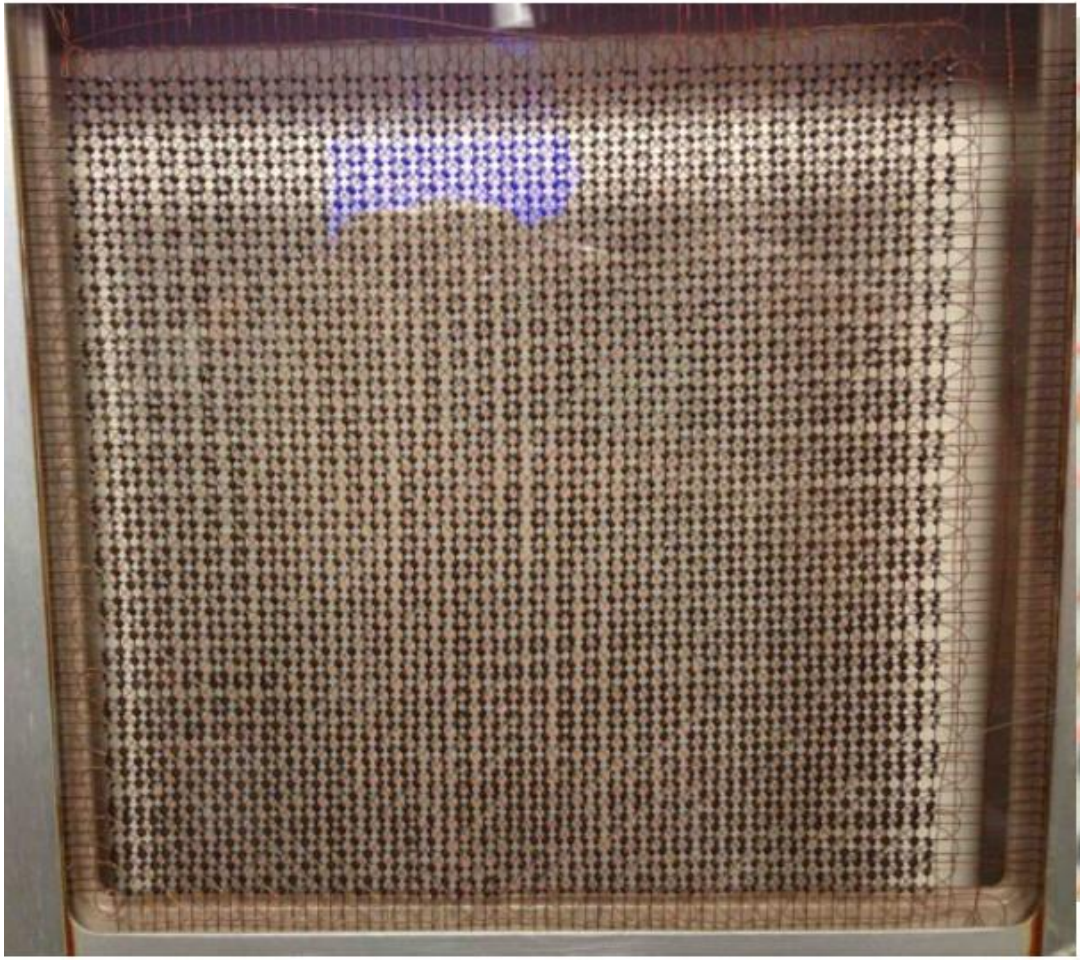
Also, regenerative capacitor memory on Atanasoff-Berry computer, and rotating magnetic drum memory on IBM 650

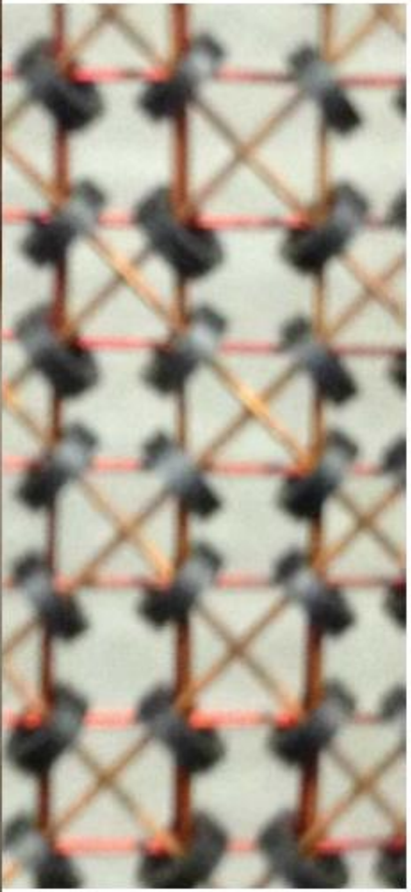
Williams Tube, Manchester Mark 1, 1947

Mercury Delay Line, Univac 1, 1951

~1,000 words

MIT Whirlwind Core Memory





Core Memory

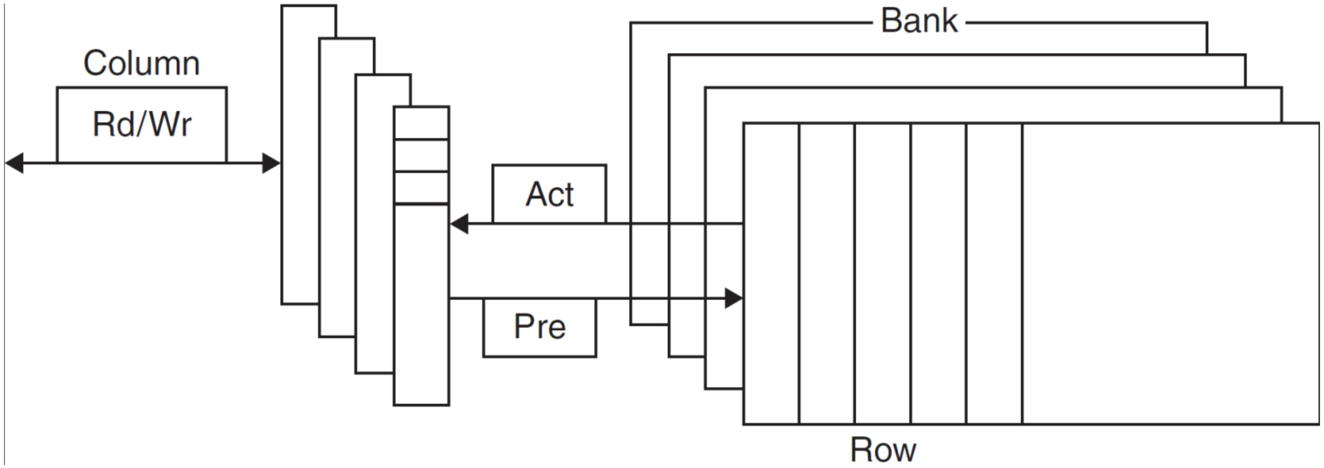
- Core memory was first large scale reliable main memory
  - invented by Forrester in late 40s/early 50s at MIT for Whirlwind project
- Bits stored as magnetization polarity on small ferrite cores threaded onto two-dimensional grid of wires
- Coincident current pulses on X and Y wires would write cell and also sense original state (destructive reads)
- Robust, non-volatile storage
- Used on space shuttle computers
- Cores threaded onto wires by hand (25 billion a year at peak production)
- Core access time ~ 1μs

Semiconductor Memory

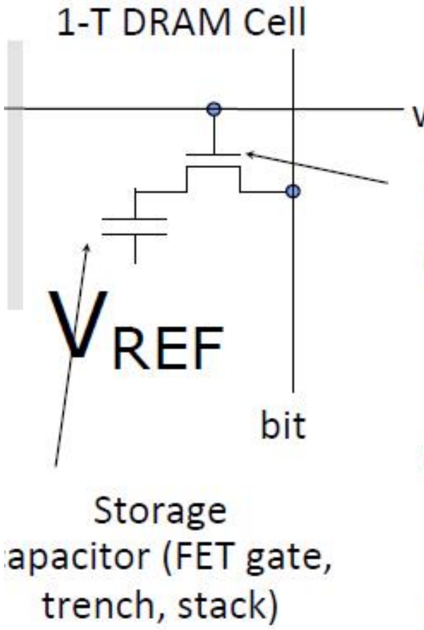
- Semiconductor memory began to be competitive in early 1970s
  - Intel formed to exploit market for semiconductor memory
  - Early semiconductor memory was Static RAM (SRAM). SRAM cell internals similar to a latch (flip-flop).
- First commercial Dynamic RAM (DRAM) was Intel 1103
  - 1Kbit of storage on single chip
  - Charge on a capacitor used to hold value
- Semiconductor memory quickly replaced core in ‘70s

DRAM Technology

- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM "row"

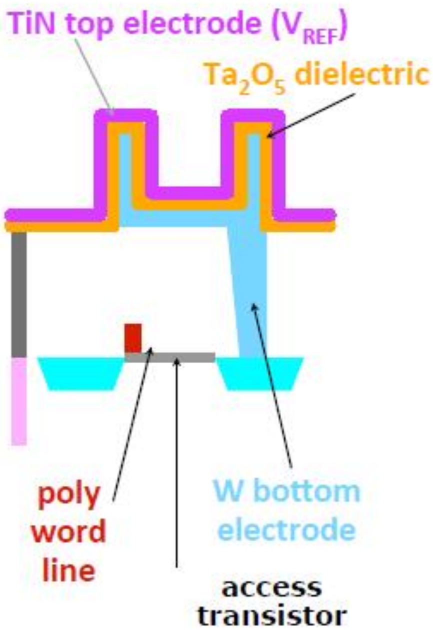


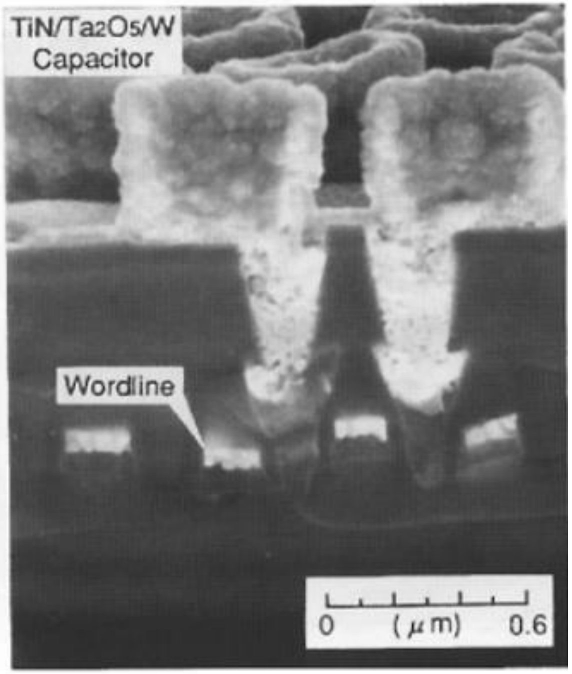
One-Transistor Dynamic RAM [Dennard, IBM]



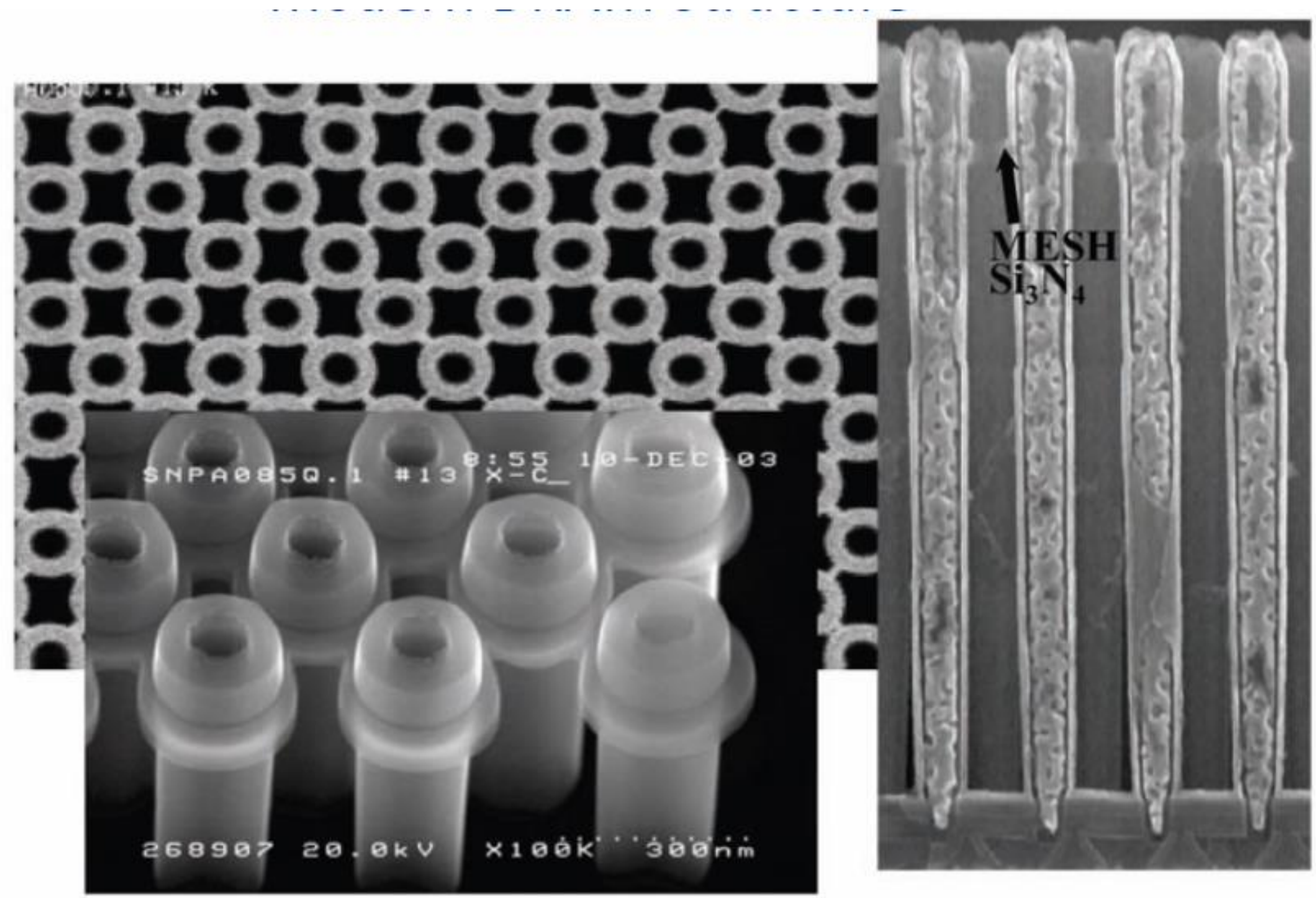
word

access transistor



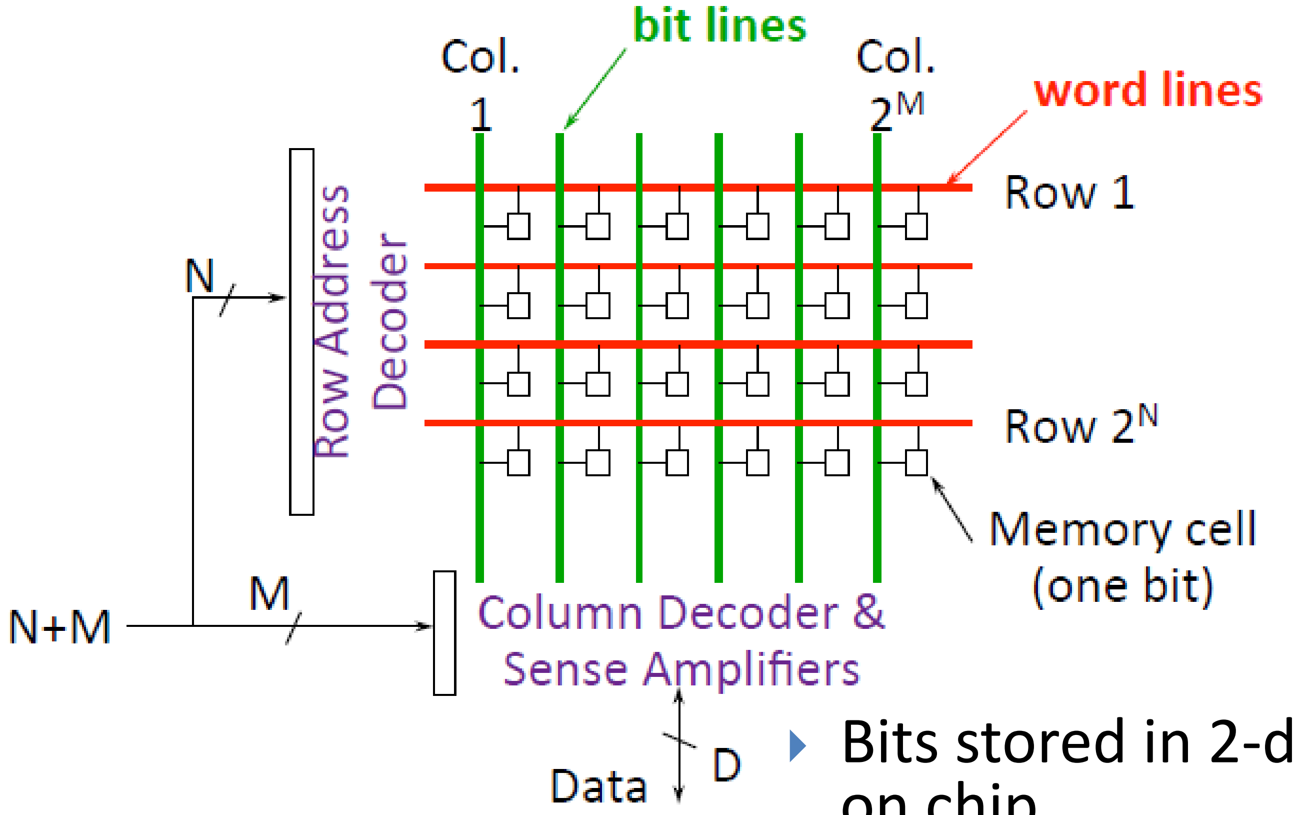


Modern DRAM Structure



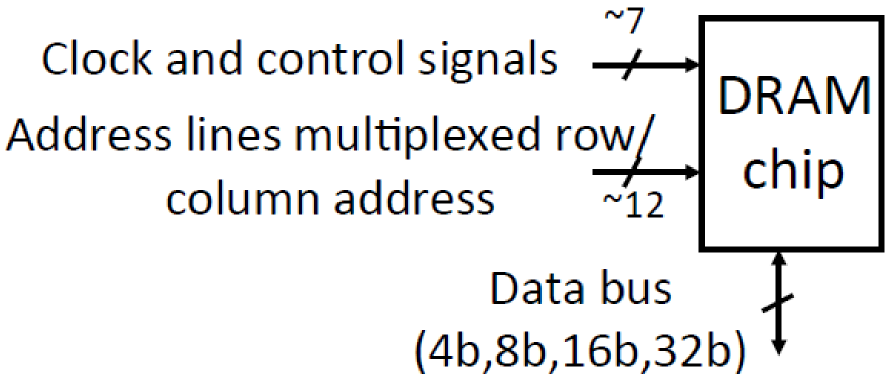
[Samsung, sub-70nm DRAM, 2004]

DRAM Architecture

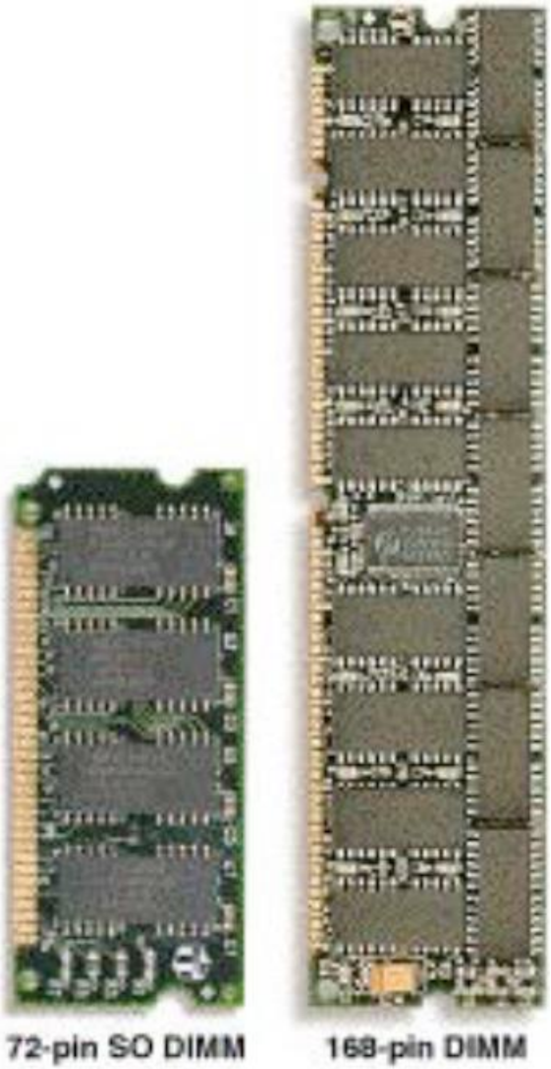


- Bits stored in 2-dimensional arrays on chip
- Modern chips have around 4-8 logical banks on each chip
  - Each logical bank physically implemented as many smaller arrays

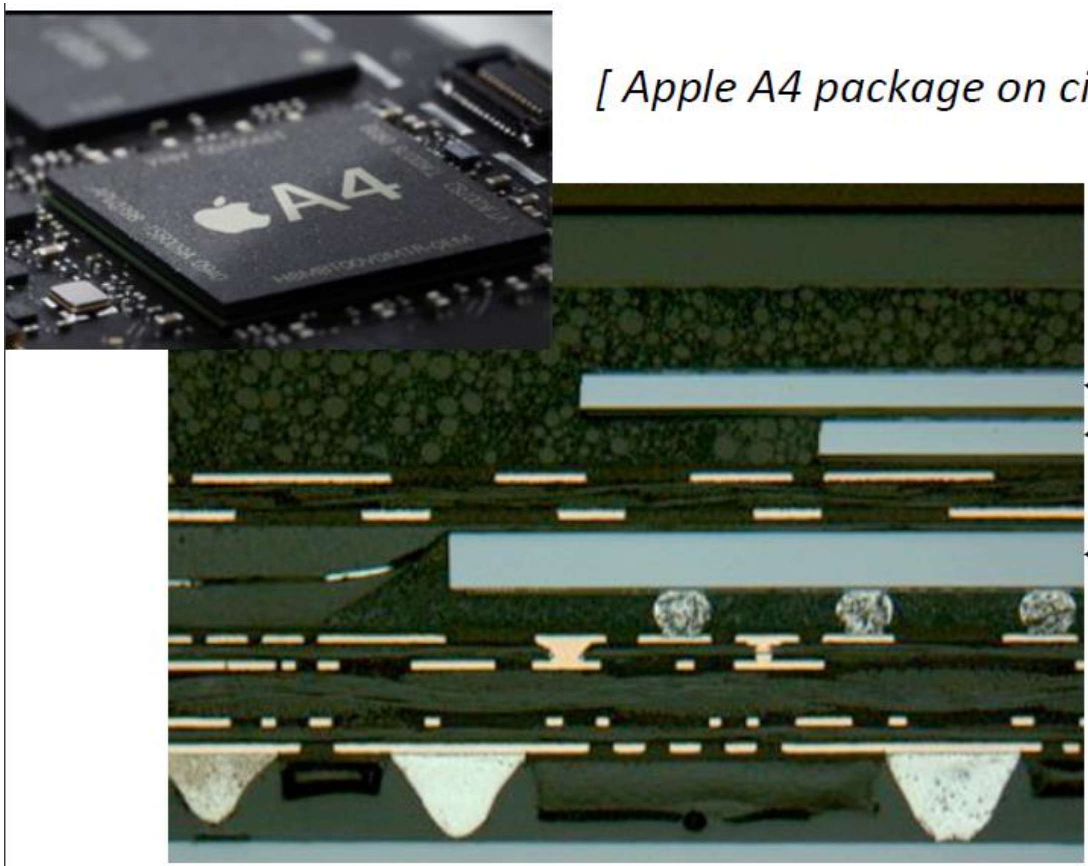
DRAM Packaging (Laptops/Desktops/Servers)


Clock and control signals
~7
Address lines multiplexed row/ column address
~12
DRAM chip
Data bus (4b,8b,16b,32b)

- DIMM (Dual Inline Memory Module) contains multiple chips with clock/control/address signals connected in parallel (sometimes need buffers to drive signals to all chips)
- Data pins work together to return wide word (e.g., 64-bit data bus using 16x4-bit parts)



DRAM Packaging, Mobile Devices



[Apple A4 package on circuit board]

Two stacked DRAM die

Processor plus logic die

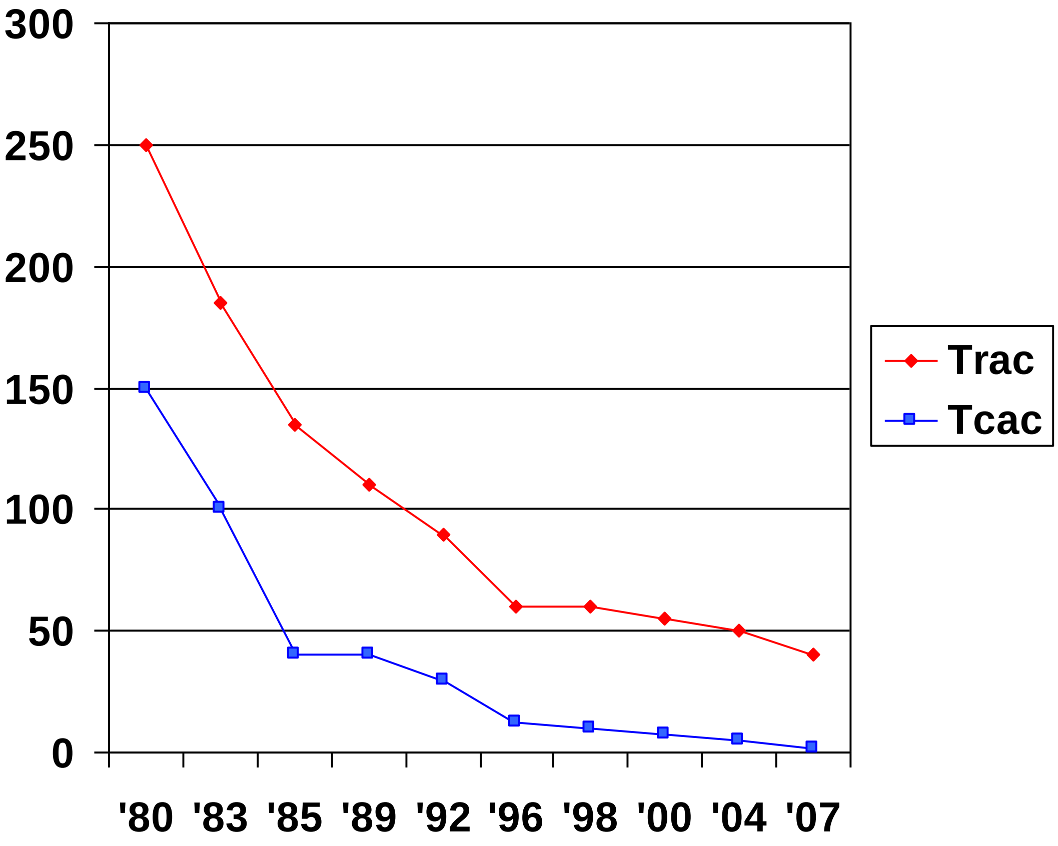
[Apple A4 package cross-section, iFixit 2010]

Advanced DRAM Organization

- Bits in a DRAM are organized as a rectangular array
  - DRAM accesses an entire row
  - Burst mode: supply successive words from a row with reduced latency
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
  - Separate DDR inputs and outputs

DRAM Generations

Year	Capacity	$/GB
1980	64Kbit	$1500000
1983	256Kbit	$500000
1985	1Mbit	$200000
1989	4Mbit	$50000
1992	16Mbit	$15000
1996	64Mbit	$10000
1998	128Mbit	$4000
2000	256Mbit	$1000
2004	512Mbit	$250
2007	1Gbit	$50



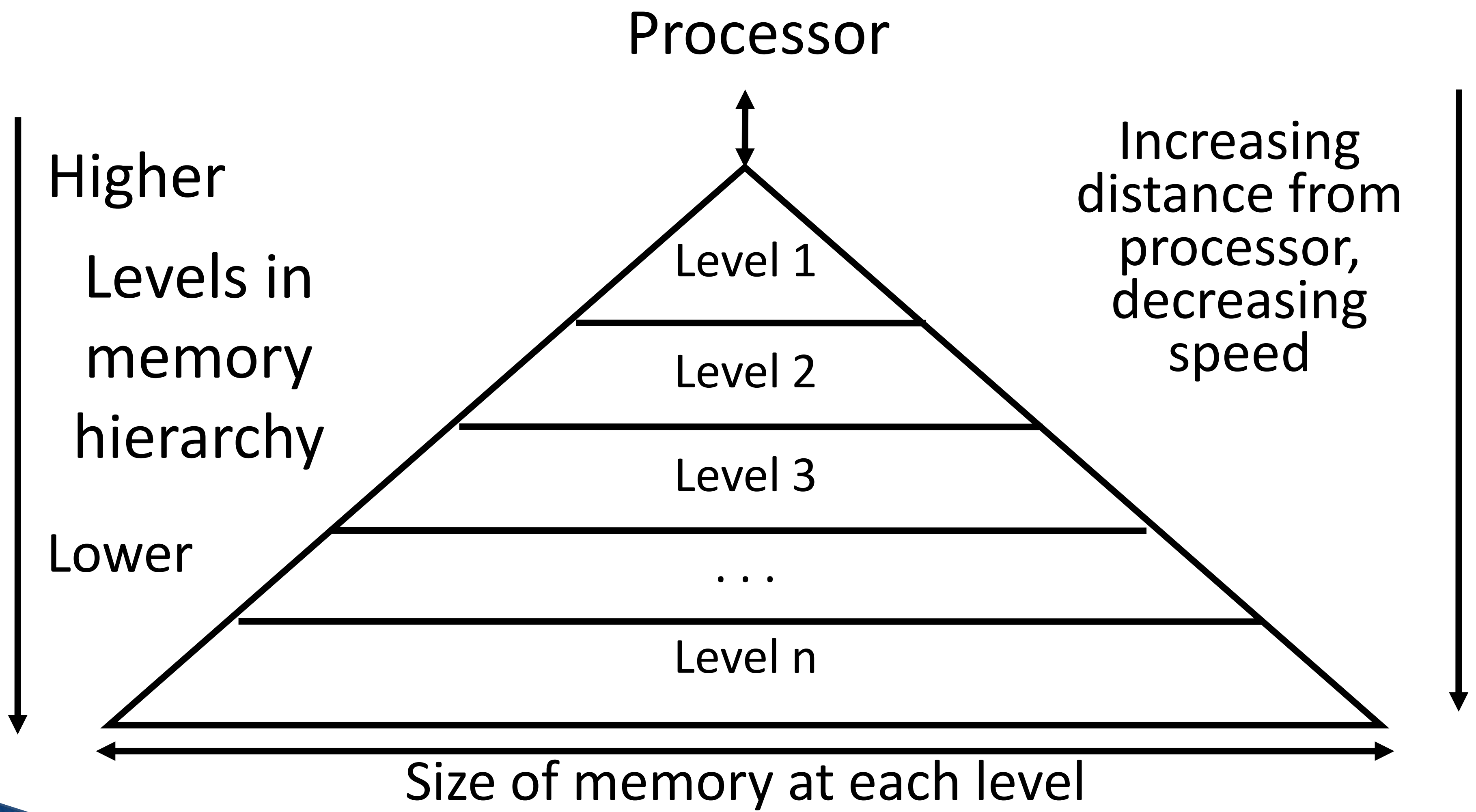
Trac: Random access time

Tcac: Column access time

Memory Caching

- Processor and memory speed mismatch leads us to add a new level: a memory cache
- Implemented with same integrated circuit processing technology as processor, integrated on-chip: faster but more expensive than DRAM memory
- Cache is a copy of a subset of main memory
- Modern processors have separate caches for instructions and data, as well as several levels of caches implemented in different sizes (why?)
  - Often use $ (“cash”) to abbreviate cache,
    e.g. D$ = Data Cache, I$ = Instruction Cache

Memory Hierarchy


Processor
Higher
Levels in memory hierarchy
Lower
Level 1
Level 2
Level 3
. . .
Level n
Increasing distance from processor, decreasing speed
Size of memory at each level

As we move to deeper levels the latency goes up and price per bit goes down. Why?

Memory Hierarchy

- If level closer to Processor, it is:
    - Smaller
    - Faster
    - More expensive
    - Subset of lower levels (contains most recently used data)
- Lowest Level (usually disk) contains all available data (does it go beyond the disk?)
- Memory Hierarchy presents the processor with the illusion of a very large & fast memory

Memory Hierarchy Basis

- Cache contains copies of data in memory that are being used.
- Memory contains copies of data on disk that are being used.
- Caches work on the principles of temporal and spatial locality.
  - Temporal Locality: if we use it now, chances are we'll want to use it again soon.
  - Spatial Locality: if we use a piece of memory, chances are we'll use the neighboring pieces soon.

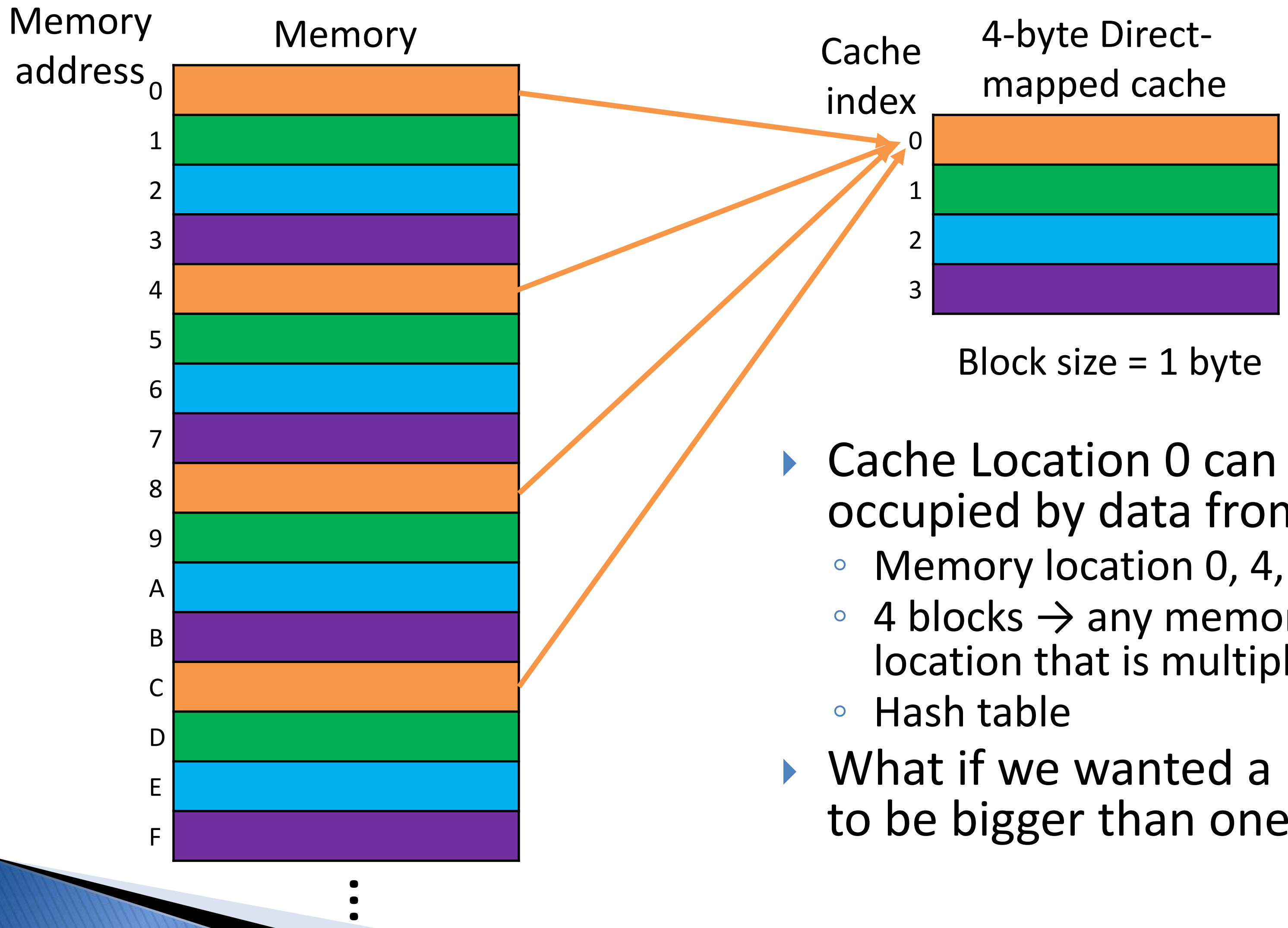
Cache Design Questions

- How do we organize cache?
- Where does each memory address map to?
  - Remember that cache is a subset of memory, so multiple memory addresses can map to the same cache location.
- How do we know which elements are in cache?
- How do we quickly locate them?

Direct-Mapped Cache (1/4)

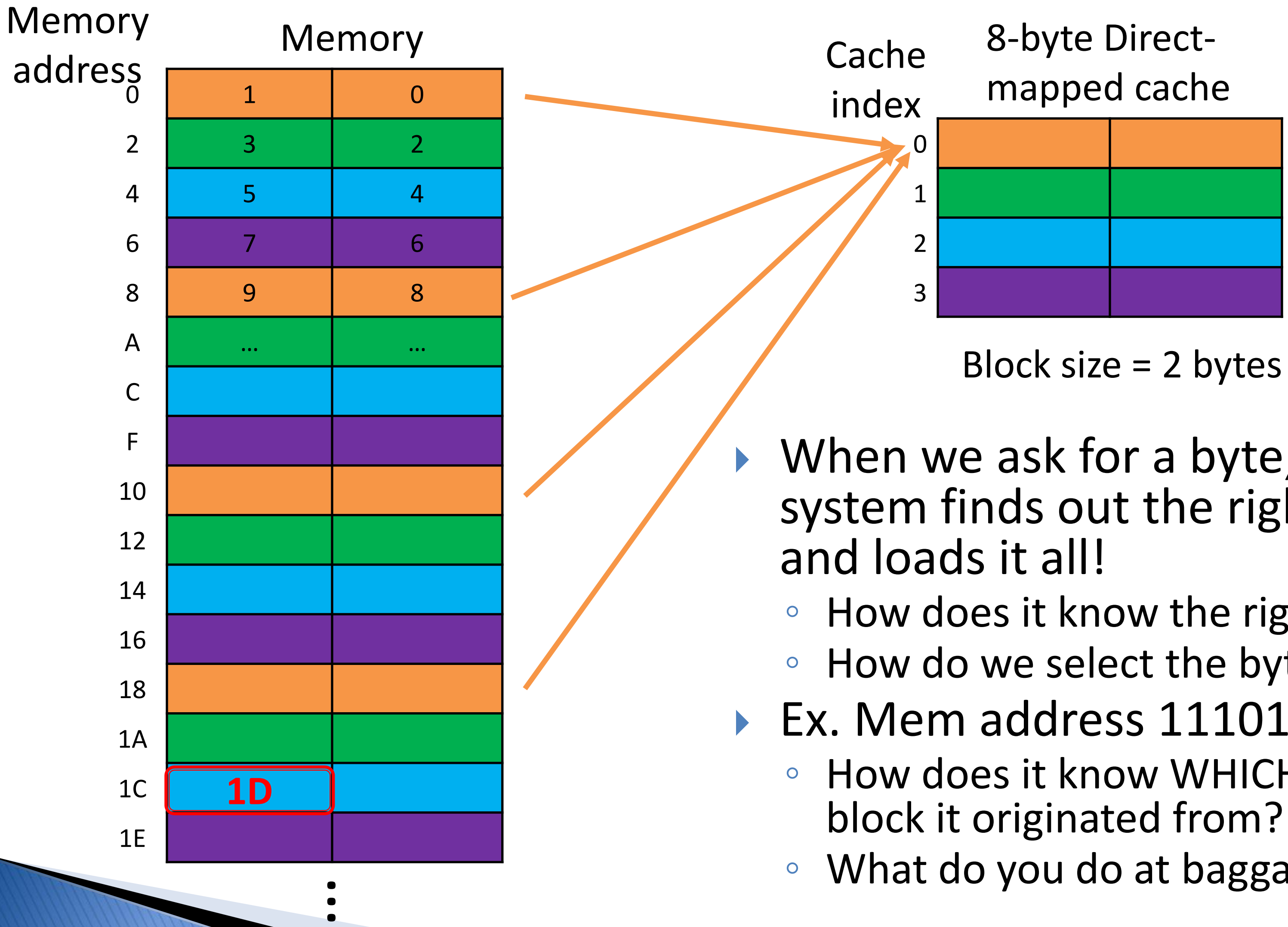
- In a direct-mapped cache, each memory address is associated with one possible block within the cache
  - Therefore, we only need to look in a single location in the cache for the data if it exists in the cache
  - Block is the unit of transfer between cache and memory

Direct-Mapped Cache (2/4)



- Cache Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ...
  - 4 blocks → any memory location that is multiple of 4
  - Hash table
- What if we wanted a block to be bigger than one byte?

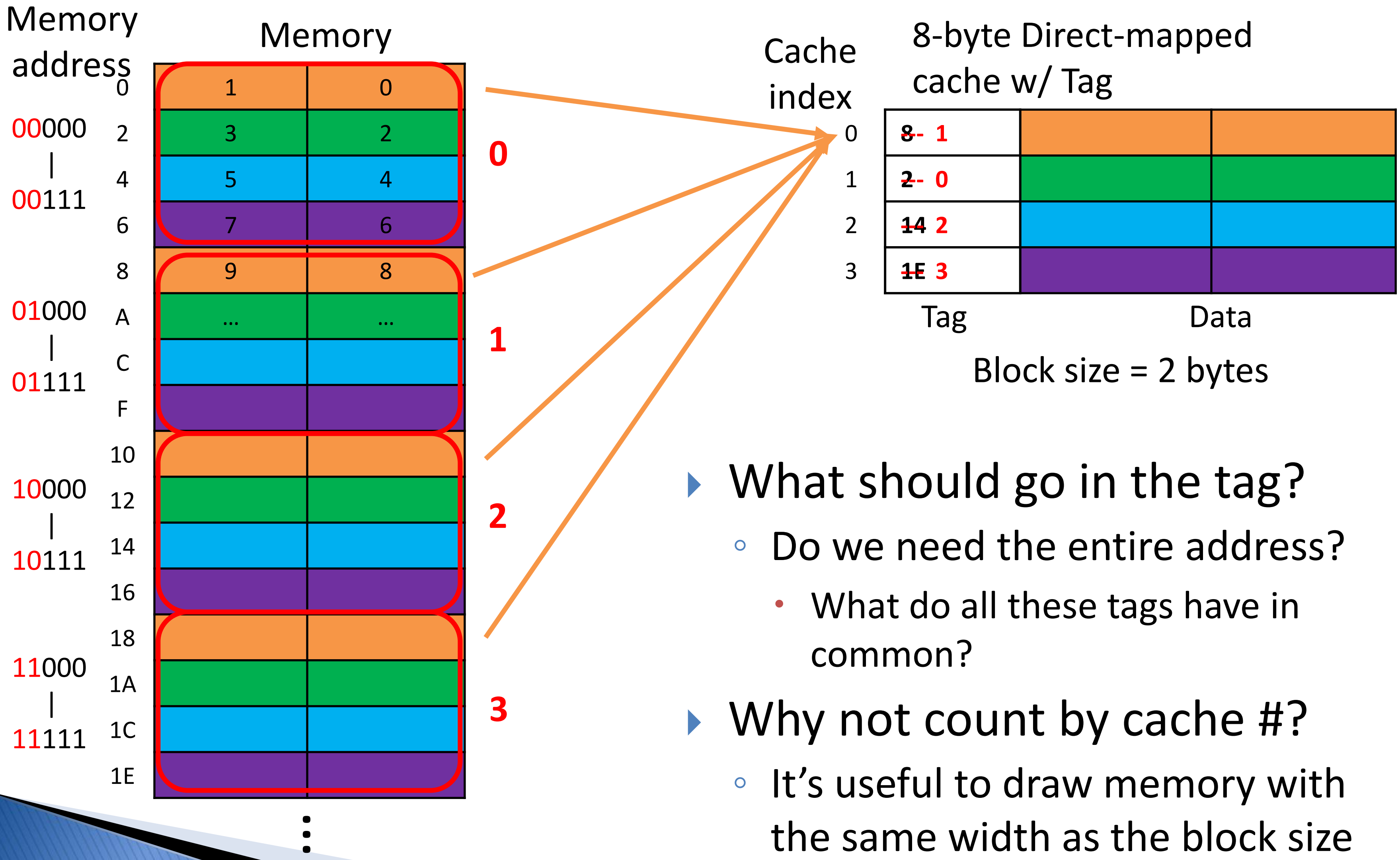
Direct-Mapped Cache (3/4)



Block size = 2 bytes

- When we ask for a byte, the system finds out the right block and loads it all!
  - How does it know the right block?
  - How do we select the byte?
- Ex. Mem address 11101 (1D)?
  - How does it know WHICH colored block it originated from?
  - What do you do at baggage claim?

Direct-Mapped Cache (4/4)



Cache index	Tag	Data	Data
0	~~8~~ 1	🟧	🟧
1	~~2~~ 0	🟩	🟩
2	~~14~~ 2	🟦	🟦
3	~~1E~~ 3	🟪	🟪

Block size = 2 bytes

- What should go in the tag?
  - Do we need the entire address?
    - What do all these tags have in common?
- Why not count by cache #?
  - It’s useful to draw memory with the same width as the block size

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

t t t t t t t t t t t t t t t t t t	i i i i i i i i i i	oooo
Tag to check if it has the correct block	Index to select block	Byte offset within block

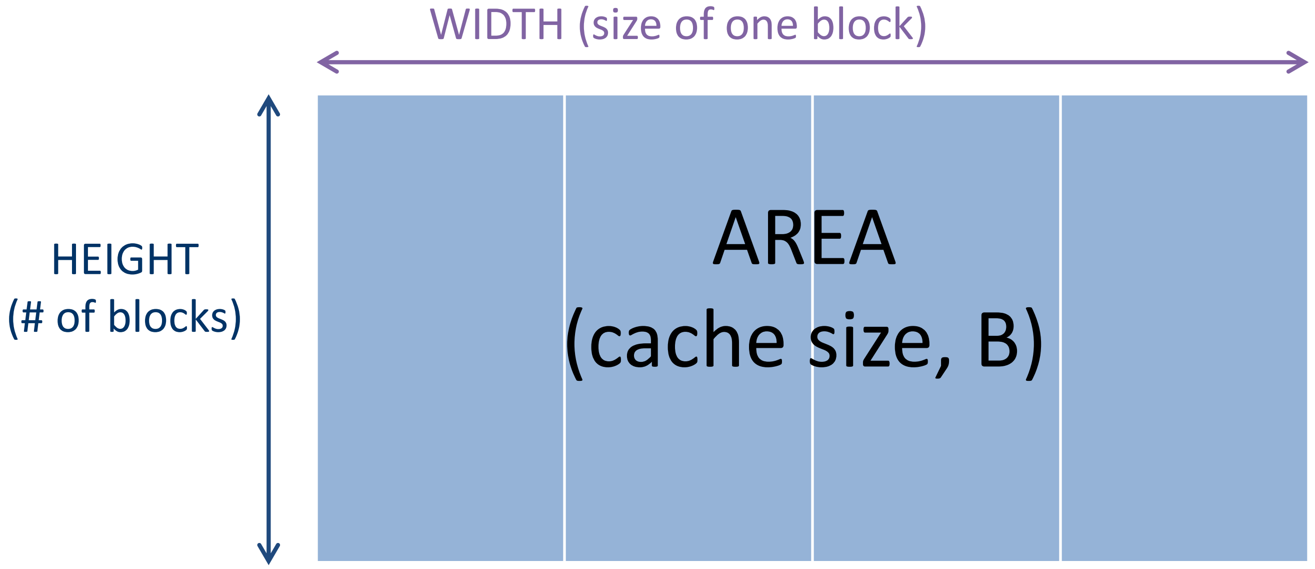
Direct-Mapped Cache Terminology

- All fields are read as unsigned integers
- Index
  - specifies the cache index (which “row”or block of the cache we should look in)
- Offset
  - once we’ve found correct block, specifies which byte within the block we want
- 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

TIO

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

Tag	Index	Offset



Direct-Mapped Cache Example (1/3)

- Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
  - Sound familiar?
- Determine the number of bits in 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 2 bytes

    $= 2^1$ bytes
  - need **1 bit** to specify correct byte

Direct-Mapped Cache Example (2/3)

- Index: (~index to an “array of blocks”)
  - need to specify correct number of block in cache
  - Cache contains 8 B = 2^3 bytes
  - block contains 2 B = 2^1 bytes
  - # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$$

$$= 2^2 \text{ blocks/cache}$$

  - need **2 bits** to specify this many blocks

Direct-Mapped Cache Example (3/3)

- Tag: use remaining bits as tag
  - tag length = addr length – offset – index
    = 32 - 1 - 2 bits
    = 29 bits
  - so tag is leftmost **29 bits** of memory address
- Why not full 32 bit address as tag?
  - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory
- Each row has Valid and Dirty bit

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 at 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)

Accessing data in a direct-mapped cache

- Ex.: 16KB of data, direct-mapped, 4 word blocks (16B)
  - Can you work out height, width, area?
- Read 4 addresses (Hex)

```
1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014
```

…	…
00000010	a
00000014	b
00000018	c
0000001C	d
…	…
00000030	e
00000034	f
00000038	g
0000003C	h
…	…
00008010	i
00008014	j
00008018	k
0000801C	l
…	…

Memory

Accessing data in a direct-mapped cache

- 4 Addresses:
  - `0x00000014, 0x0000001C, 0x00000034, 0x00008014`
- 4 Addresses divided (for convenience) into Tag, Index, Byte Offset fields

Tag	Index	Offset
000000000000000000	0000000001	0100
000000000000000000	0000000001	1100
000000000000000000	0000000011	0100
000000000000000010	0000000001	0100

16 KB Direct Mapped Cache, 16B blocks

- Valid bit: determines whether anything is stored in that row (start with 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					

Read 0x00000014

Tag	Index	Offset
000000000000000000	0000000001	0100

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					

Read block 1

Tag	Index	Offset
000000000000000000	**0000000001**	0100

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

Tag	Index	Offset
000000000000000000	**0000000001**	0100

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					

Load data into cache, set tag and valid

Tag	Index	Offset
000000000000000000	0000000001	0100

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					

Read from cache at offset

Tag	Index	Offset
000000000000000000	0000000001	**0100**

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					

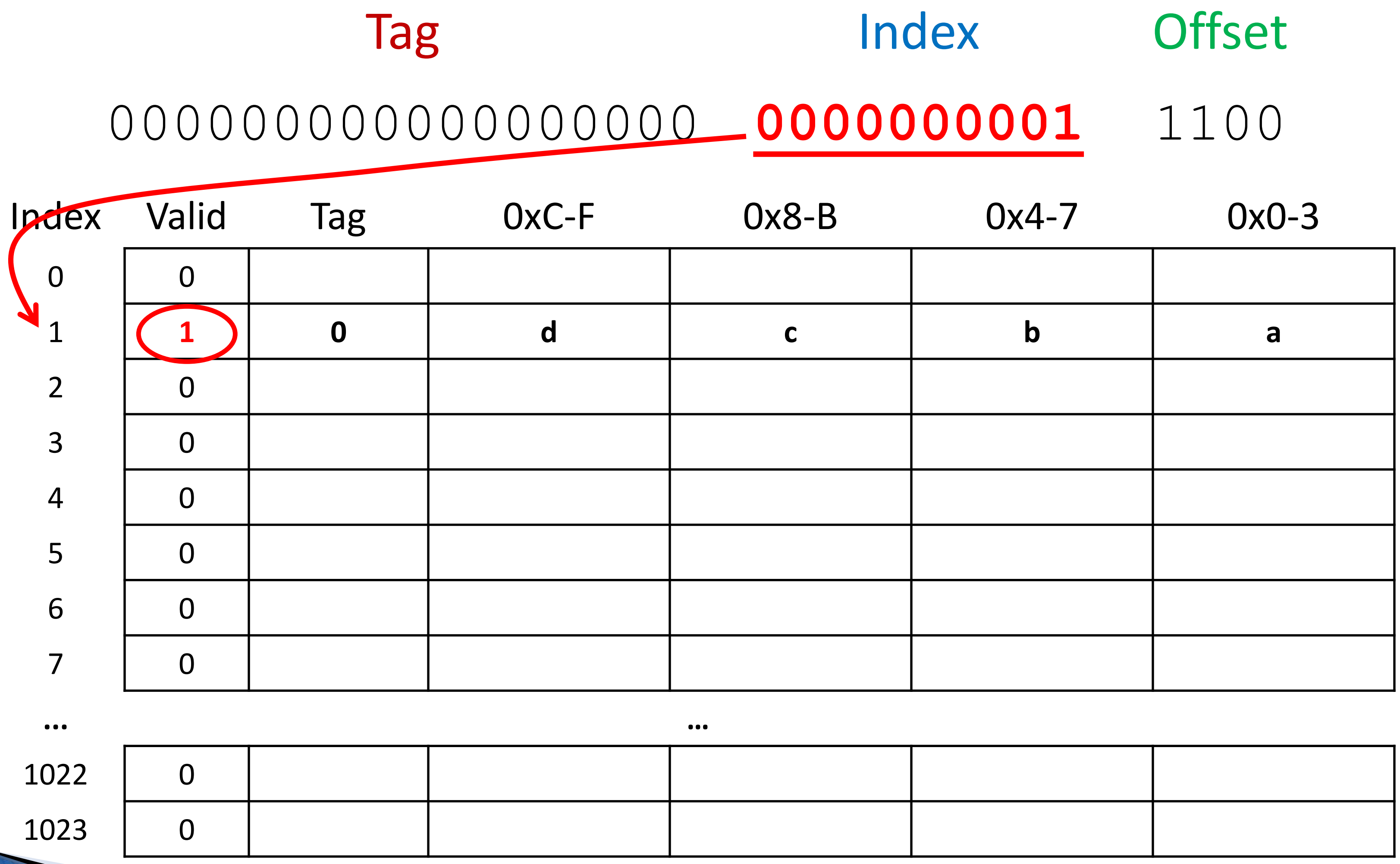
Return word b

Read 0x0000001C

Tag	Index	Offset
000000000000000000	0000000001	1100

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



Tag | Index | Offset

0000000000000000000 **0000000001** 1100

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					

Tag matches

Tag	Index	Offset
000000000000000000	0000000001	1100

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					

Read from cache at offset

Tag	Index	Offset
000000000000000000	0000000001	**1100**

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					

Return word d

Read 0x00000034

Tag	Index	Offset
000000000000000000	0000000011	0100

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					

Read block 3

Tag	Index	Offset
000000000000000000	**0000000011**	0100

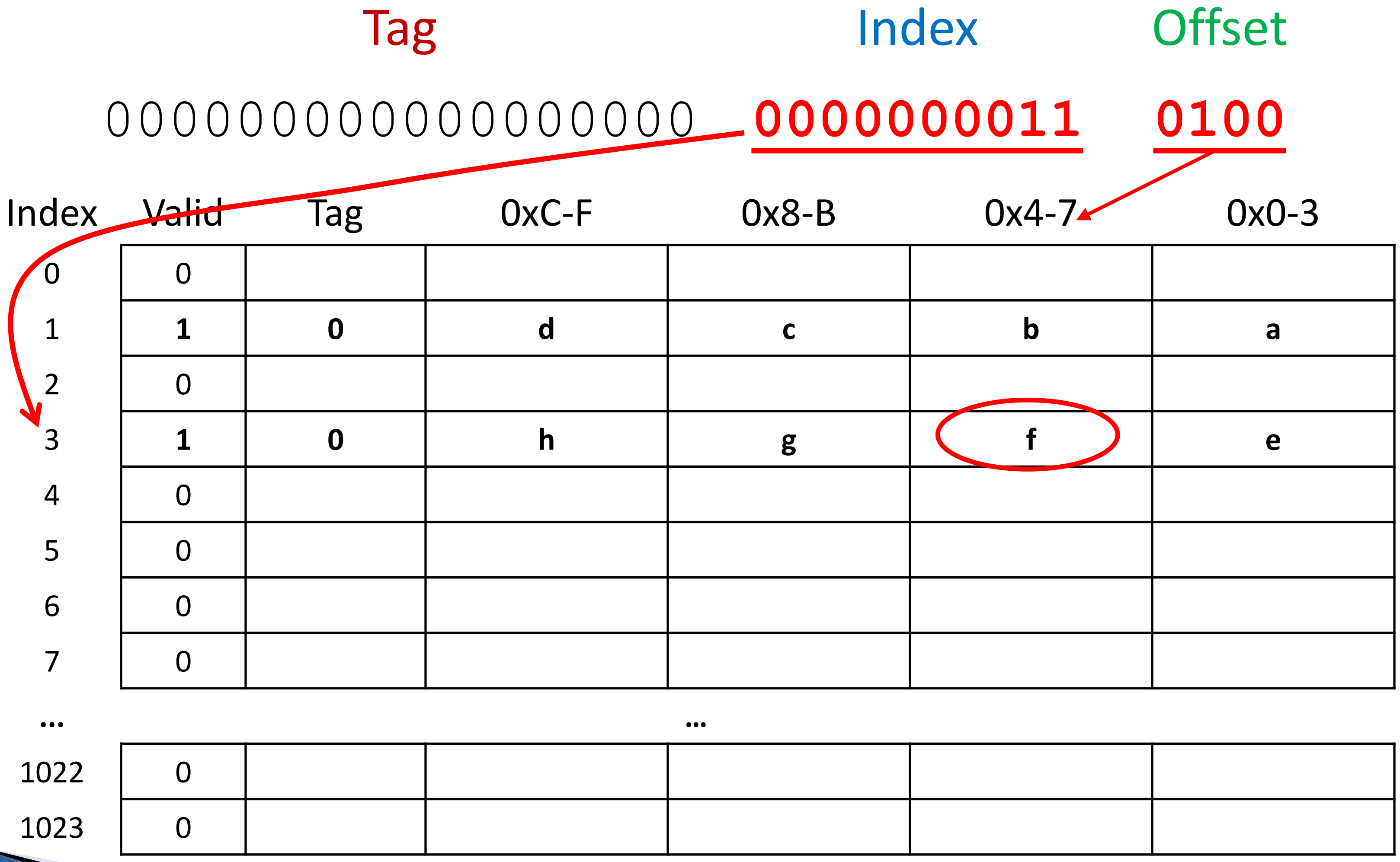
Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	**1**	**0**	**d**	**c**	**b**	**a**
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

No valid data

Tag	Index	Offset
000000000000000000	**0000000011**	0100

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					

Load data, read cache at offset



Tag Index Offset

000000000000000000 0000000011 0100

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					

Return word f

Read 0x00008014

Tag Index Offset

000000000000000010 0000000001 0100

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					

Read block 1, valid data

Tag	Index	Offset
000000000000000010	**0000000001**	0100

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					

Tag not matched!

Tag	Index	Offset
000000000000000010	0000000001	0100

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: replace block 1

Tag	Index	Offset
000000000000000010	0000000001	0100

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					

Read cache at offset

Tag	Index	Offset
000000000000000010	0000000001	**0100**

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					

Return word j

Do these examples yourself

- Read address 0x00000030
- Read address 0x0000001C
- Cache: Hit, Miss, or Miss with replace?
- Returned values: a, b, c, …, k, l?

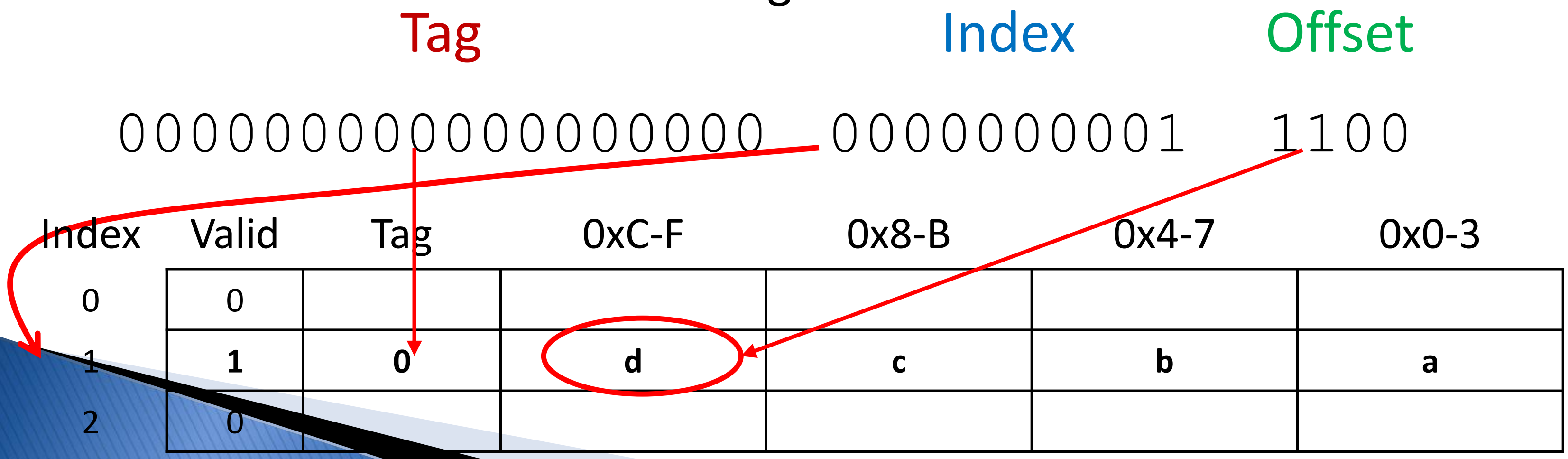
Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	**1**	**2**	**l**	**k**	**j**	**i**
2	0					
3	**1**	**0**	**h**	**g**	**f**	**e**
4	0					
5	0					
6	0					
7	0					

Answers

- 0x00000030: hit
  - Index = 3, Tag matches
  - Offset = 0, returned value = e
- 0x0000001c: miss
  - Index = 1, Tag mismatch, so replace from memory
  - Offset = 0xc, value = d

Summary

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



Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	**1**	**0**	**d**	**c**	**b**	**a**
2	0					