



# EECS4030: Computer Architecture

## Memory Hierarchy (I)

Prof. Chung-Ta King

Department of Computer Science

National Tsing Hua University, Taiwan

(Adapted from textbook slides <https://www.elsevier.com/books-and-journals/book-companion/9780128122754/lecture-slides>)



國立清華大學

National Tsing Hua University



# Outline

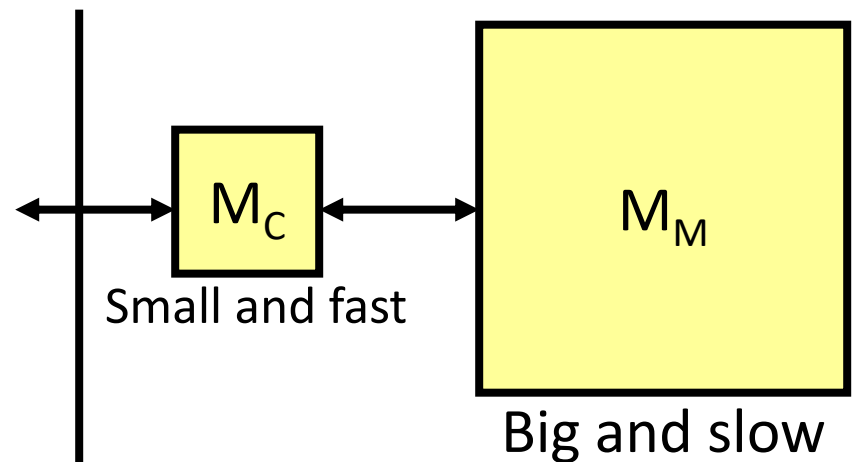
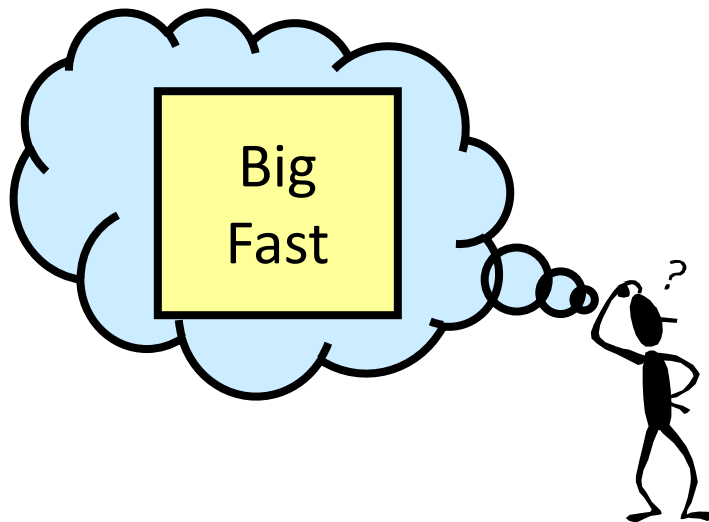
- Introduction to memory hierarchy (Sec. 5.1)
- Memory technologies (Sec. 5.2, 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies: cache coherence, redundant arrays of inexpensive disks (Sec. 5.10, 5.11)



# Why Memory Hierarchy?

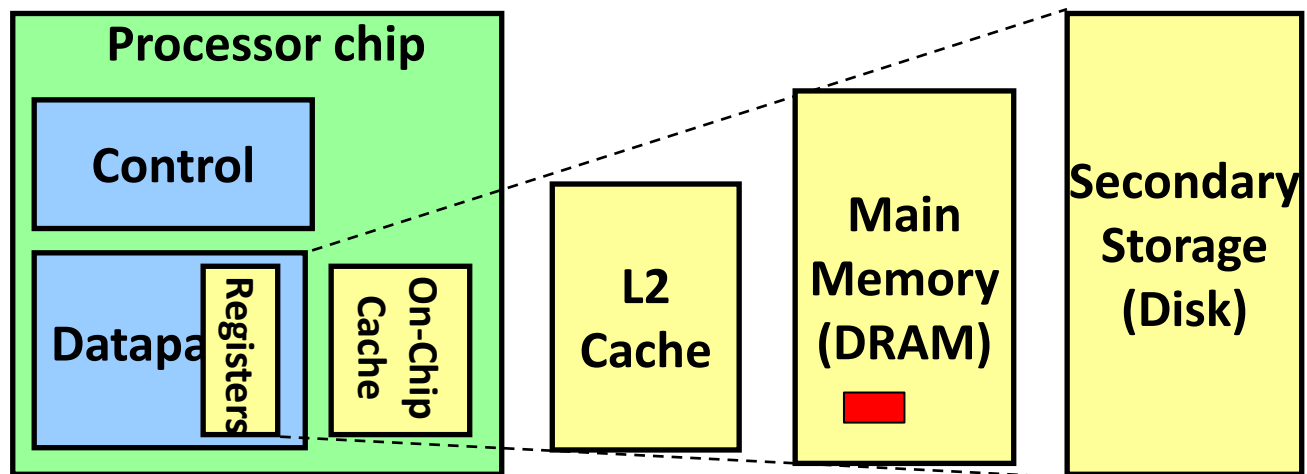
- Programmers want unlimited amounts of memory with low latency
  - But fast memory more expensive than slower memory
- Solution: small fast memory + big slow memory  
= Logically looks like a big fast memory

*Our view in Ch. 1~4*



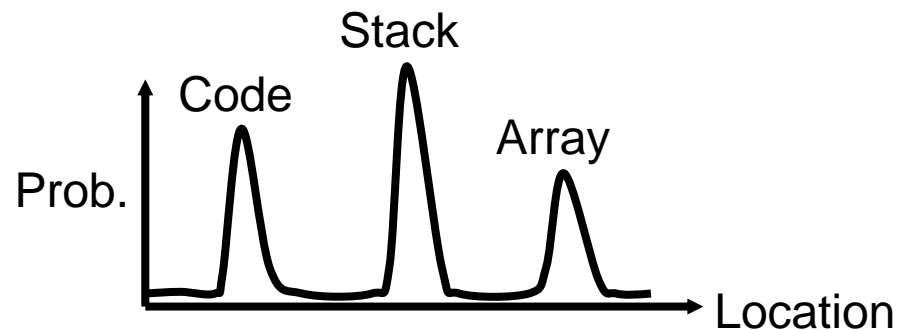
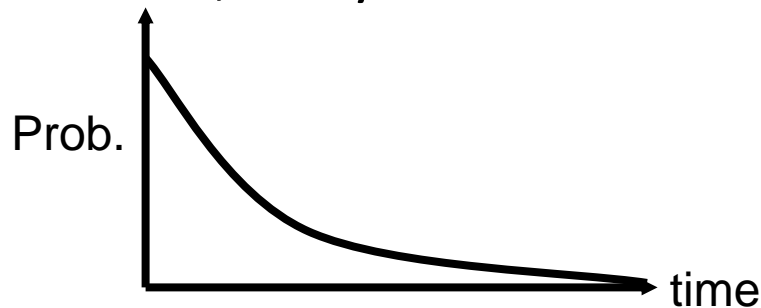
# How to Use Memory Hierarchy?

- Entire addressable memory space (program's view) available in largest, slowest physical memory
  - Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed toward processor
  - *Upper level*: closer to processor (smaller, faster, expensive)
  - *Lower level*: away from processor (bigger, slower, cheap)



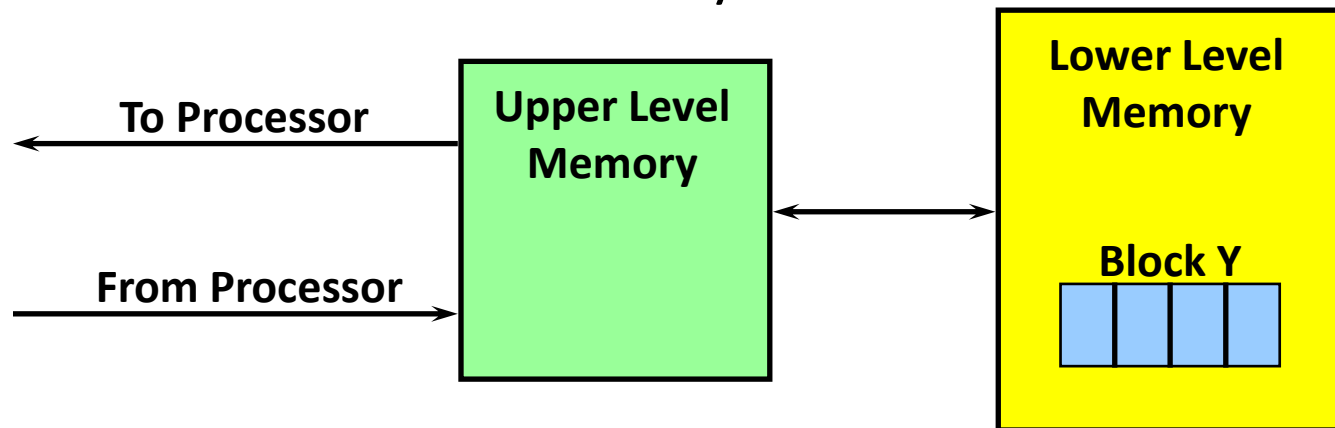
# Why Memory Hierarchy Works?

- **Principle of locality:**
  - Programs access a small proportion of their address space (data and instruction) at any time → a program property
- Two types of locality:
  - *Temporal locality*: items accessed recently are likely to be accessed again soon, e.g., instructions in a loop
  - *Spatial locality*: items near those accessed recently are likely to be accessed soon, e.g., sequential instruction access, array data

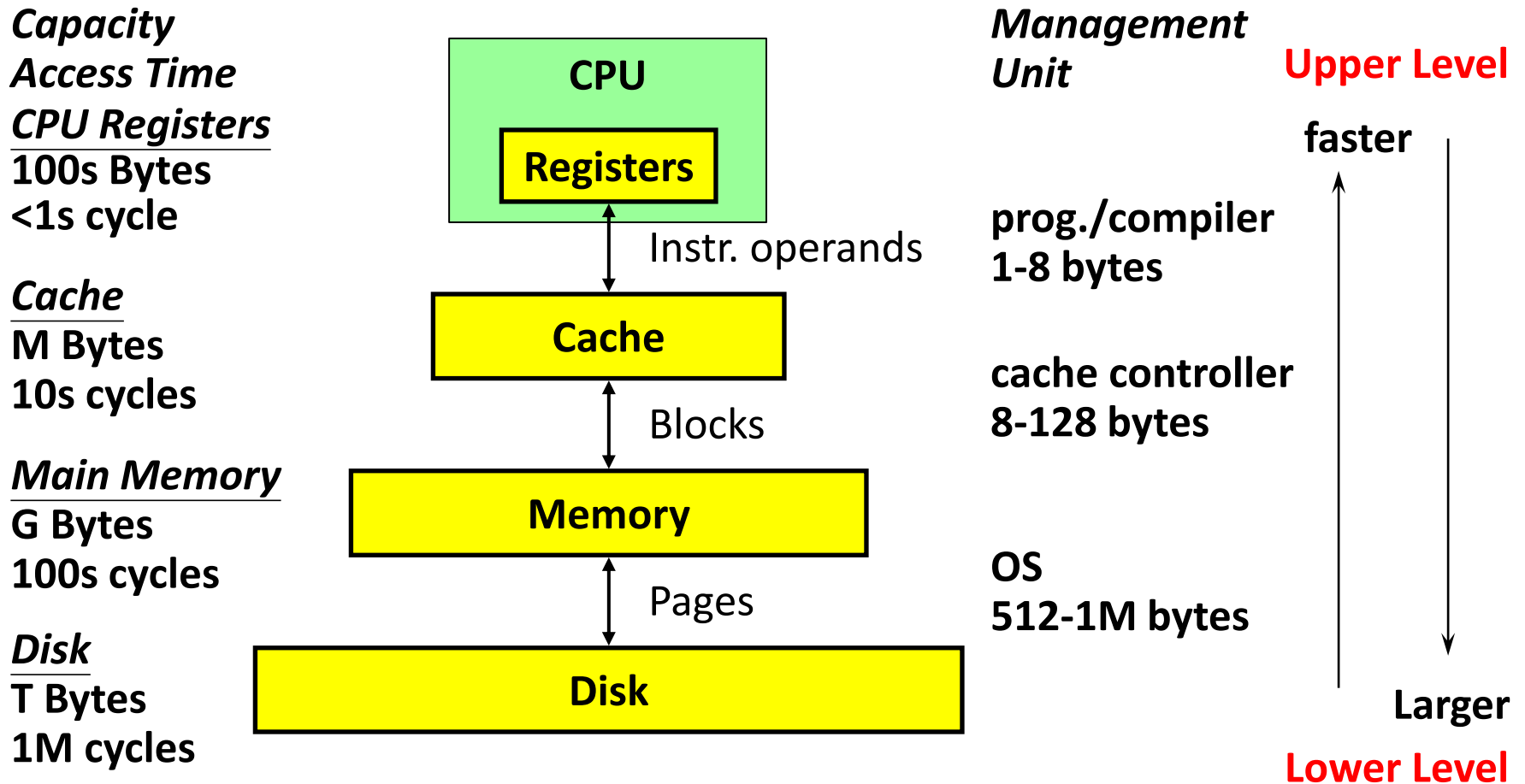


# How Does It Work?

- Temporal locality: keep most recently accessed data items closer to the processor (faster reuse)
- Spatial locality: move *blocks* consisting of contiguous words to the upper levels (bring in neighbors)
- *Block* (or *page*): basic unit of data transfer
  - Minimum unit of data that can either be present or not present in a level of the hierarchy

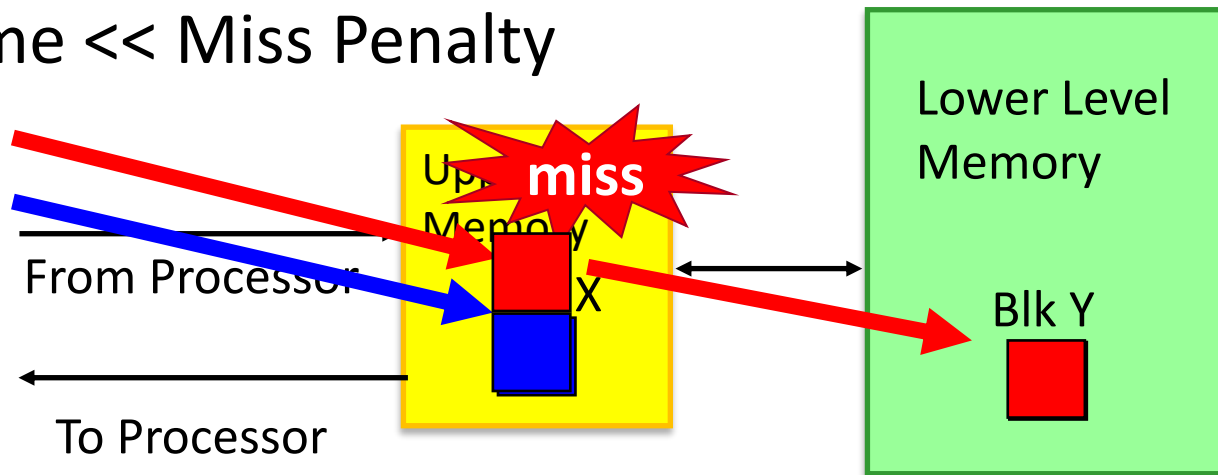


# Levels of Memory Hierarchy



# Hit or Miss

- *Hit*: accessed data appears in upper level
  - **Hit rate**: % of memory accesses found in upper level
  - **Hit time**: time to access upper level
- *Miss*: data not in upper level
  - **Miss rate** =  $1 - (\text{hit rate})$
  - **Miss penalty**: time to replace a block in upper level + time to deliver the block to the processor
- Hit Time  $\ll$  Miss Penalty







# 4 Questions for Memory Hierarchy Design

Q1: Where can a block be placed in the upper level?

- One fixed location, a few locations, anywhere

Q2: How is a block found if it is in the upper level?

- Depending on strategy taken in Q1
- Indexing, table lookup, parallel search

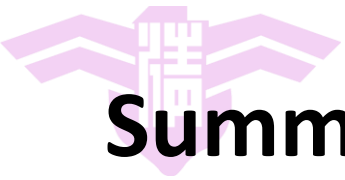
Q3: Which block should be replaced on a miss?

- Ideal: one that not to be used in the farthest future
- Practical: use the past to predict the future  
→ how to “track” the past?

Q4: What happens on a write?

- Write to upper level only or to lower level as well
- What if write miss?





# Summary of Memory Hierarchy

- Two different types of locality:
  - Temporal locality (locality in time)
  - Spatial locality (locality in space)
- Using the principle of locality:
  - Present the user/program with as much memory as is available in the cheapest technology
  - Provide access at the speed offered by the fastest memory
- End result: provide user/program an illusion of a **big fast random access memory**





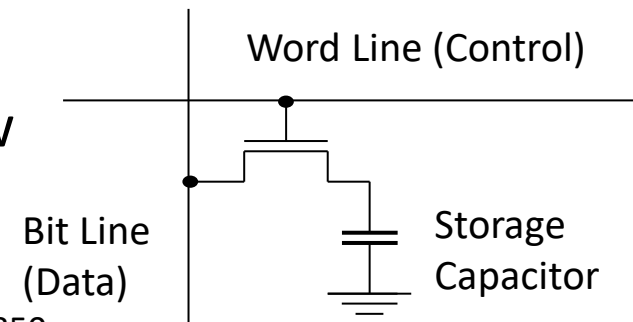
# Outline

- Introduction to memory hierarchy (Sec. 5.1)
- **Memory technologies (Sec. 5.2, 5.5)**
  - Dependable memory hierarchy (Sec. 5.5)
- Caches (Sec. 5.3, 5.4, 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies: cache coherence, redundant arrays of inexpensive disks (Sec. 5.10, 5.11)





- [illegible]





# Comparisons of Memory Technologies

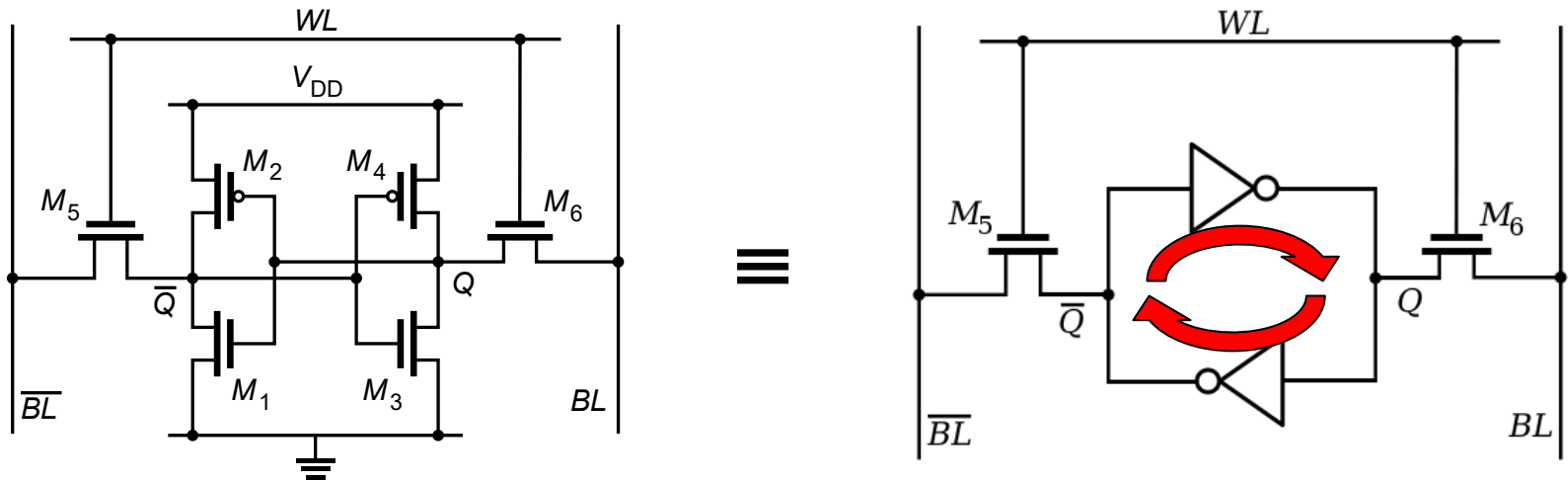
Memory technology	Typical access time	\$ per GB in 2012
SRAM	0.5 ~ 2.5 ns	\$500 ~ \$1,000
DRAM	50 ~ 70 ns	\$10 ~ \$20
Flash memory	5 ~ 50 ms	\$0.75 ~ \$1.00
Magnetic disk	5 ~ 20 ms	\$0.05 ~ \$0.10

- Ideal memory
  - Access time of SRAM
  - Capacity and cost/GB of disk



# SRAM Cell

- Typical 6-transistor SRAM cell:



- Two cross-coupled inverters create a bi-stable circuit
- Reading:  $WL = 1 \rightarrow M_5, M_6$  on  $\rightarrow BL$  gives stored bit
- Writing:  $WL = 1 \rightarrow$  if  $M_5/M_6$  driven stronger than  $M_1 \sim M_4$   
 $\rightarrow$  change state of  $Q$

[https://en.wikipedia.org/wiki/Memory\\_cell\\_\(computing\)](https://en.wikipedia.org/wiki/Memory_cell_(computing))

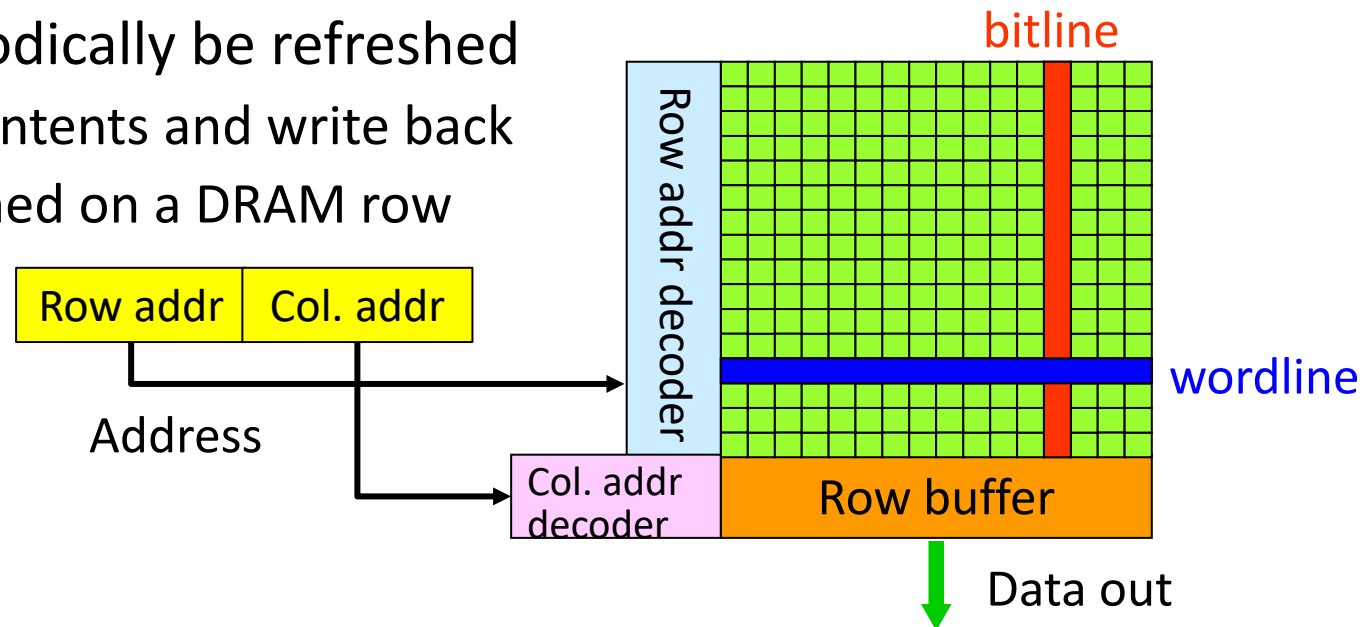
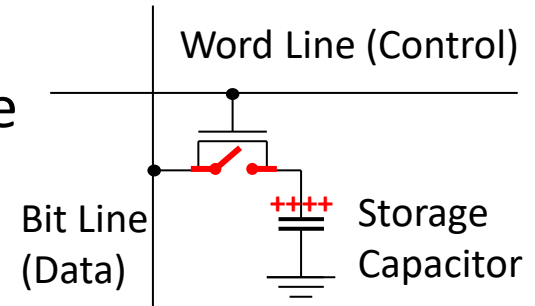


國立清華大學

National Tsing Hua University

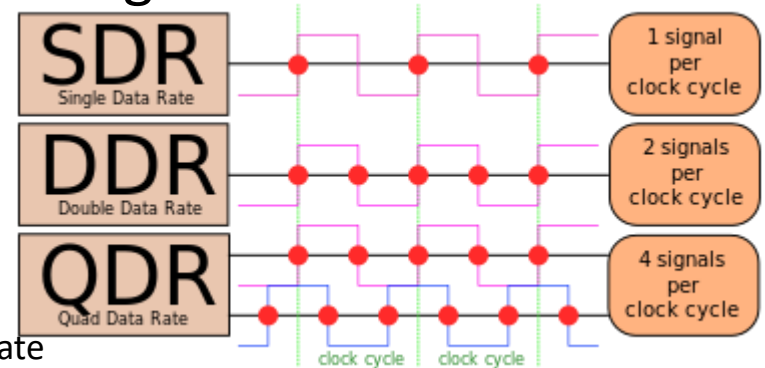
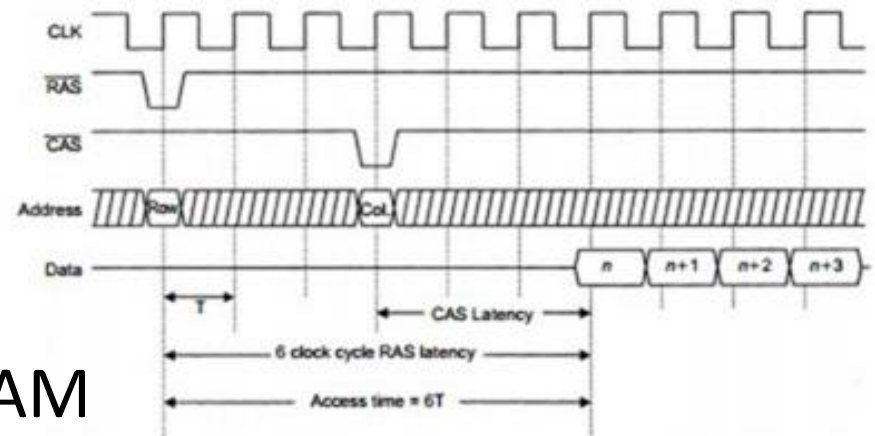
# DRAM Technology

- Data stored as a charge in a capacitor
  - Single transistor used to access the charge
  - Destructive read and need to write back
  - Bits are organized as a rectangular array
    - An entire row is accessed into a *row buffer*
  - Must periodically be refreshed
    - Read contents and write back
    - Performed on a DRAM row



# Advanced DRAM Organizations

- Burst mode:
  - Supply successive words from row buffer  $\rightarrow$  reduce latency
- Synchronous DRAM:
  - Clocking DRAM interface for repeated transfers without handshaking overhead
- Double data rate (DDR) DRAM
  - Transfer on rising and falling clock edges
- DRAM banking:
  - Multiple DRAM banks allowing simultaneous accesses (next slide)



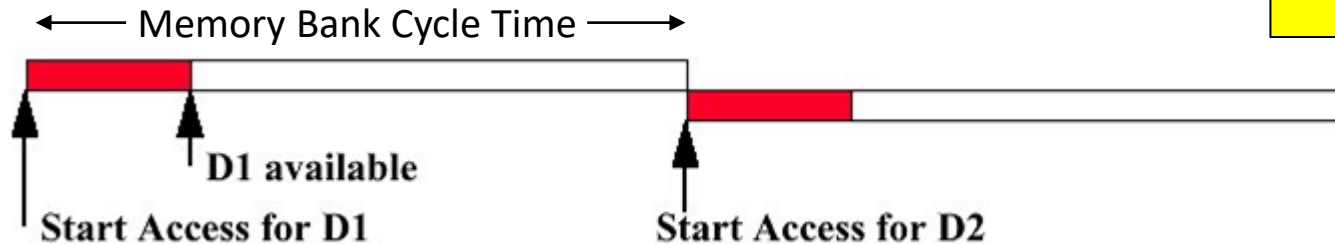
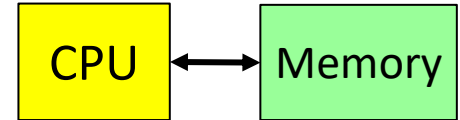
[https://en.wikipedia.org/wiki/Double\\_data\\_rate](https://en.wikipedia.org/wiki/Double_data_rate)



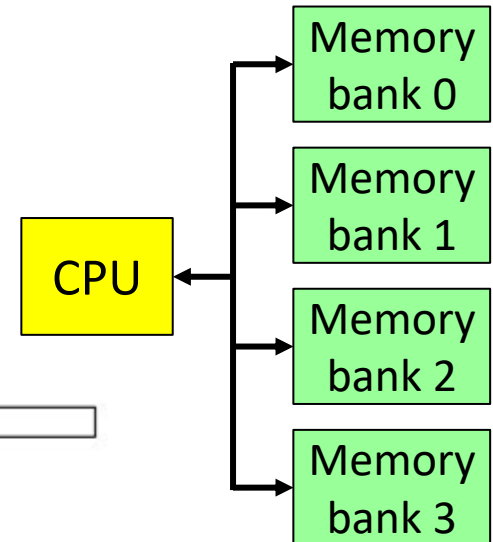
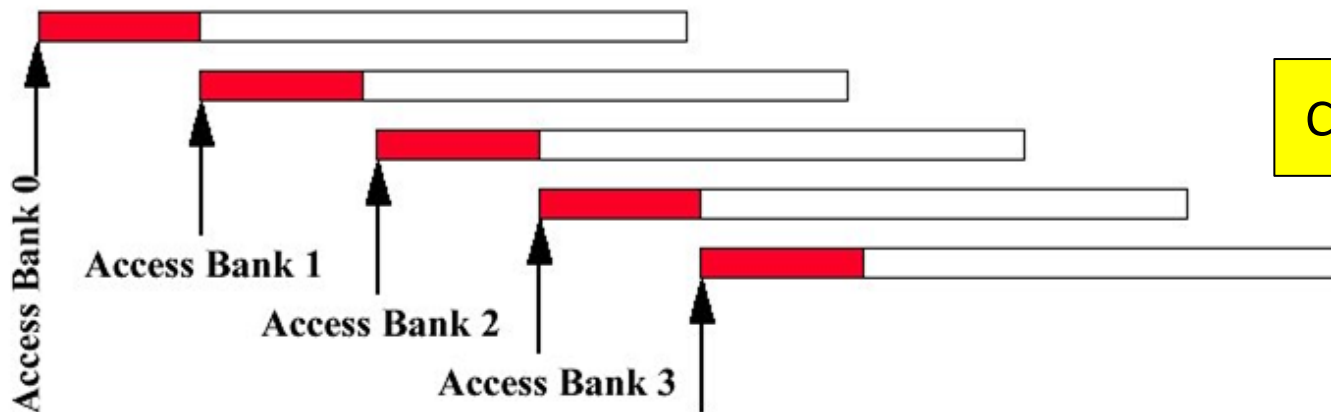


# DRAM Banking

- One memory bank



- Four interleaved memory banks
  - Pipelined access for increased bandwidth



We can Access Bank 0 again

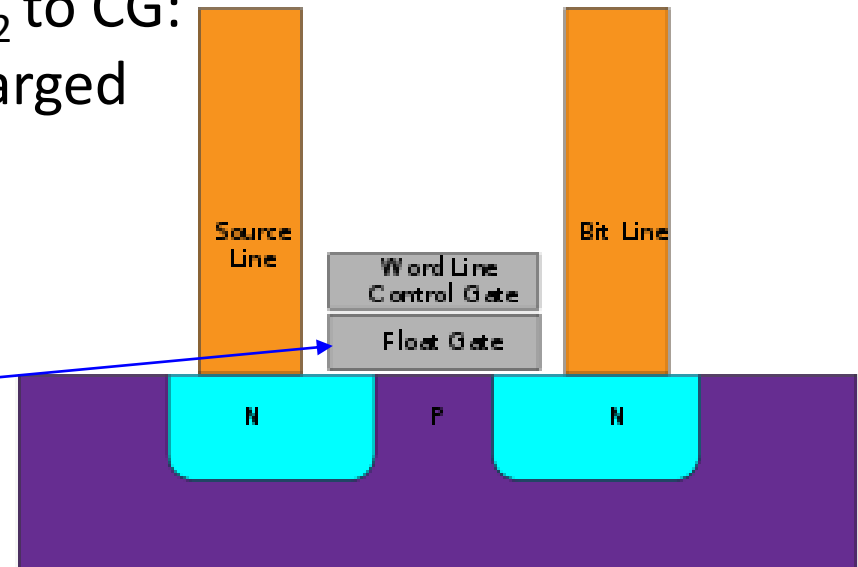
<https://people.rit.edu/meseec/cmpe550-spring2018/>



# Flash Storage

- Nonvolatile semiconductor storage
  - 100x ~ 1000x faster than disk
  - Smaller, lower power, more robust, more \$/GB
  - Store bits in *floating-gate transistors*
  - Apply a voltage  $V_{T1} < V < V_{T2}$  to CG:  
if transistor on  $\rightarrow$  FG uncharged  
 $\rightarrow$  a logic “1” is stored

*Floating gate (FG)*  
traps N charges and  
changes threshold  
voltage ( $V_{T1} \rightarrow V_{T2}$ ) at  
*control gate (CG)* to  
turn on transistor

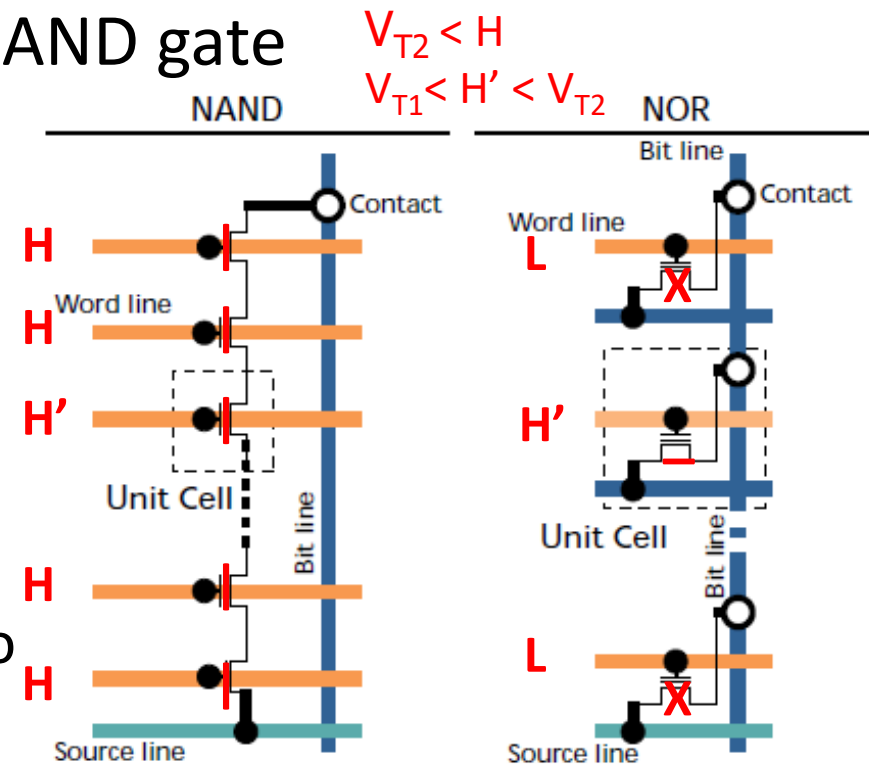


(By Cyferz at English Wikipedia, CC BY 2.5,  
<https://commons.wikimedia.org/w/index.php?curid=47813471>)



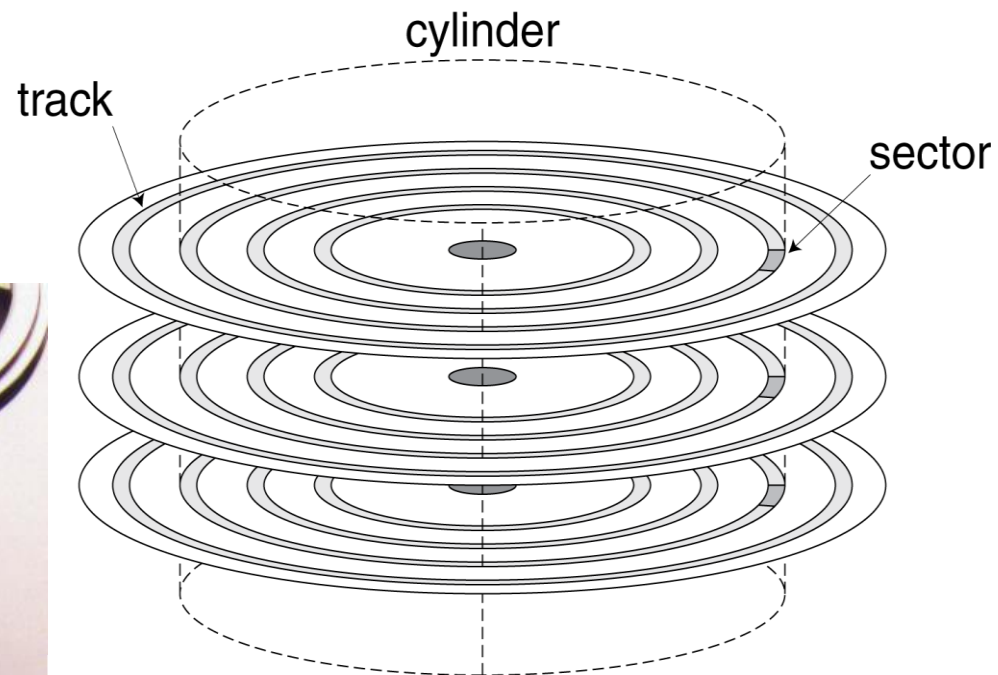
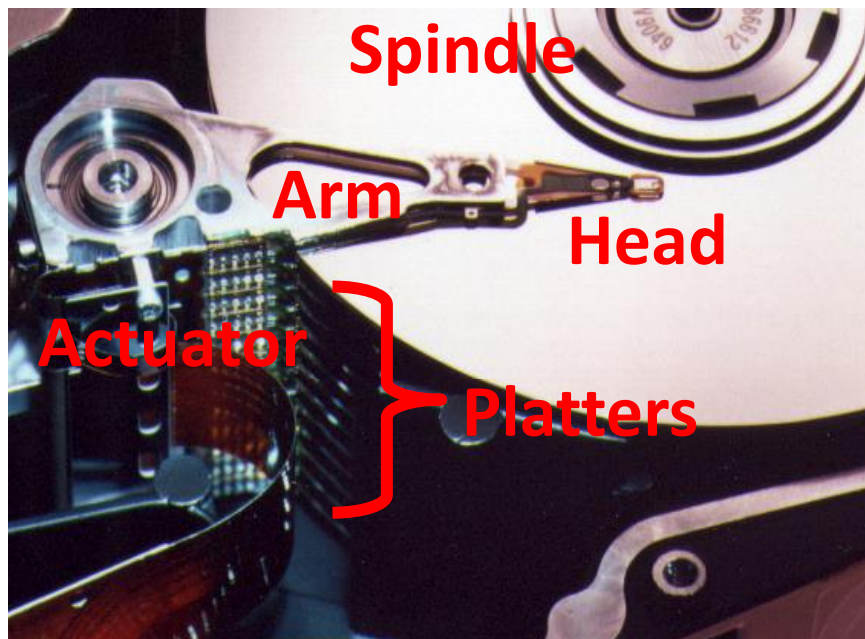
# Flash Storage

- NOR flash: bit cell like a NOR gate
  - Random read/write access
  - Used for instruction memory in embedded systems
- NAND flash: bit cell like a NAND gate
  - Denser and cheaper, block-at-a-time access
  - Used for USB keys, media storage, ...
- Flash bits wears out after 1000's of accesses
  - *Wear leveling*: remap data to less used blocks



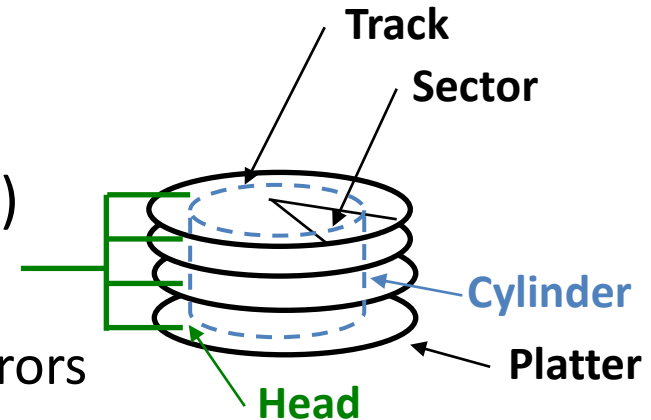
# Disk Storage

- Nonvolatile, rotating magnetic storage



# Disk Sectors and Access

- Each sector records
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - *Error correcting code (ECC)*
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- Access to a sector involves
  - Queuing delay: if other accesses are pending
  - Seek time: time to move the heads to desired track
  - Rotational latency: time for desired sector under head
  - Data transfer: time to transfer desired sector
  - Controller overhead





# Disk Access Example

- Given
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk
- Average read time
  - 4ms seek time
  - +  $\frac{1}{2} / (15,000/60) = 2\text{ms}$  rotational latency
  - +  $512 / 100\text{MB/s} = 0.005\text{ms}$  transfer time
  - + 0.2ms controller delay
  - = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms





# Outline

- Introduction to memory hierarchy (Sec. 5.1)
- **Memory technologies (Sec. 5.2, 5.5)**
  - **Dependable memory hierarchy (Sec. 5.5)**
- Caches (Sec. 5.3, 5.4, 5.9)
- Virtual memory (Sec. 5.7)
- Framework for memory hierarchy (Sec. 5.8)
- Virtual machines (Sec. 5.6)
- Parallelism and memory hierarchies: cache coherence, redundant arrays of inexpensive disks (Sec. 5.10, 5.11)





# Is Your Memory Reliable?

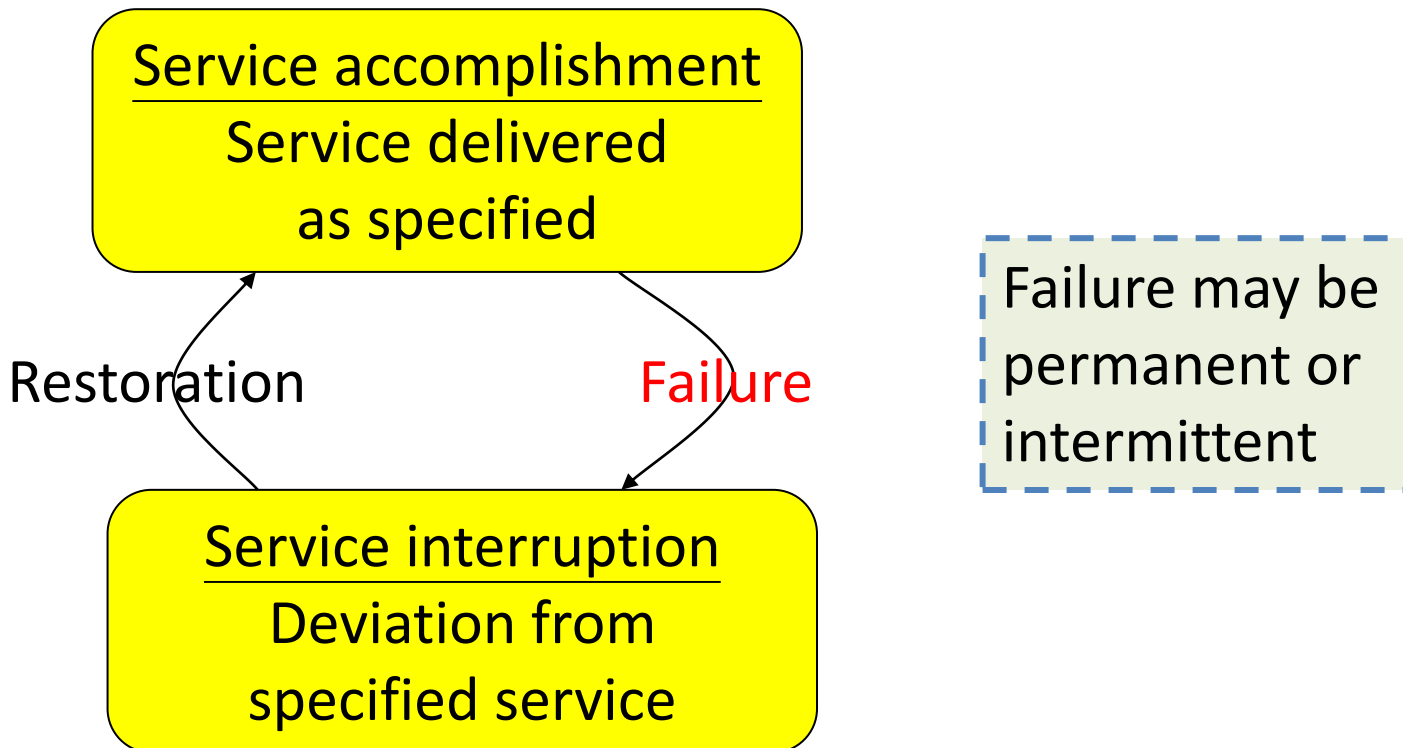
- Suppose a processor writes a number  $0\dots 0101$  to a memory location  $X$  and immediately reads from  $X$ . Will the processor always get  $0\dots 0101$ ?
- Ans.: not necessarily
  - Because memory may be unreliable and have failures, e.g., one or two bits are flipped
- How do you know there is an error in the data that is returned from the memory?
  - We say that the memory has *failures*





# Is Your Memory Reliable?

- What is *failure*?
  - Given a specification of proper service (of memory)



- *Fault*: failure of a component, may or may not cause failure





# Reliability vs. Availability

- *Reliability*: a measure of continuous service accomplishment (or the time to failure), starting from a reference point
  - *Mean time to failure* (MTTF): a reliability measure
  - *Mean time to repair* (MTTR): a measure of service interruption
  - *Mean time between failures* (MTBF):  $= \text{MTTF} + \text{MTTR}$
- *Availability*: a measure of service accomplishment with respect to the alternation between the two states of accomplishment and interruption
  - $\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$





# Reliability vs. Availability

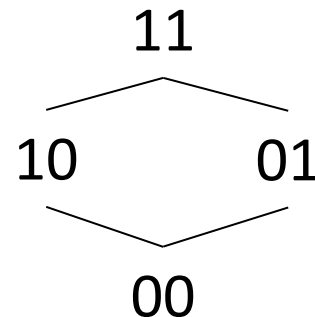
- We want *high availability* (HA)
  - Can be expressed as # of “nines of availability” per year
    - One nine: 90% → 36.5 days of repair/year
    - Five nines: 99.999% → 5.26 minutes of repair/year
- HA can be achieved by
  - Reducing MTTR: improved tools and processes for fault detection, diagnosis and repair
  - Increasing MTTF: improve quality of components or design systems to continue operation in the presence of components that have failed
    - *Fault avoidance*: reliable components to prevent faults
    - *Fault tolerance*: redundant components, fault but not failure
    - *Fault forecasting*: predict and replace before failure



# Fault Tolerance for Memory

## Hamming Single Error Correcting, Double Error Detecting Code (SECDED)

- *Hamming distance*:
  - Minimum number of bits that are different between any two correct bit patterns
    - A measure of how “close” correct bit patterns can be
    - e.g., Hamming distance between 011011 and 001111 is two




- Single Error Detecting Code:
  - A set of legal bit patterns that can detect single bit error




# Single Error Detecting Code

- The simplest way to encode four different things:

 → 00


 → 01

 → 10


 → 11


- Suppose you send the code “01” to ask your friend to bring you a banana, but the code was erroneously transmitted as “11” → Can she/he tell there is a bit error?
- How about this code?



 → 00**0**

 → 01**1**

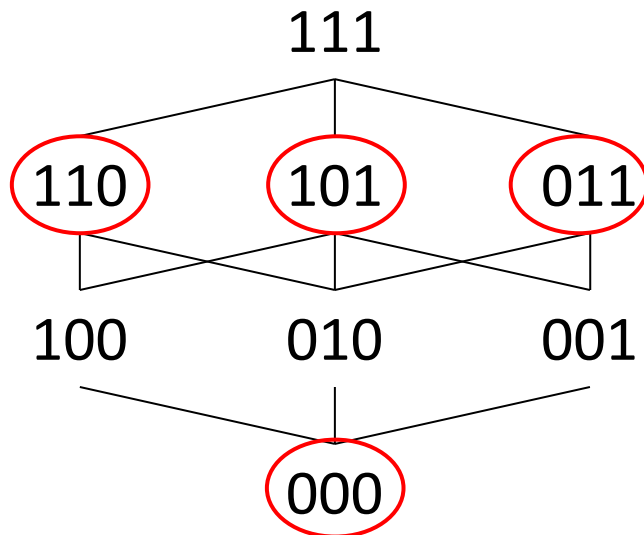
 → 10**1**

 → 11**0**



# Single Error Detecting Code

- Hamming distance of 3-bit codes
  - A link between two codes whose Hamming distance = 1



Legal bit patterns are actually original codes (00, 01, 10, 11) plus even parity (bit 0)

0010 1100 → 0010 1100 1  
0010 1101 → 0010 1101 0

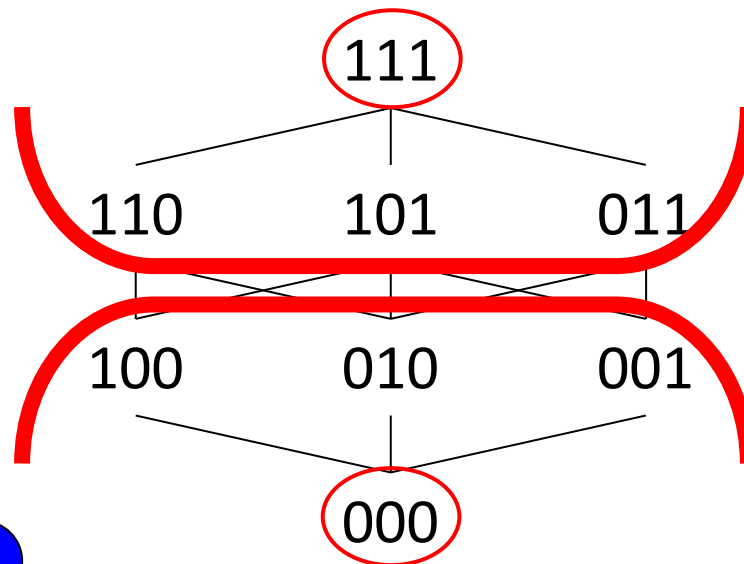
- Single Error Detecting Code:
  - Condition: if only every pair of legal bit patterns has a Hamming distance of 2, e.g., parity code

But which bit is in error?



# Single Error Correcting (SEC) Code

- A code with a Hamming distance of 3 provides *single error correction* (Why?)



Need an  
algorithm to  
decide the code

Given any bit pattern that has one bit error,  
we can identify the closest correct code



# Single Error Correcting (SEC) Code

- *Hamming Error Correction Code (ECC):*
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks group of data bits via even parity:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

If a bit is altered, it is checked by parities of the covering groups (11 = p1,p2,p8)

Fig. 5.24





# Single Error Correcting (SEC) Code

- Sum of positions of erroneous parity bits identifies the erroneous bit, e.g.,
  - Erroneous parity bits = 0000 indicates no error
  - Erroneous parity bits = 1001  $\rightarrow$  bit  $1+8=9$  was flipped
- Encoding example:
  - Encode  $10011010_2 \rightarrow \_ \_ 1 \_ 0 0 1 \_ 1 0 1 0?$ 
    - p1 should make bits 1, 3, 5, 7, 9, 11 even parity  
 $\_ \_ \textcolor{red}{1} \_ \textcolor{red}{0} 0 \textcolor{red}{1} \_ \textcolor{red}{1} 0 \textcolor{red}{1} 0 \rightarrow p1 = 0$
    - P2 should make bits 2, 3, 6, 7, 10, 11 even parity  
 $\_ \_ \textcolor{red}{1} \_ 0 0 \textcolor{red}{1} \_ 1 \textcolor{red}{0} \textcolor{red}{1} 0 \rightarrow p2 = 1$
    - ...
  - Final code word =  $\textcolor{red}{0}\textcolor{red}{1}\textcolor{red}{1}\textcolor{red}{1}00101010$



# Single Error Correcting (SEC) Code

- Correction example:

How to tell if 1-  
or 2-bit error?

- Consider the code word = **0111**001**0**1010
- If we detect bit pattern 011100**00**1010, which bit is error?
  - Bits 1, 3, 5, 7, 9, 11 should be even parity, but they are not  
**0** 1 **1** 1 **0** 0 **0** 0 **1** 0 **1** 0  $\rightarrow H = h_1h_2h_4h_8 = 1 \_ \_ \_$
  - Bits 2, 3, 6, 7, 10, 11 should be even parity, but they are not  
0 **1** **1** 1 0 **0** 0 0 1 **0** **1** 0  $\rightarrow H = h_1h_2h_4h_8 = 1 \ 1 \_ \_$
  - Bits 4, 5, 6, 7, 12 should be even parity, but they are not  
0 1 1 **1** **0** **0** 0 0 1 0 1 **0**  $\rightarrow H = h_1h_2h_4h_8 = 1 \ 1 \ 1 \_$
  - Bits 8, 9, 10, 11, 12 should be even parity, and they are  
0 1 1 1 0 0 0 **0** **1** **0** **1** **0**  $\rightarrow H = h_1h_2h_4h_8 = 1 \ 1 \ 1 \ 0$
  - $1 + 2 + 4 = 7$  and the 7<sup>th</sup> bit is in error and can be corrected

- Example:

- How about this bit pattern: 0111001**1**1010?



# SECEDED

Single error correcting, double error detecting (SECEDED)

- Add an additional parity bit for the whole word ( $p_n$ )
  - Final code word = 011100101010 0 ←
  - This makes Hamming distance = 4 (so?)
- Decoding: let  $H = h_1h_2h_4h_8$  and  $h_n$  = parity of whole word
  - $H = 0$ ,  $h_n$  even  $\rightarrow$  no error
  - $H \neq 0$ ,  $h_n$  odd  $\rightarrow$  correctable single bit error (as in SEC)
  - $H = 0$ ,  $h_n$  odd  $\rightarrow$  error in  $p_n$  bit
  - $H \neq 0$ ,  $h_n$  even  $\rightarrow$  double error occurred
- Note: ECC DRAM uses SECDEC with 8 bits to protect each 64-bit data in memory



- Example 1: detect bit pattern = 011100001010 0
  - We have shown that  $H = h_1h_2h_4h_8 = 1\ 1\ 1\ 0$
  - In addition, the whole word is odd parity  $\rightarrow h_n = \text{odd}$
  - So,  $H \neq 0$ ,  $p_n$  odd  $\rightarrow$  correctable single bit error at bit 7
- Example 2: detect bit pattern = 0111000011010 0
  - 0 1 1 1 0 0 0 1 1 0 1 0  $\rightarrow H = h_1h_2h_4h_8 = 1\ \_\_\_\_$
  - 0 1 1 1 0 0 0 1 1 0 1 0  $\rightarrow H = h_1h_2h_4h_8 = 1\ 1\ \_\_\_\_$
  - 0 1 1 1 0 0 0 1 1 0 1 0  $\rightarrow H = h_1h_2h_4h_8 = 1\ 1\ 1\ \_\_\_\_$
  - 0 1 1 1 0 0 0 1 1 0 1 0  $\rightarrow H = h_1h_2h_4h_8 = 1\ 1\ 1\ 1\ \_\_\_\_$
  - The whole word is even parity  $\rightarrow h_n = \text{even}$
  - So,  $H \neq 0$ ,  $h_n$  even  $\rightarrow$  double errors

