CS 110
Computer Architecture

Dependability and RAID

Instructor:
Sören Schwertfeger

https://robotics.shanghaitech.edu.cn/courses/ca

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Review: I/O

- "Memory mapped I/O": Device control/data registers mapped to CPU address space
- CPU synchronizes with I/O device:
  - Polling
  - Interrupts
- "Programmed I/O":
  - CPU execs lw/sw instructions for all data movement to/from devices
  - CPU spends time doing 2 things:
    1. Getting data from device to main memory
    2. Using data to compute

Working with real devices

- "Memory mapped I/O": Device control/data registers mapped to CPU address space
- CPU synchronizes with I/O device:
  - Polling
  - Interrupts
- ~~"Programmed I/O":~~ **DMA**
  - ~~CPU execs lw/sw instructions for all data movement to/from devices~~
  - CPU spends time doing ~~2 things~~:
    1. ~~Getting data from device to main memory~~
    2. Using data to compute

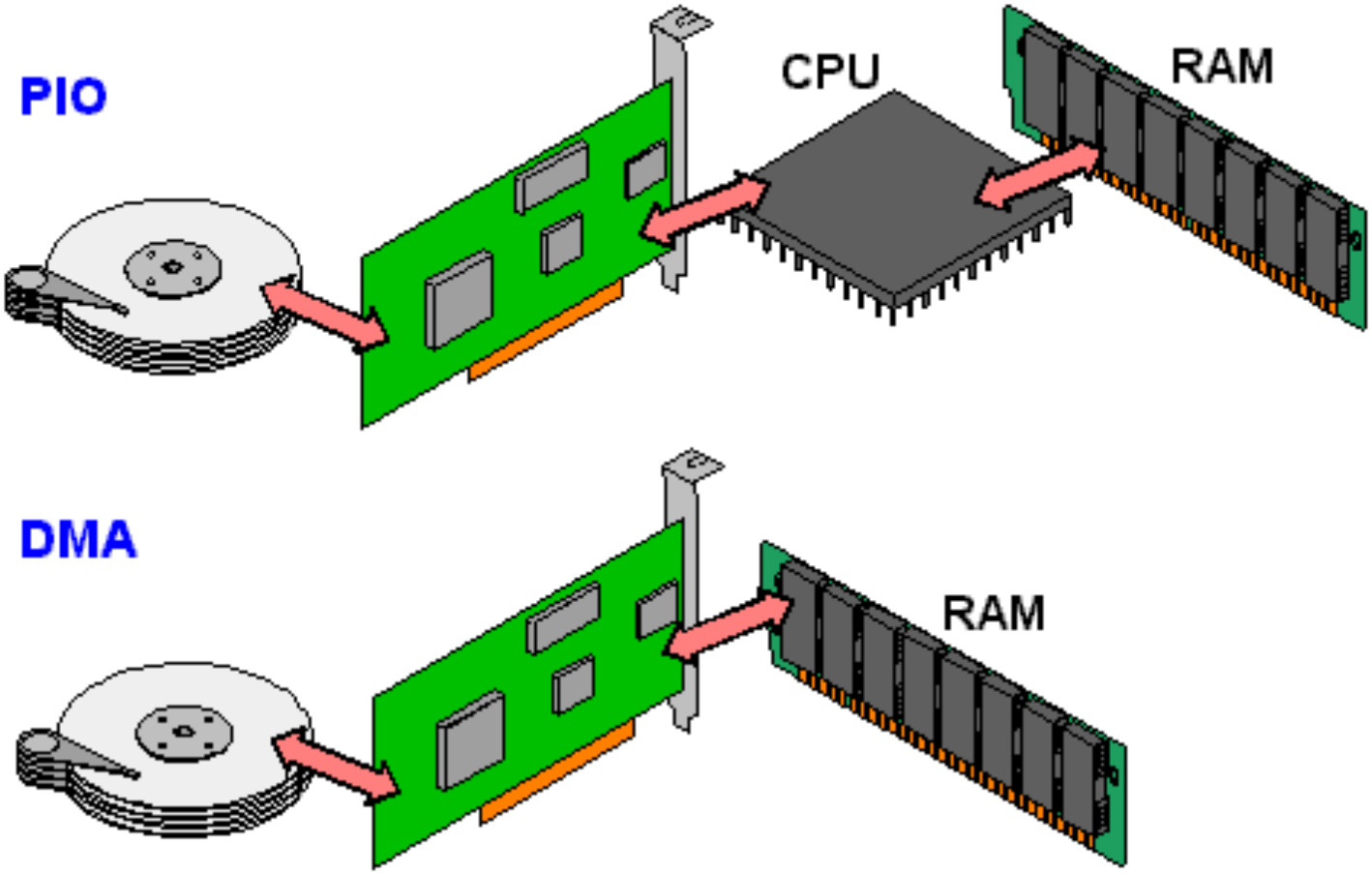
Agenda

- **Direct Memory Access (DMA)**
- Disks
- Dependability

What's wrong with Programmed I/O?

- Not ideal because …
  1. CPU has to execute all transfers, could be doing other work
  2. Device speeds don't align well with CPU speeds
  3. Energy cost of using beefy general-purpose CPU where simpler hardware would suffice
- Until now CPU has sole control of main memory

PIO vs. DMA



Direct Memory Access (DMA)

- Allows I/O devices to directly read/write main memory
- New Hardware: the DMA Engine
- DMA engine contains registers written by CPU:
  - Memory address to place data
  - # of bytes
  - I/O device #, direction of transfer
  - unit of transfer, amount to transfer per burst

Operation of a DMA Transfer

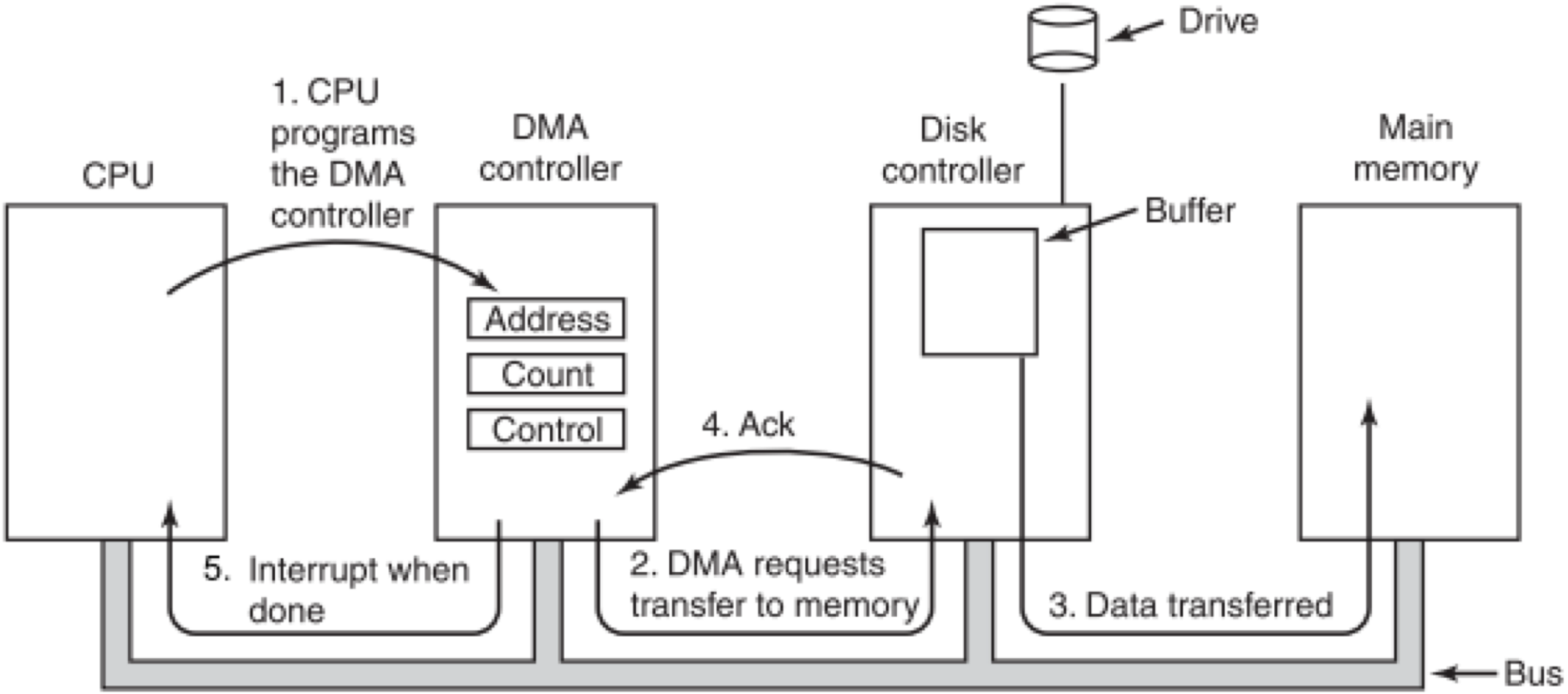


Figure 5-4. Operation of a DMA transfer.

[From Section 5.1.4 Direct Memory Access in *Modern Operating Systems* by Andrew S. Tanenbaum, Herbert Bos, 2014]

DMA: Incoming Data

1. Receive interrupt from device
2. CPU takes interrupt, begins transfer
   - Instructs DMA engine/device to place data @ certain address
3. Device/DMA engine handle the transfer
   - CPU is free to execute other things
4. Upon completion, Device/DMA engine interrupt the CPU again

DMA: Outgoing Data

1. CPU decides to initiate transfer, confirms that external device is ready
2. CPU begins transfer
   - Instructs DMA engine/device that data is available @ certain address
3. Device/DMA engine handle the transfer
   - CPU is free to execute other things
4. Device/DMA engine interrupt the CPU again to signal completion

DMA: Some new problems

- Where in the memory hierarchy do we plug in the DMA engine? Two extremes:
  - Between L1 and CPU:
    - Pro: Free coherency
    - Con: Trash the CPU's working set with transferred data
  - Between Last-level cache and main memory:
    - Pro: Don't mess with caches
    - Con: Need to explicitly manage coherency

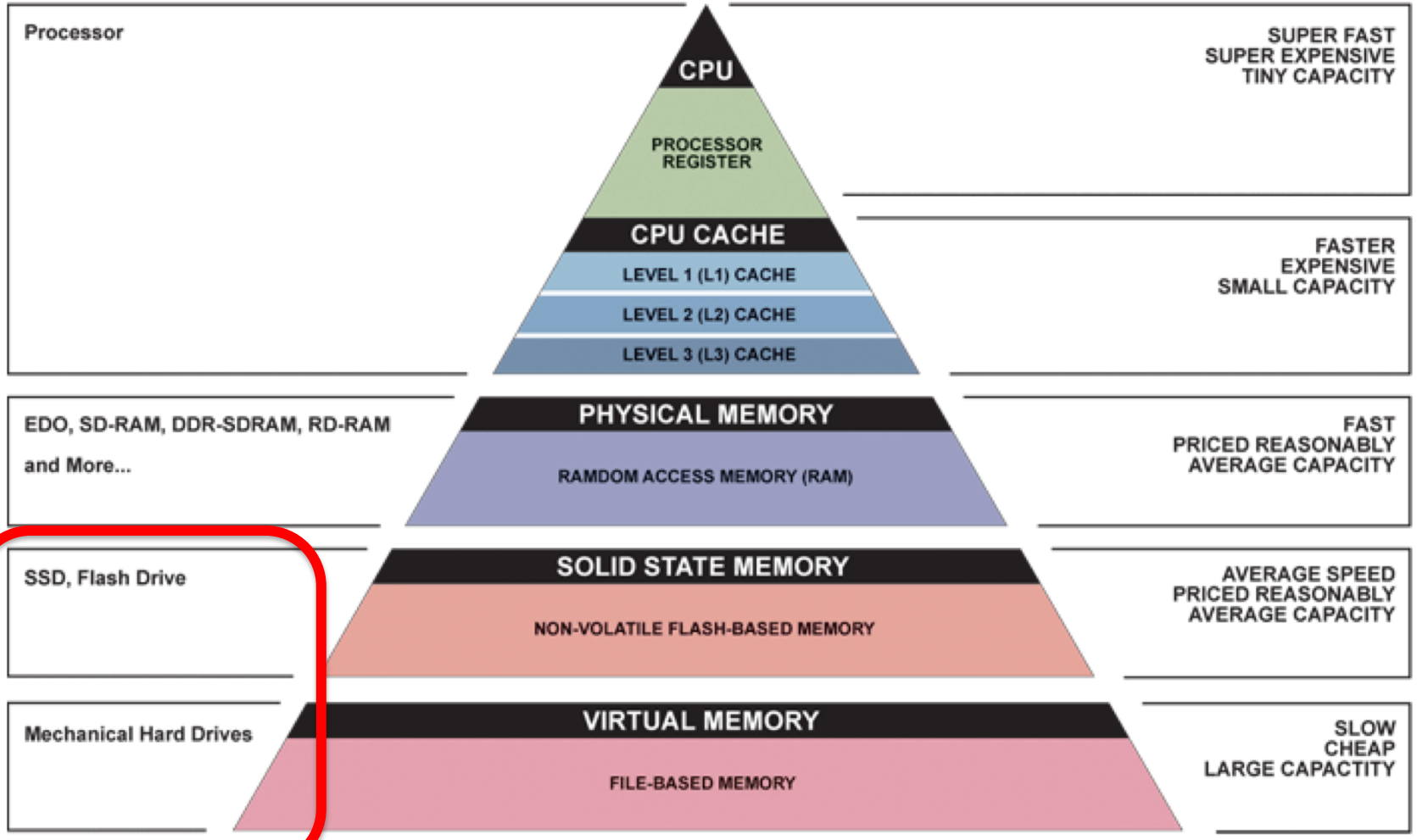
DMA: Some new problems

- How do we arbitrate between CPU and DMA Engine/Device access to memory? Three options:
  - Burst Mode
    - Start transfer of data block, CPU cannot access memory in the meantime
  - Cycle Stealing Mode
    - DMA engine transfers a byte, releases control, then repeats - interleaves processor/DMA engine accesses
  - Transparent Mode
    - DMA transfer only occurs when CPU is not using the system bus

Agenda

- Direct Memory Access (DMA)
- **Disks**
- Dependability

Computer Memory Hierarchy



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

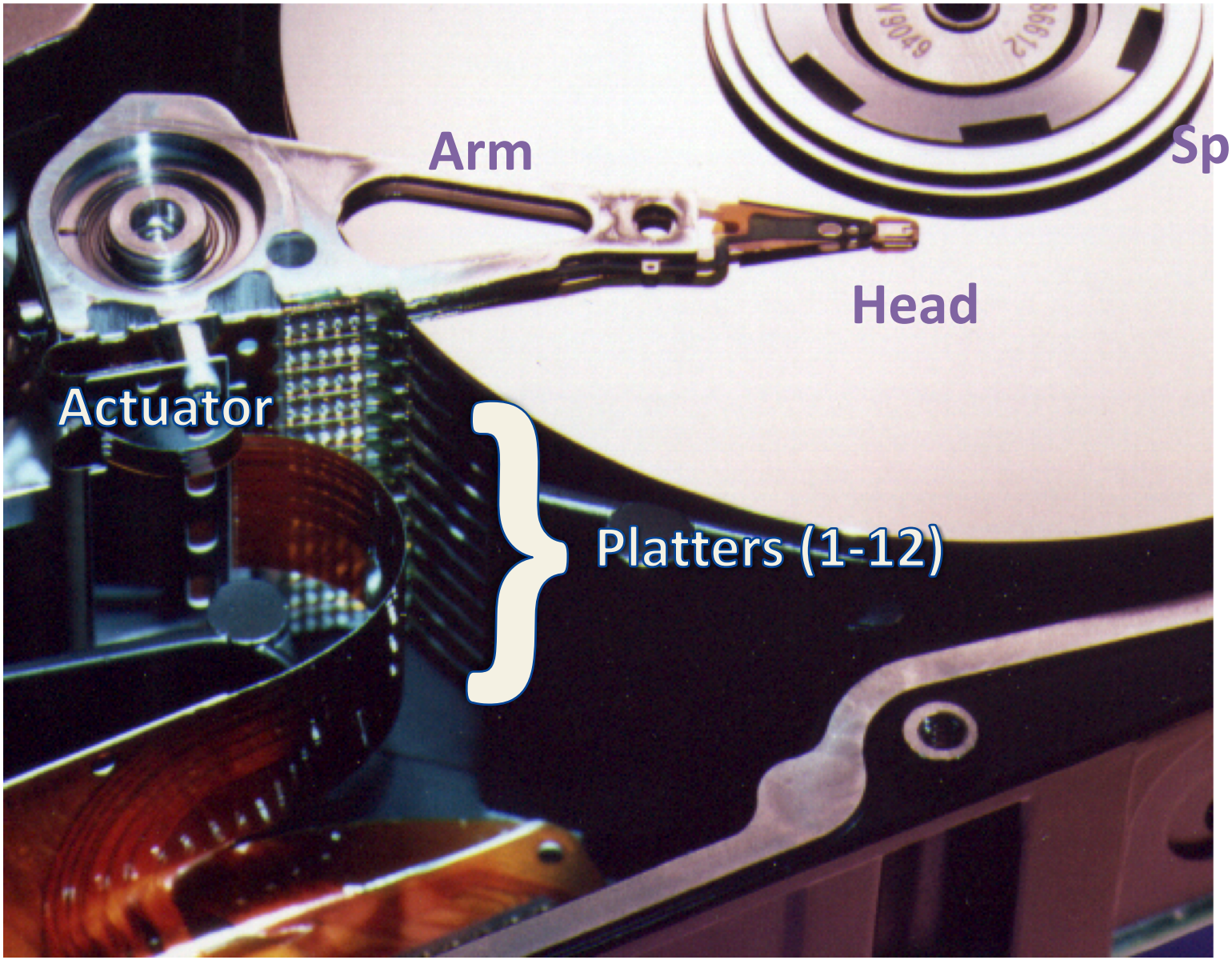
Today

Magnetic Disk – common I/O device

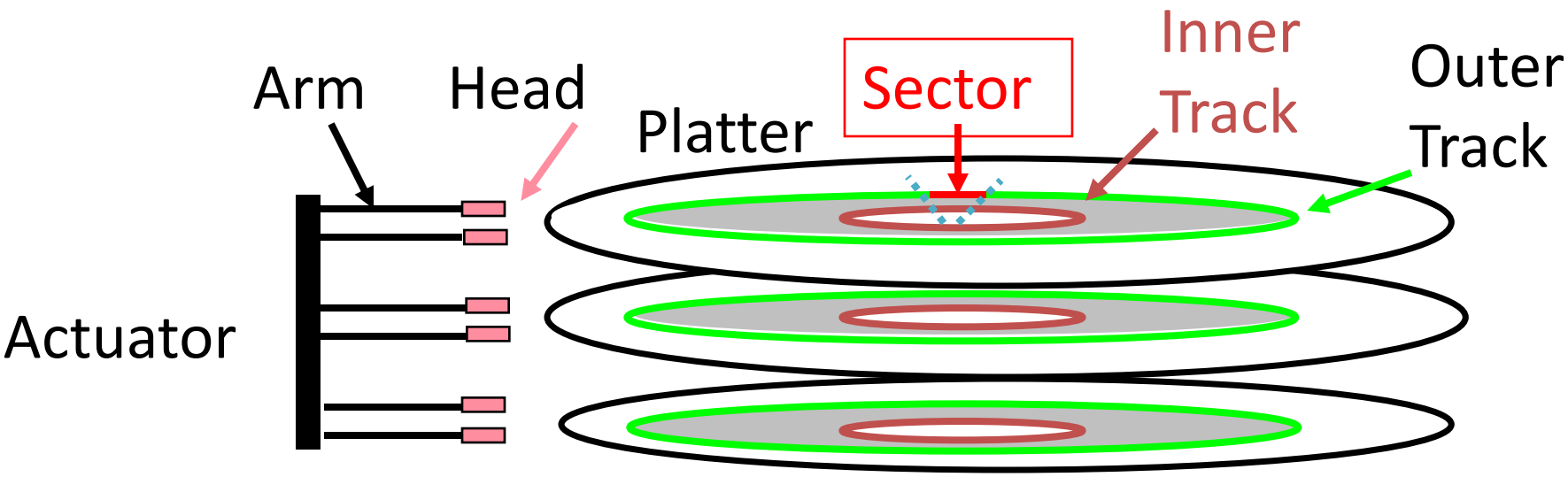
- A kind of computer memory
  - Information stored by magnetizing ferrite material on surface of rotating disk
    - similar to tape recorder except digital rather than analog data
- A type of non-volatile storage
  - retains its value without applying power to disk.
- Magnetic Disk
  1. Hard Disk Drives (HDD) – faster, more dense, non-removable.

- Purpose in computer systems (Hard Drive):
  1. Working file system + long-term backup for files
  2. Secondary "backing store" for main-memory. Large, inexpensive, slow level in the memory hierarchy (virtual memory)

Photo of Disk Head, Arm, Actuator



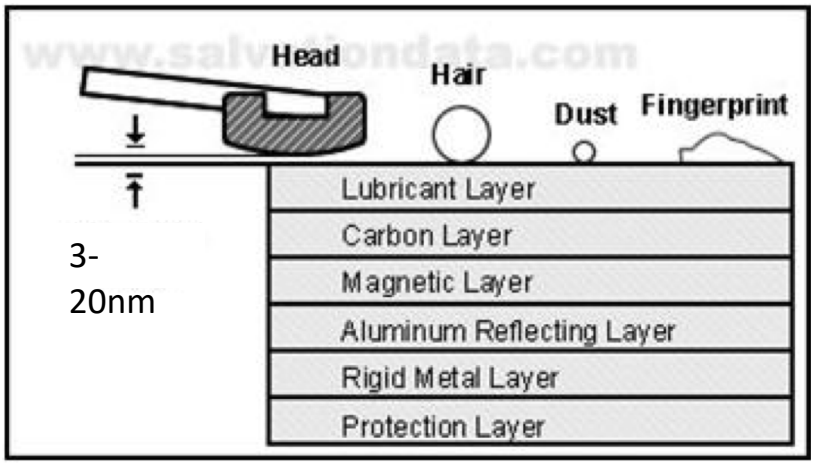
Disk Device Terminology

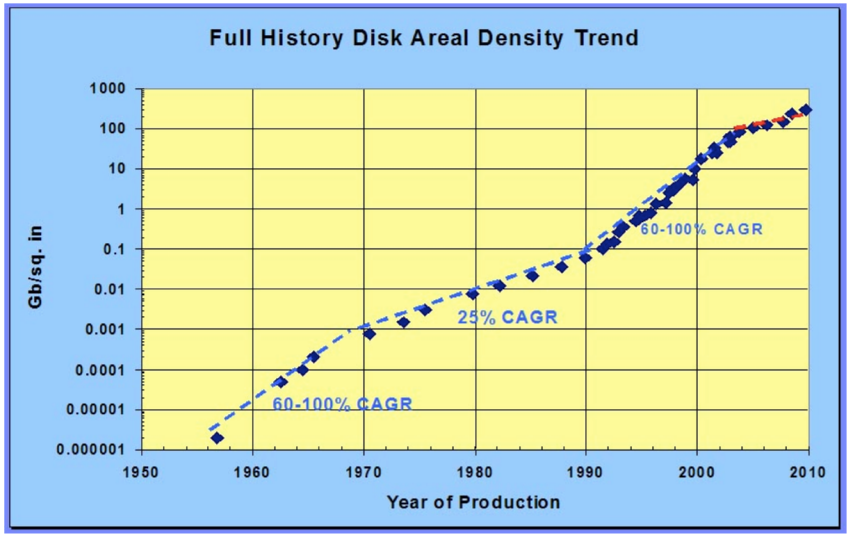


- Several platters, with information recorded magnetically on both surfaces (usually)
- **Bits recorded in tracks, which in turn divided into sectors (e.g., 512 Bytes)**
- **Actuator moves head (end of arm) over track ("seek"), wait for sector rotate under head, then read or write**

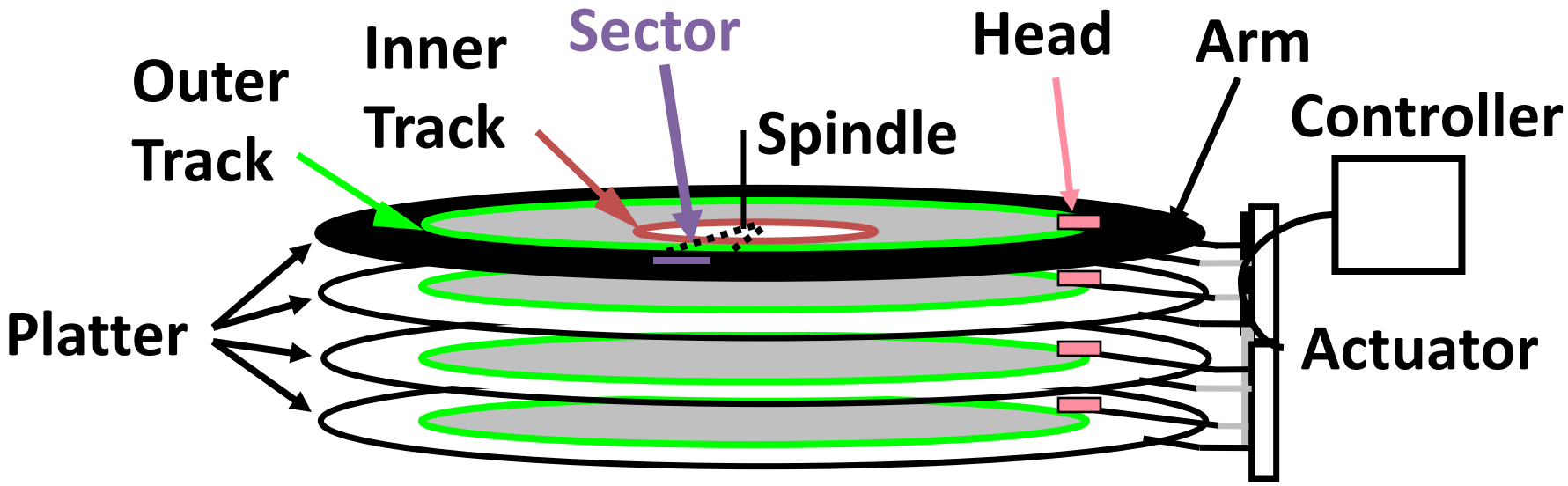
Hard Drives are Sealed. Why?

- The closer the head to the disk, the smaller the “spot size” and thus the denser the recording.
  - Measured in Gbit/in^2
  - ~900 Gbit/in^2 is state of the art
  - Started out at 2 Kbit/in^2
  - ~450,000,000x improvement in ~60 years
- Disks are sealed to keep the dust out.
  - Heads are designed to “fly” at around 3-20nm above the surface of the disk.
  - 99.999% of the head/arm weight is supported by the air bearing force (air cushion) developed between the disk and the head.





Disk Device Performance (1/2)



- **Disk Access Time = Seek Time + Rotation Time + Transfer Time + Controller Overhead**
  - Seek Time = time to position the head assembly at the proper cylinder
  - Rotation Time = time for the disk to rotate to the point where the first sectors of the block to access reach the head
  - Transfer Time = time taken by the sectors of the block and any gaps between them to rotate past the head

Disk Device Performance (2/2)

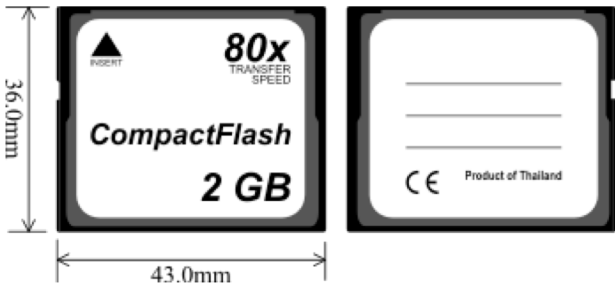
- Average values to plug into the formula:
- Rotation Time: Average distance of sector from head?
  - 1/2 time of a rotation
    - 7200 Revolutions Per Minute => 120 Rev/sec
    - 1 revolution = 1/120 sec => 8.33 milliseconds
    - 1/2 rotation (revolution) => 4.17 ms
- Seek time: Average no. tracks to move arm?
  - Number of tracks/ 3
  - Then, seek time = number of tracks moved × time to move across one track

But wait!

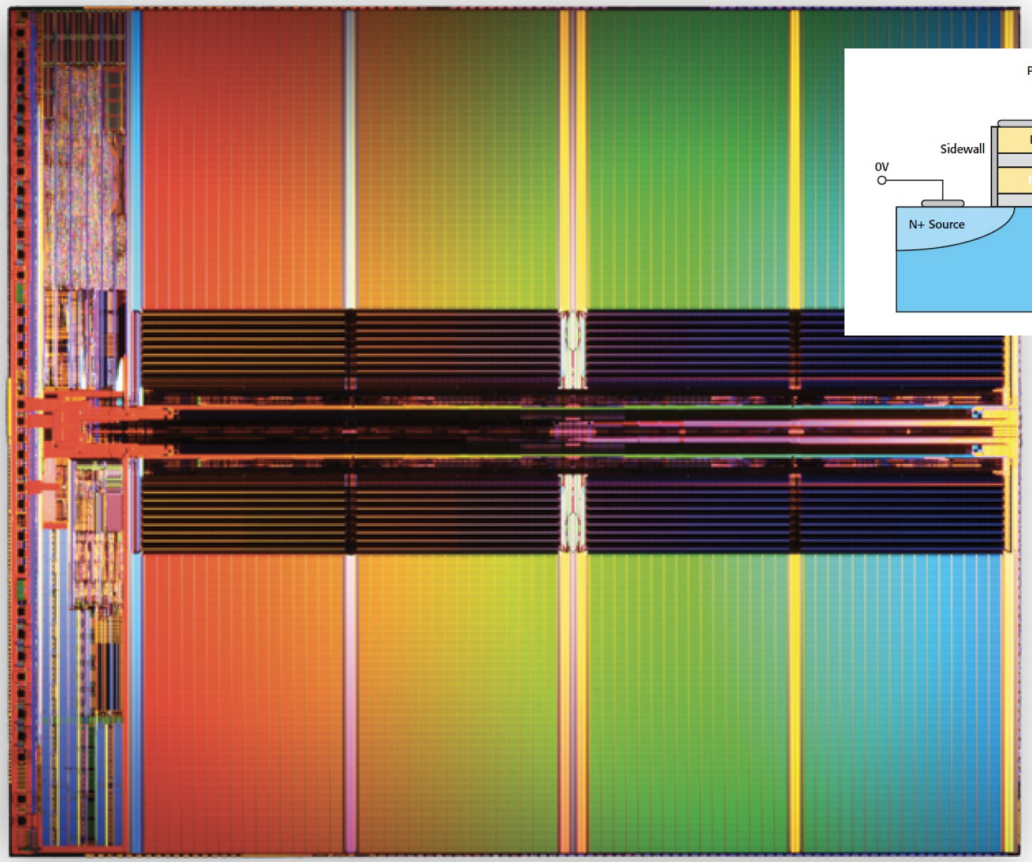
- Performance estimates are different in practice:
- Many disks have on-disk caches, which are completely hidden from the outside world
  - Previous formula completely replaced with on-disk cache access time

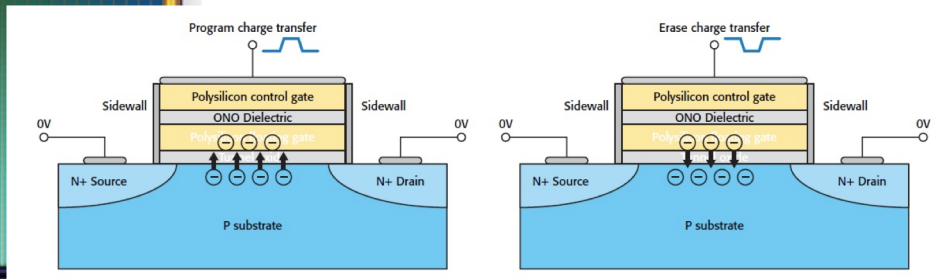
Where does Flash memory come in?

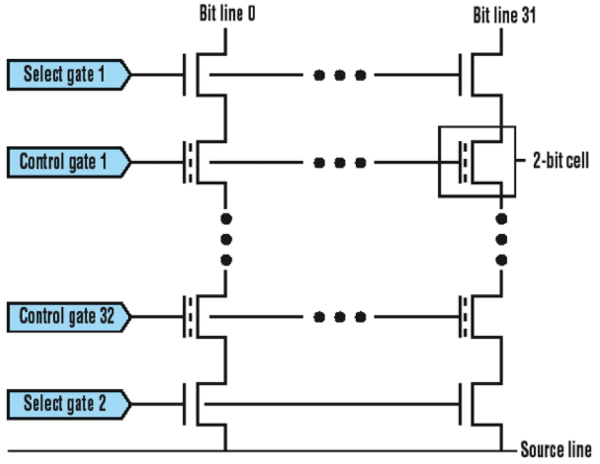
- ~15 years ago: Microdrives and Flash memory (e.g., CompactFlash) went head-to-head
  - Both non-volatile (retains contents without power supply)
  - Flash benefits: lower power, no crashes (no moving parts, need to spin µdrives up/down)
  - Disk cost = fixed cost of motor + arm mechanics, but actual magnetic media cost very low
  - Flash cost = most cost/bit of flash chips
  - Over time, cost/bit of flash came down, became cost competitive



Flash Memory / SSD Technology







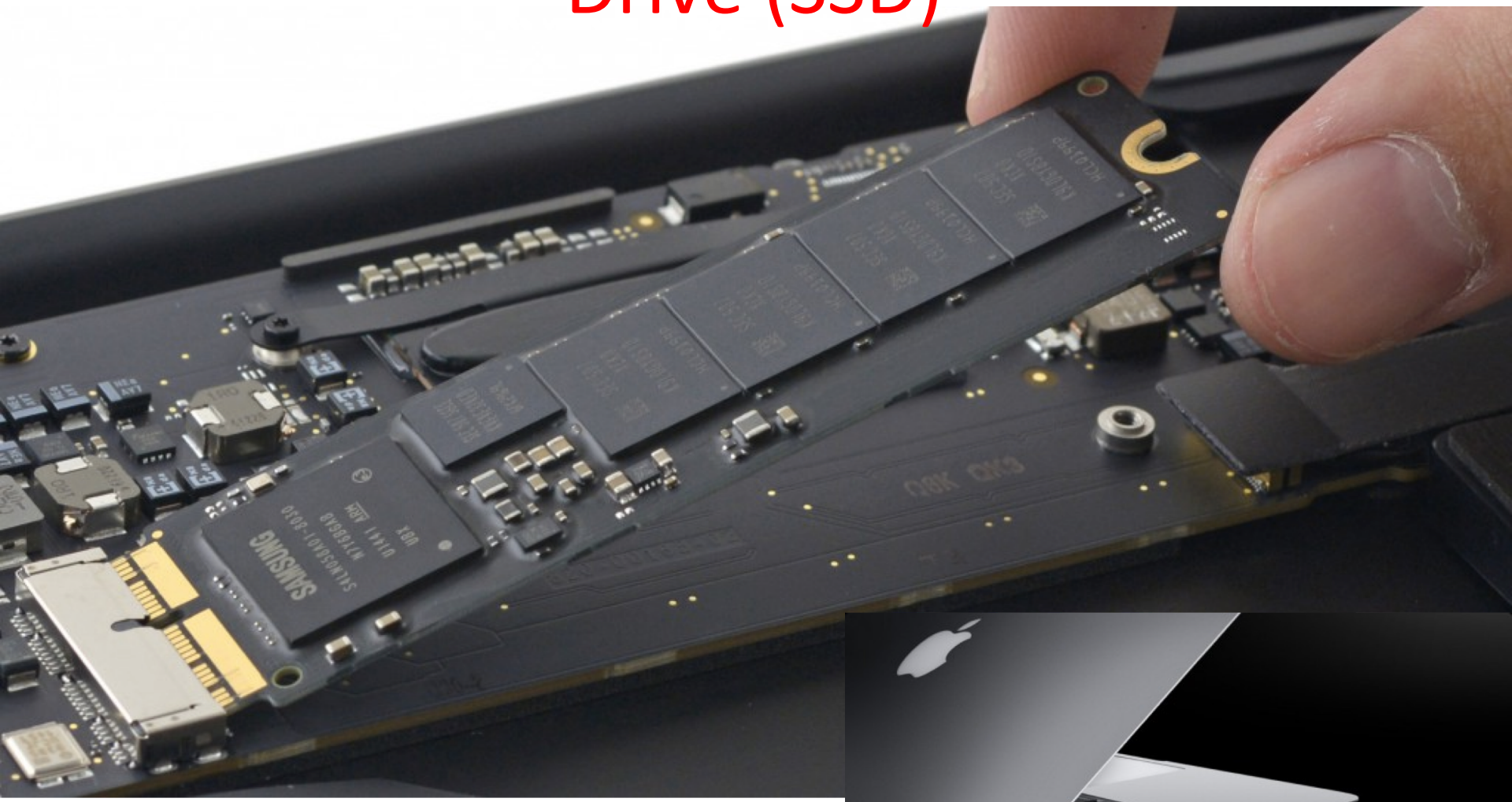
2. Micron's triple-level cell (TLC) flash memory stores 3 bits of data in each transistor.

In the basic functional block used in multilevel NAND flash memories, 32 rows of bit lines and 32 control-gate lines form a building block that's repeated many times to form the memory array. The select gate lines are used with the control gate lines to control access to the array.

- NMOS transistor with an additional conductor between gate and source/drain which "traps" electrons. The presence/absence is a 1 or 0
- Memory cells can withstand a limited number of program-erase cycles. Controllers use a technique called *wear leveling* to distribute writes as evenly as possible across all the flash blocks in the SSD.

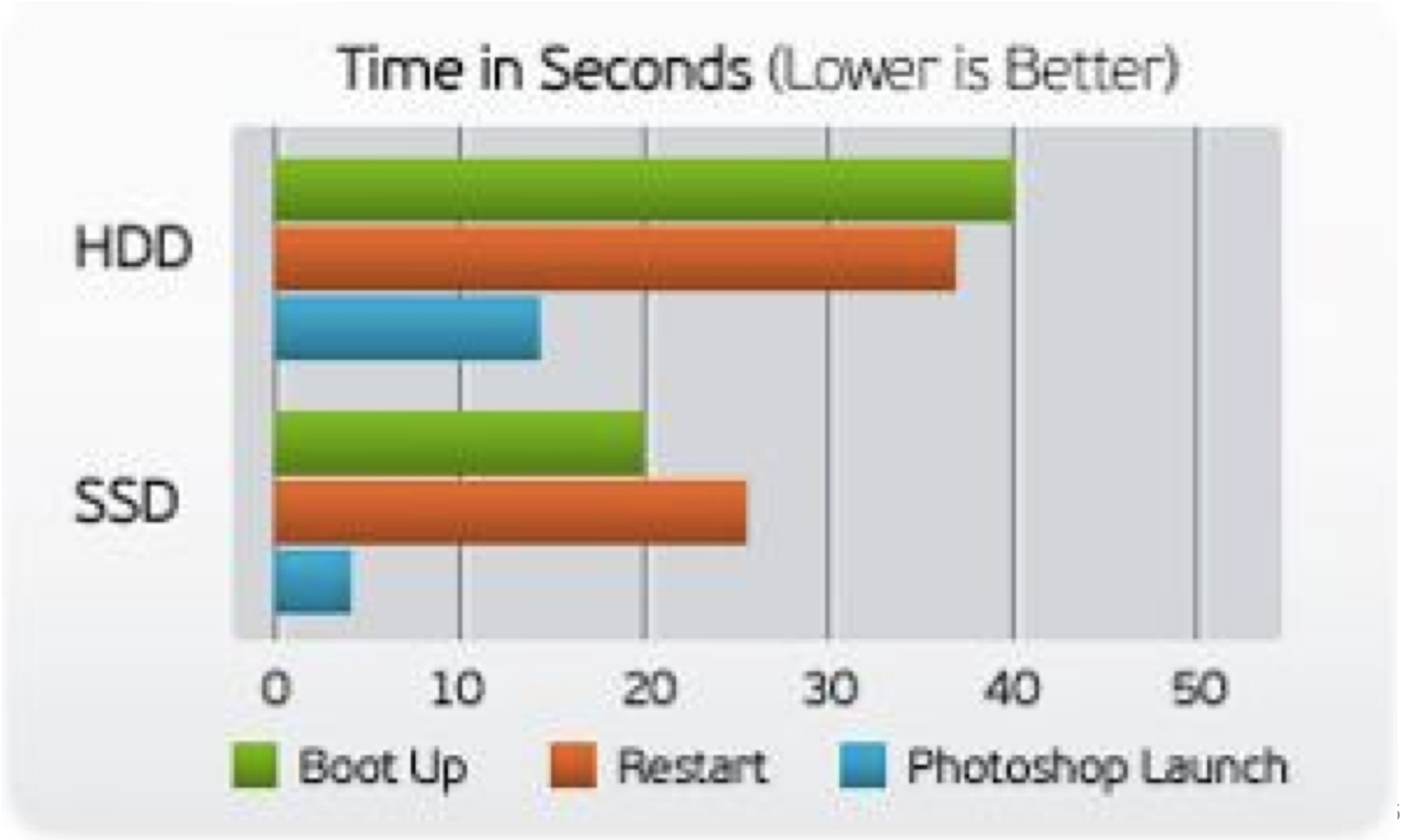
Flash Memory in Smart Phones

Flash Memory in Laptops – Solid State Drive (SSD)



capacities up to 4TB

HDD vs SSD speed



Question

- We have the following disk:
  - 15000 Tracks, 1 ms to cross 1000 Tracks
  - 15000 RPM = 4 ms per rotation
  - Want to copy 1 MB, transfer rate of 1000 MB/s
  - 1 ms controller processing time
- What is the access time using our model?

Disk Access Time = Seek Time + Rotation Time + Transfer Time + Controller Processing Time

A	B	C	D	E
10.5 ms	9 ms	8.5 ms	11.4 ms	12 ms

Question

- We have the following disk:
  - 15000 Cylinders, 1 ms to cross 1000 Cylinders
  - 15000 RPM = 4 ms per rotation
  - Want to copy 1 MB, transfer rate of 1000 MB/s
  - 1 ms controller processing time
- What is the access time?

Seek = # cylinders/3 * time = 15000/3 * 1ms/1000 cylinders = 5ms

Rotation = time for ½ rotation = 4 ms / 2 = 2 ms

Transfer = Size / transfer rate = 1 MB / (1000 MB/s) = 1 ms

Controller = 1 ms

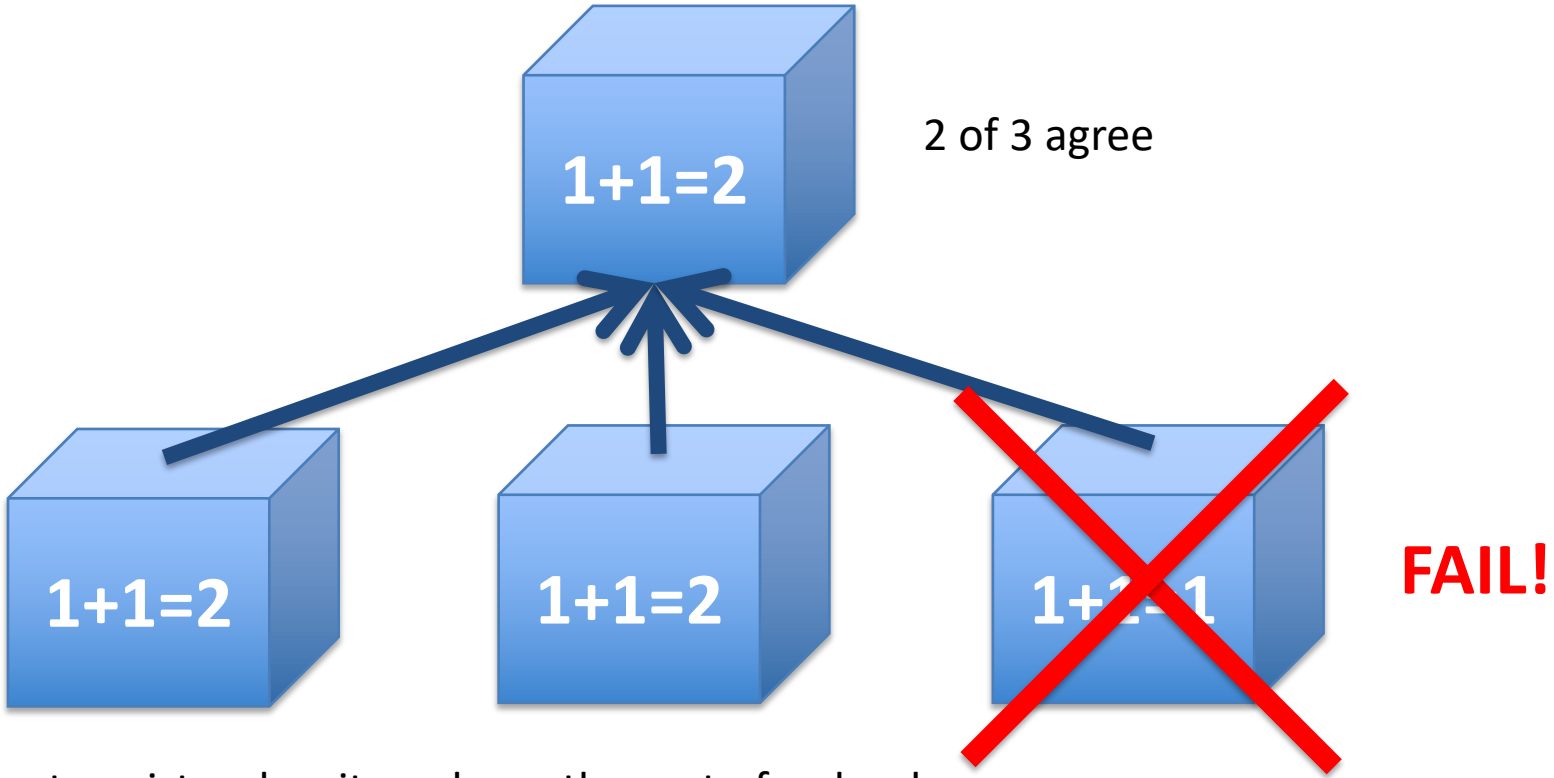
Total = 5 + 2 + 1 + 1 = 9 ms

Agenda

- Direct Memory Access (DMA)
- Disks
- **Dependability**

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail

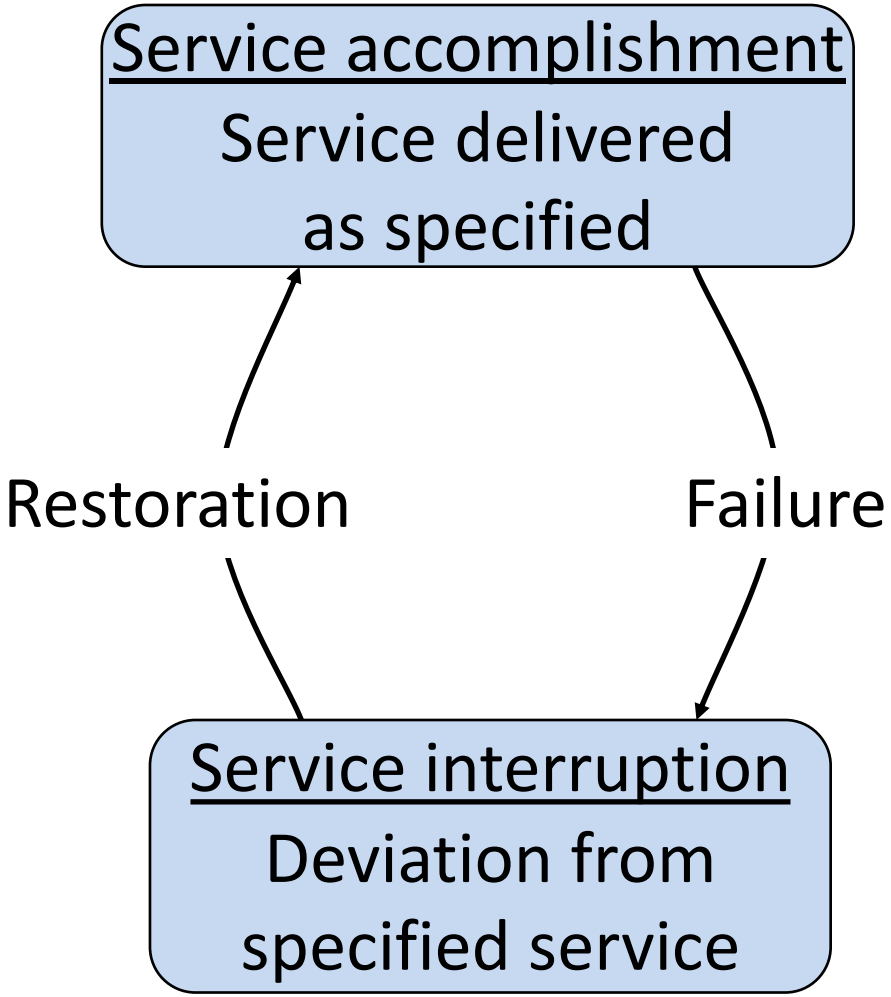


Increasing transistor density reduces the cost of redundancy

Great Idea #6: Dependability via Redundancy

- Applies to everything from datacenters to memory
  - Redundant datacenters so that can lose 1 datacenter but Internet service stays online
  - Redundant routes so can lose nodes but Internet doesn't fail
  - Redundant disks so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
  - Redundant memory bits of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

Dependability



- Fault: failure of a component
  - May or may not lead to system failure

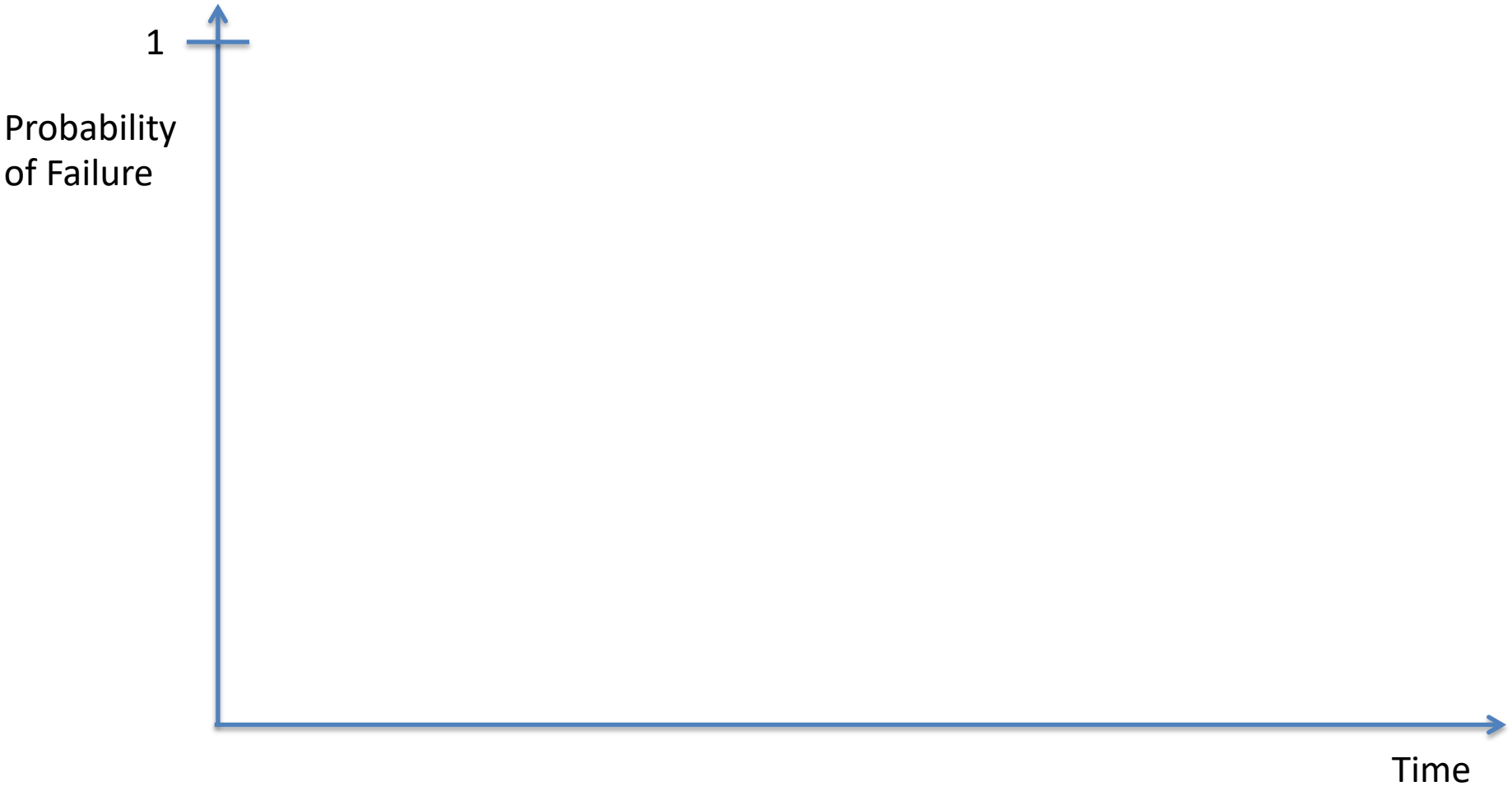
Dependability via Redundancy: Time vs. Space

- *Spatial Redundancy* – replicated data or check information or hardware to handle hard and soft (transient) failures
- *Temporal Redundancy* – redundancy in time (retry) to handle soft (transient) failures

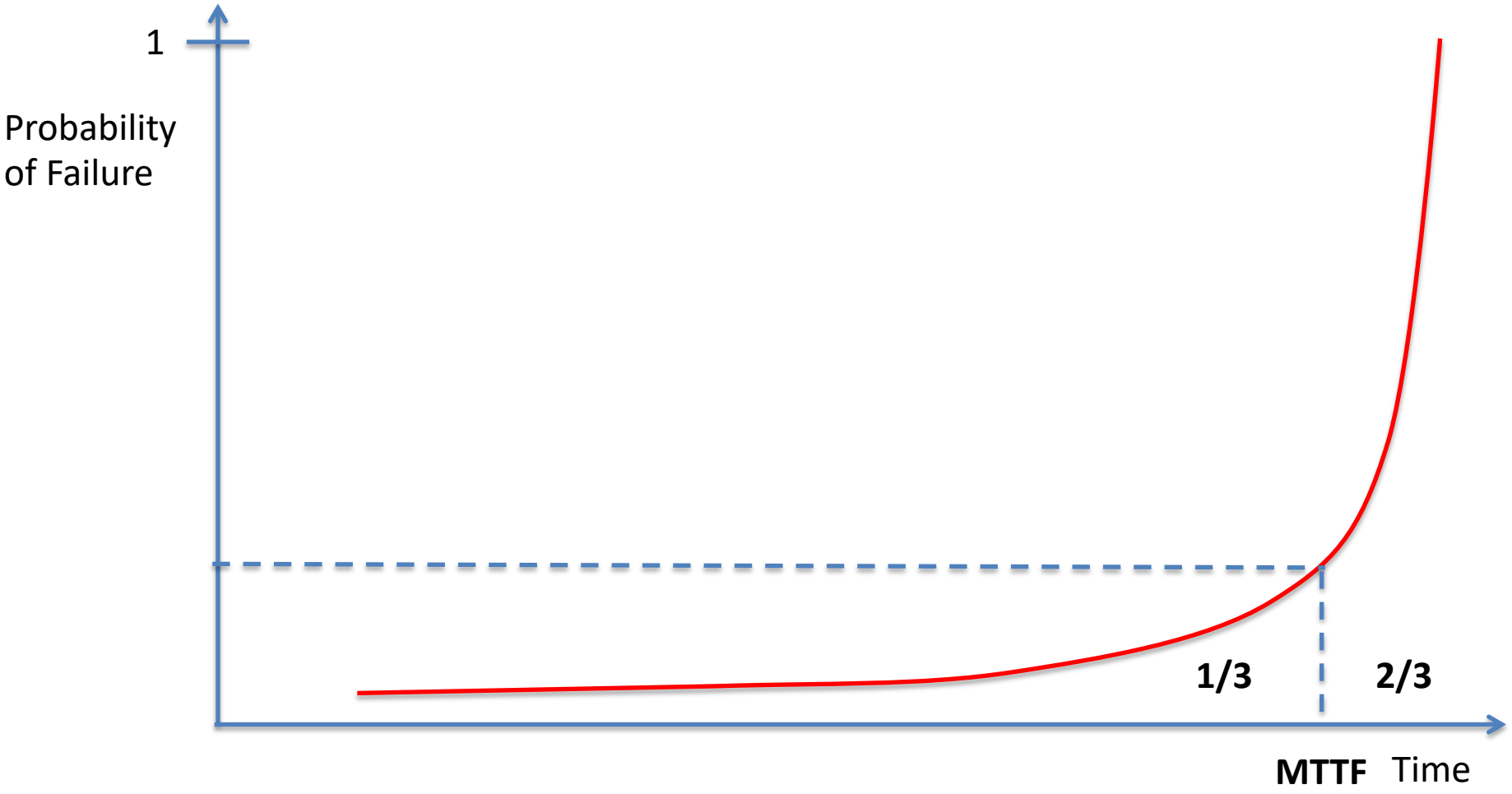
Dependability Measures

- Reliability: Mean Time To Failure (MTTF)
- Service interruption: Mean Time To Repair (MTTR)
- Mean time between failures (MTBF)
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: More reliable hardware/software + Fault Tolerance
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Understanding MTTF



Understanding MTTF



Availability Measures

- Availability = MTTF / (MTTF + MTTR) as %
  - MTTF, MTBF usually measured in hours
- Since hope rarely down, shorthand is "number of 9s of availability per year"
- 1 nine: 90% => 36 days of repair/year
- 2 nines: 99% => 3.6 days of repair/year
- 3 nines: 99.9% => 526 minutes of repair/year
- 4 nines: 99.99% => 53 minutes of repair/year
- 5 nines: 99.999% => 5 minutes of repair/year

Reliability Measures

- Another is average number of failures per year: Annualized Failure Rate (AFR)
  - E.g., 1000 disks with 100,000 hour MTTF
  - 365 days * 24 hours = 8760 hours
  - (1000 disks * 8760 hrs/year) / 100,000 = 87.6 failed disks per year on average
  - 87.6/1000 = 8.76% annual failure rate
- Google's 2007 study* found that actual AFRs for individual drives ranged from 1.7% for first year drives to over 8.6% for three-year old drives

research.google**.com/archive/disk_failures.pdf*

Dependability Design Principle

- Design Principle: No single points of failure
  - "Chain is only as strong as its weakest link"
- Dependability Corollary of Amdahl's Law
  - Doesn't matter how dependable you make one portion of system
  - Dependability limited by part you do not improve

Error Detection/ Correction Codes

- Memory systems generate errors (accidentally flipped-bits)
  - DRAMs store very little charge per bit
  - "Soft" errors occur occasionally when cells are struck by alpha particles or other environmental upsets
  - "Hard" errors can occur when chips permanently fail
  - Problem gets worse as memories get denser and larger
- Memories protected against failures with EDC/ECC
- Extra bits are added to each data-word
  - Used to detect and/or correct faults in the memory system
  - Each data word value mapped to unique *code word*
  - A fault changes valid code word to invalid one, which can be detected

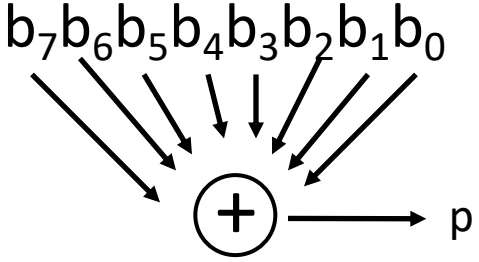
Block Code Principles

- Hamming distance = difference in # of bits
- p = 011011, q = 001111, Ham. distance (p,q) = 2
- p = 011011,
  q = 110001,
  distance (p,q) = ?
- Can think of extra bits as creating a code with the data
- What if minimum distance between members of code is 2 and get a 1-bit error?

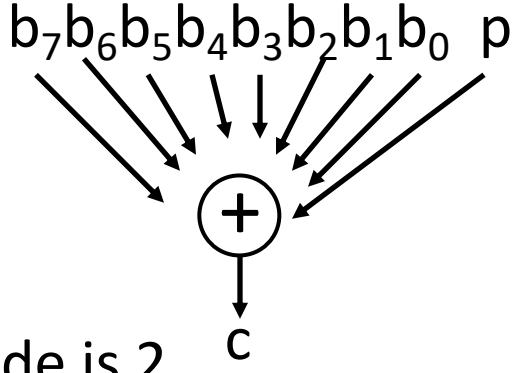
Richard Hamming, 1915-98
Turing Award Winner

Parity: Simple Error-Detection Coding

- Each data value, before it is written to memory is "tagged" with an extra bit to force the stored word to have *even parity*:



- Each word, as it is read from memory is "checked" by finding its parity (including the parity bit).



- Minimum Hamming distance of parity code is 2
- A non-zero parity check indicates an error occurred:
  - 2 errors (on different bits) are not detected
  - nor any even number of errors, just odd numbers of errors are detected

Parity Example

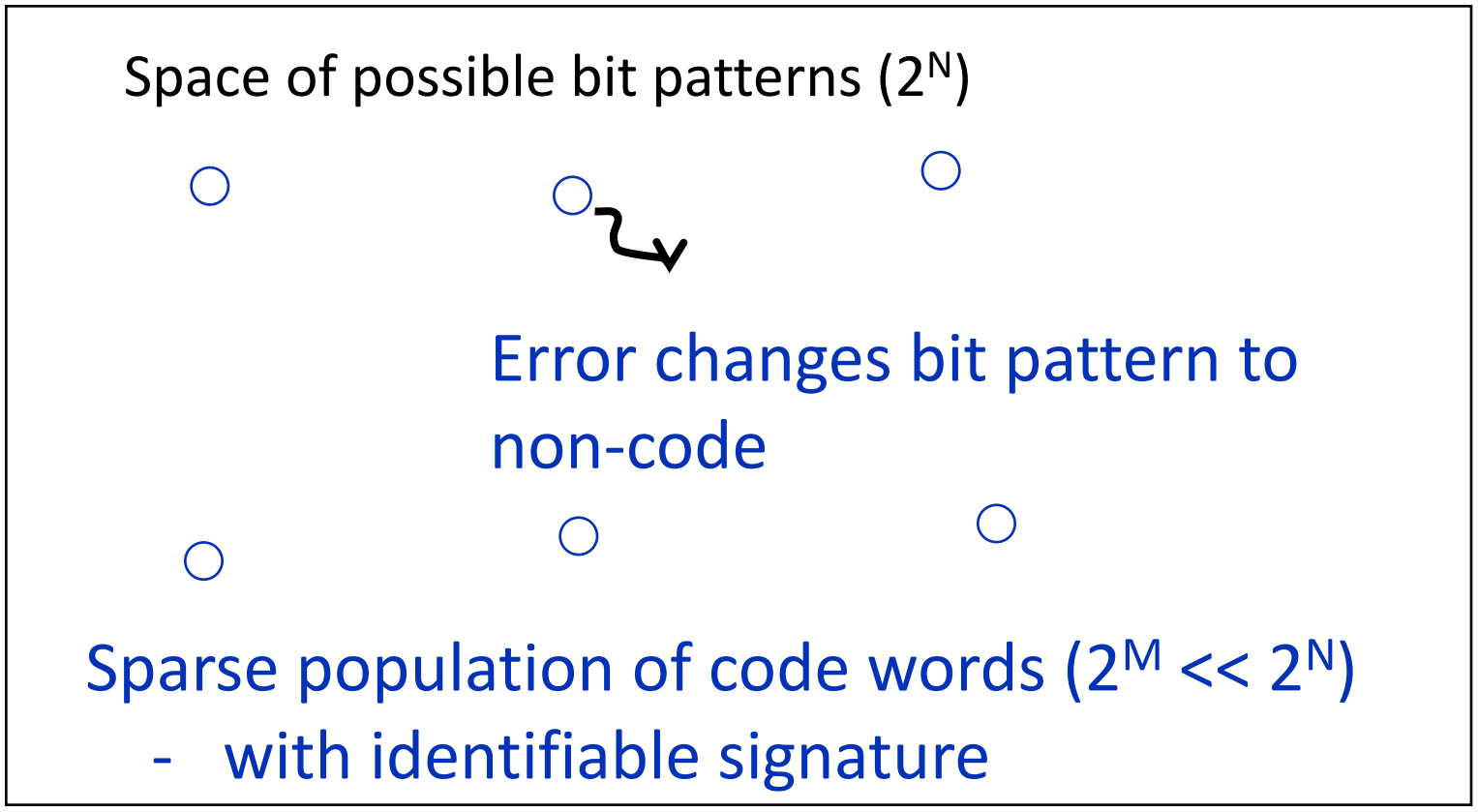
- Data 0101 0101
- 4 ones, even parity now
- Write to memory:
  0101 0101 0
  to keep parity even
- Data 0101 0111
- 5 ones, odd parity now
- Write to memory:
  0101 0111 1
  to make parity even

- Read from memory
  0101 0101 0
- 4 ones => even parity, so no error
- Read from memory
  1101 0101 0
- 5 ones => odd parity, so error
- What if error in parity bit?

Suppose Want to Correct 1 Error?

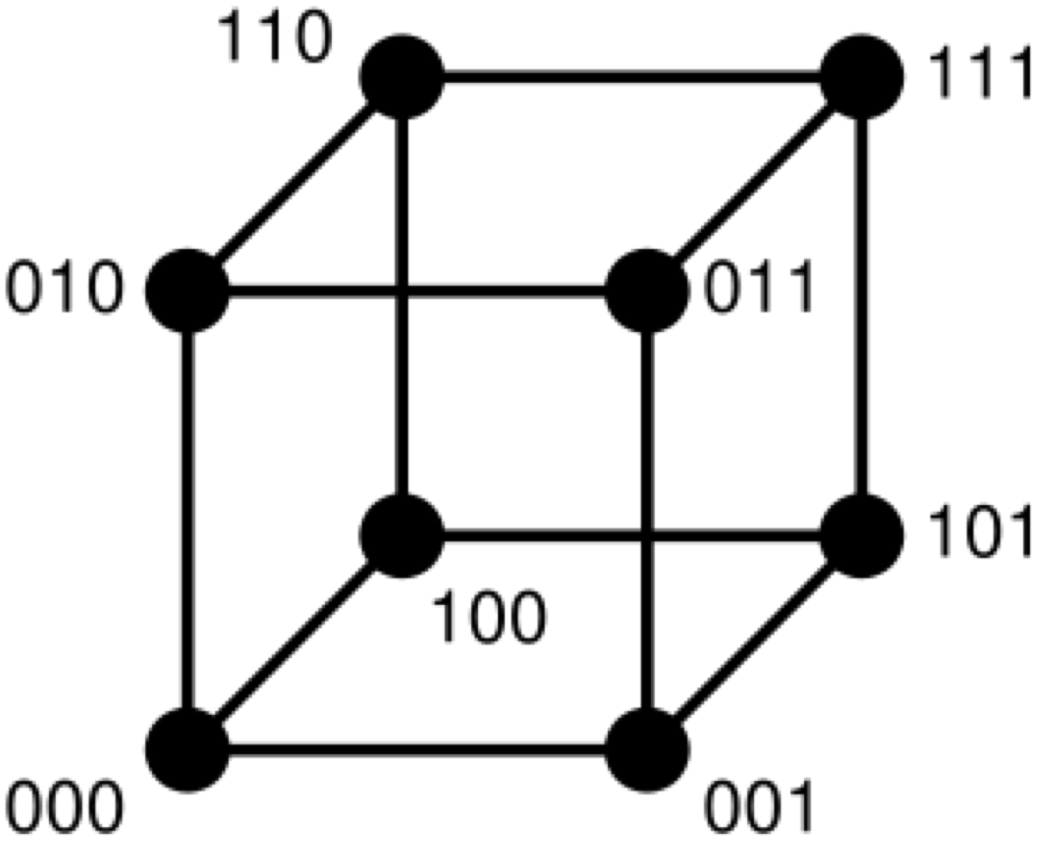
- Richard Hamming came up with simple to understand mapping to allow Error Correction at minimum distance of 3
  - Single error correction, double error detection
- Called "Hamming ECC"
  - Worked weekends on relay computer with unreliable card reader, frustrated with manual restarting
  - Got interested in error correction; published 1950
  - R. W. Hamming, "Error Detecting and Correcting Codes," *The Bell System Technical Journal*, Vol. XXVI, No 2 (April 1950) pp 147-160.

Detecting/Correcting Code Concept



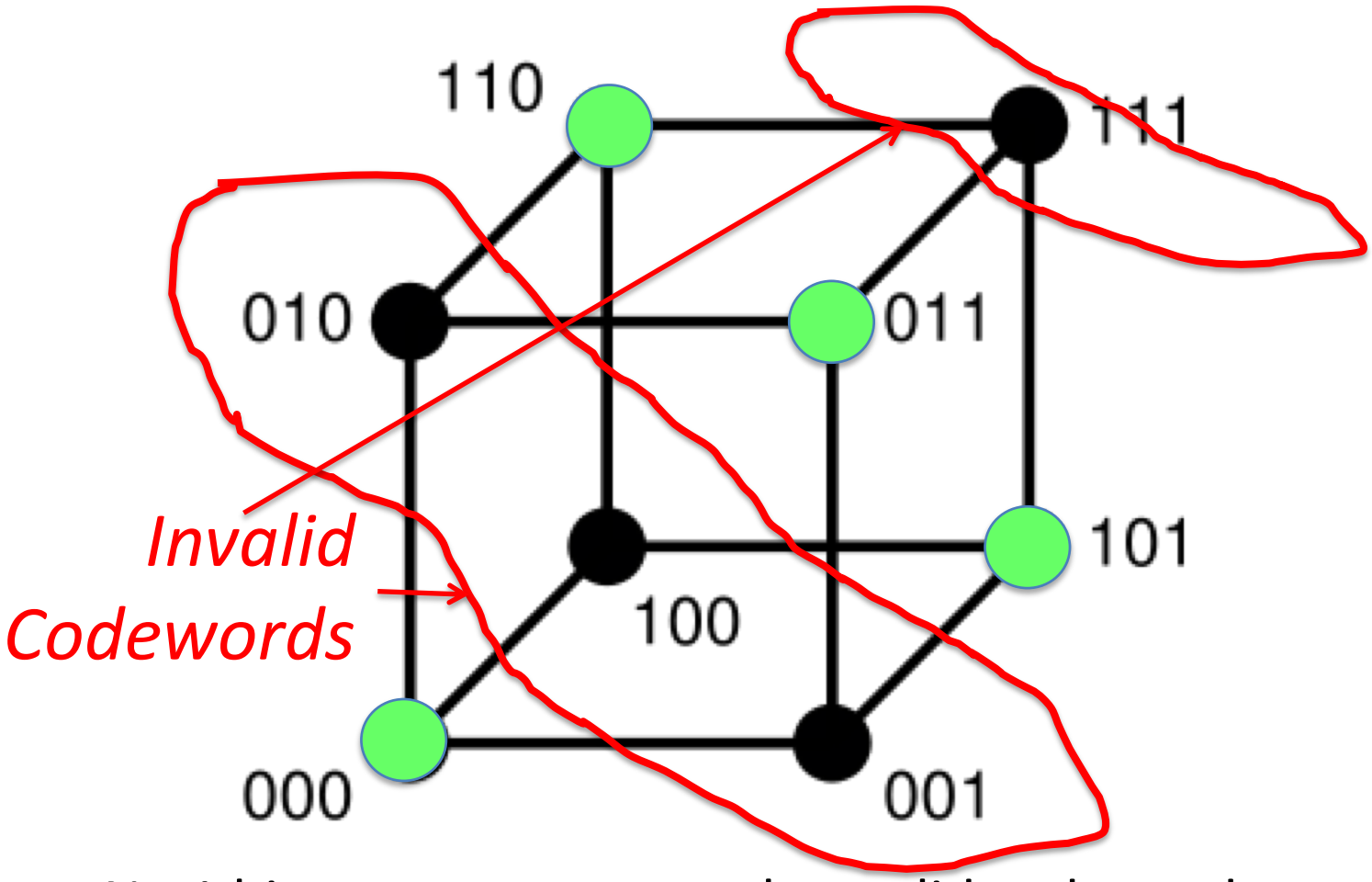
- Detection: bit pattern fails codeword check
- Correction: map to nearest valid code word

Hamming Distance: 8 code words



Hamming Distance 2: Detection

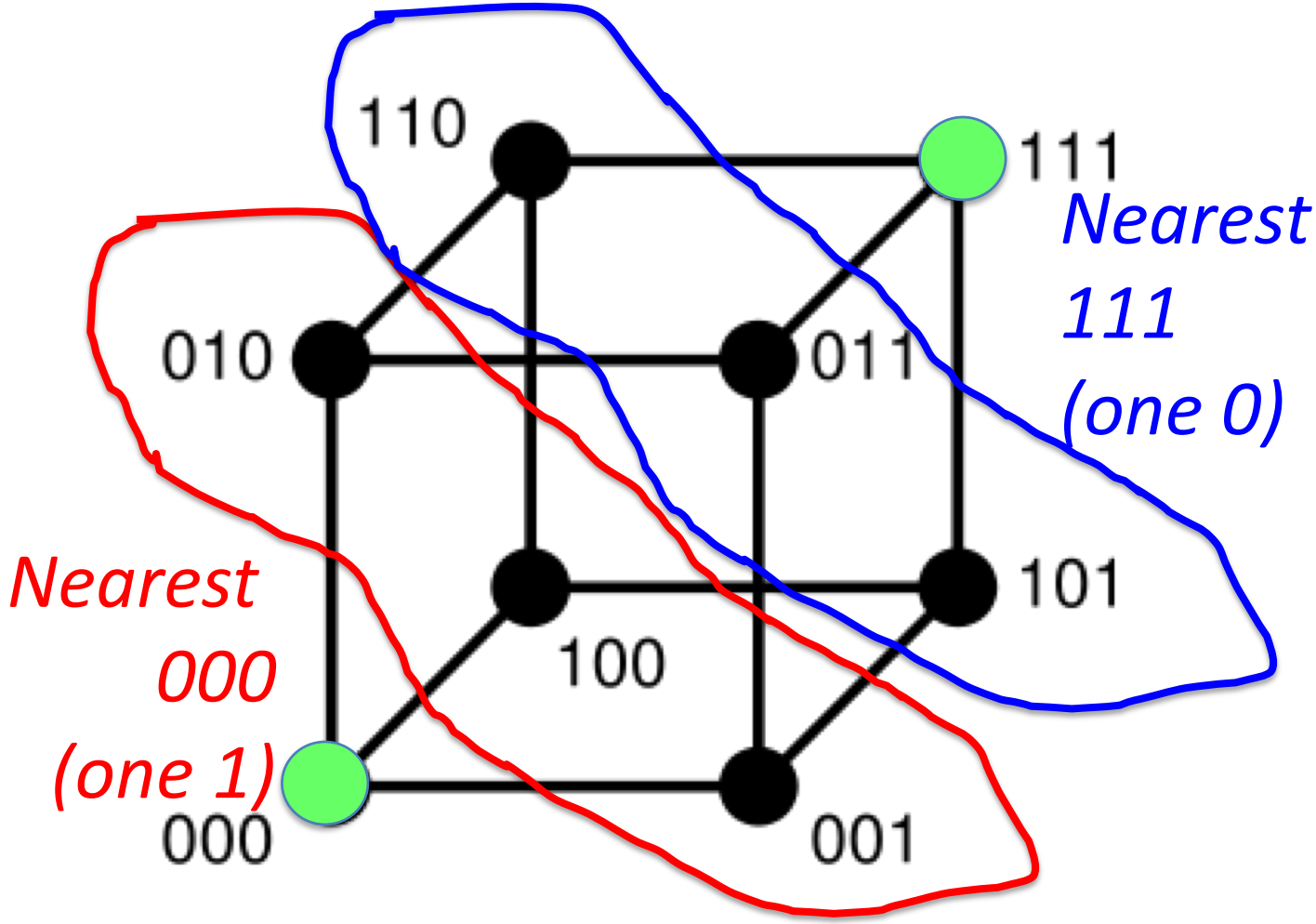
Detect Single Bit Errors



- No 1 bit error goes to another valid codeword
- ½ codewords are valid

Hamming Distance 3: Correction

Correct Single Bit Errors, Detect Double Bit Errors



- No 2 bit error goes to another valid codeword; 1 bit error near
- 1/4 codewords are valid

Administrivia

- Final Exam
  - Tuesday, June 26, 2017, 9:00-11:00
  - Location: Teaching Center 301 + 302
  - THREE cheat sheets (MT1, MT2, post-MT2)
    - Hand-written by you, English, A4
- Project 4 published
- HW 7 published

Hamming Error Correction Code

- Use of extra parity bits to allow the position identification of a single error

1. Mark all bit positions that are powers of 2 as parity bits (positions 1, 2, 4, 8, 16, …)
   - Start numbering bits at 1 at left (not at 0 on right)
2. All other bit positions are data bits (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, …)
3. Each data bit is covered by 2 or more parity bits

Hamming ECC

4. The position of parity bit determines sequence of data bits that it checks

- Bit 1 (0001_2): checks bits (1,3,5,7,9,11,...)
  - Bits with least significant bit of address = 1
- Bit 2 (0010_2): checks bits (2,3,6,7,10,11,14,15,...)
  - Bits with 2nd least significant bit of address = 1
- Bit 4 (0100_2): checks bits (4-7, 12-15, 20-23, ...)
  - Bits with 3rd least significant bit of address = 1
- Bit 8 (1000_2): checks bits (8-15, 24-31, 40-47 ,...)
  - Bits with 4th least significant bit of address = 1

Graphic of Hamming Code

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

- http://en.wikipedia.org/wiki/Hamming_code

Hamming ECC

5. Set parity bits to create even parity for each group

- A byte of data: 10011010
- Create the coded word, leaving spaces for the parity bits:
- _ _ 1 _ 0 0 1 _ 1 0 1 0

  0 0 0 0 0 0 0 0 0 0 1 1 1

  1 2 3 4 5 6 7 8 9 0 1 2
- Calculate the parity bits

Hamming ECC

- Position 1 checks bits 1,3,5,7,9,11 (bold):
  ? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0. set position 1 to a _:
  _ _ **1** _ **0** 0 **1** _ **1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11 (bold):
  0 **?** **1** _ 0 **0** **1** _ 1 **0** **1** 0. set position 2 to a _:
  0 _ **1** _ 0 **0** **1** _ 1 **0** **1** 0
- Position 4 checks bits 4,5,6,7,12 (bold):
  0 1 1 **?** **0** **0** **1** _ 1 0 1 **0**. set position 4 to a _:
  0 1 1 _ **0** **0** **1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
  0 1 1 1 0 0 1 **?** **1** **0** **1** **0**. set position 8 to a _:
  0 1 1 1 0 0 1 _ **1** **0** **1** **0**

Hamming ECC

- Position 1 checks bits 1,3,5,7,9,11:
  ? _ **1** _ **0** 0 **1** _ **1** 0 **1** 0. set position 1 to a 0:
  0 _ **1** _ **0** 0 **1** _ **1** 0 **1** 0
- Position 2 checks bits 2,3,6,7,10,11:
  0 **?** **1** _ 0 **0** **1** _ 1 **0** **1** 0. set position 2 to a 1:
  0 **1** **1** _ 0 **0** **1** _ 1 **0** **1** 0
- Position 4 checks bits 4,5,6,7,12:
  0 1 1 **?** **0** **0** **1** _ 1 0 1 **0**. set position 4 to a 1:
  0 1 1 **1** **0** **0** **1** _ 1 0 1 **0**
- Position 8 checks bits 8,9,10,11,12:
  0 1 1 1 0 0 1 **?** **1** **0** **1** **0**. set position 8 to a 0:
  0 1 1 1 0 0 1 **0** **1** **0** **1** **0**

Hamming ECC

- Final code word: 01110010101010
- Data word: 1 001 1010

Hamming ECC Error Check

- Suppose receive
  0 1 1 1 0 0 1 0 1 1 1 0

0 1 1 1 0 0 1 0 1 1 1 0

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

Hamming ECC Error Check

- Suppose receive
  0 1 1 1 0 0 1 0 1 1 1 0

Hamming ECC Error Check

- Suppose receive

  0 1 1 1 0 0 1 0 1 1 1 0

  0 1 0 1 1 1 √

  1 1 0 1 1 1 X-Parity 2 in error

  1 0 0 1 0 √

  0 1 1 1 0 X-Parity 8 in error

- *Implies position 8+2=10 is in error*

  0 1 1 1 0 0 1 0 **1** 1 0

Hamming ECC Error Correct

- Flip the incorrect bit …
  0 1 1 1 0 0 1 0 1 0 1 0

Hamming ECC Error Correct

- Suppose receive

  0 1 1 1 0 0 1 0 1 0 1 0

  0 1 0 1 1 1 √

  1 1 0 1 0 1 √

  1 0 0 1 0 √

  0 1 0 1 0 √

Hamming Error Correcting Code

- Overhead involved in single error-correction code
- Let *p* be total number of parity bits and *d* number of data bits in *p + d* bit word
- If p error correction bits are to point to error bit (*p + d* cases) + indicate that no error exists (1 case), we need:

  $2^p >= p + d + 1$,

  thus $p >= \log(p + d + 1)$

  for large *d*, *p* approaches log(*d*)
- *8 bits data => d = 8,* $2^p = p + 8 + 1 => p = 4$
- *16 data => 5 parity,*
  32 data => 6 parity,
  64 data => 7 parity

Hamming Single-Error Correction, Double-Error Detection (SEC/DED)

- Adding extra parity bit covering the entire word provides double error detection as well as single error correction

  1 2 3 4 5 6 7 8

  p_1 p_2 d_1 p_3 d_2 d_3 d_4 p_4

- Hamming parity bits H (p_1 p_2 p_3) are computed (even parity as usual) plus the even parity over the entire word, p_4:

  H=0 p_4=0, no error

  H≠0 p_4=1, correctable single error (odd parity if 1 error => p_4=1)

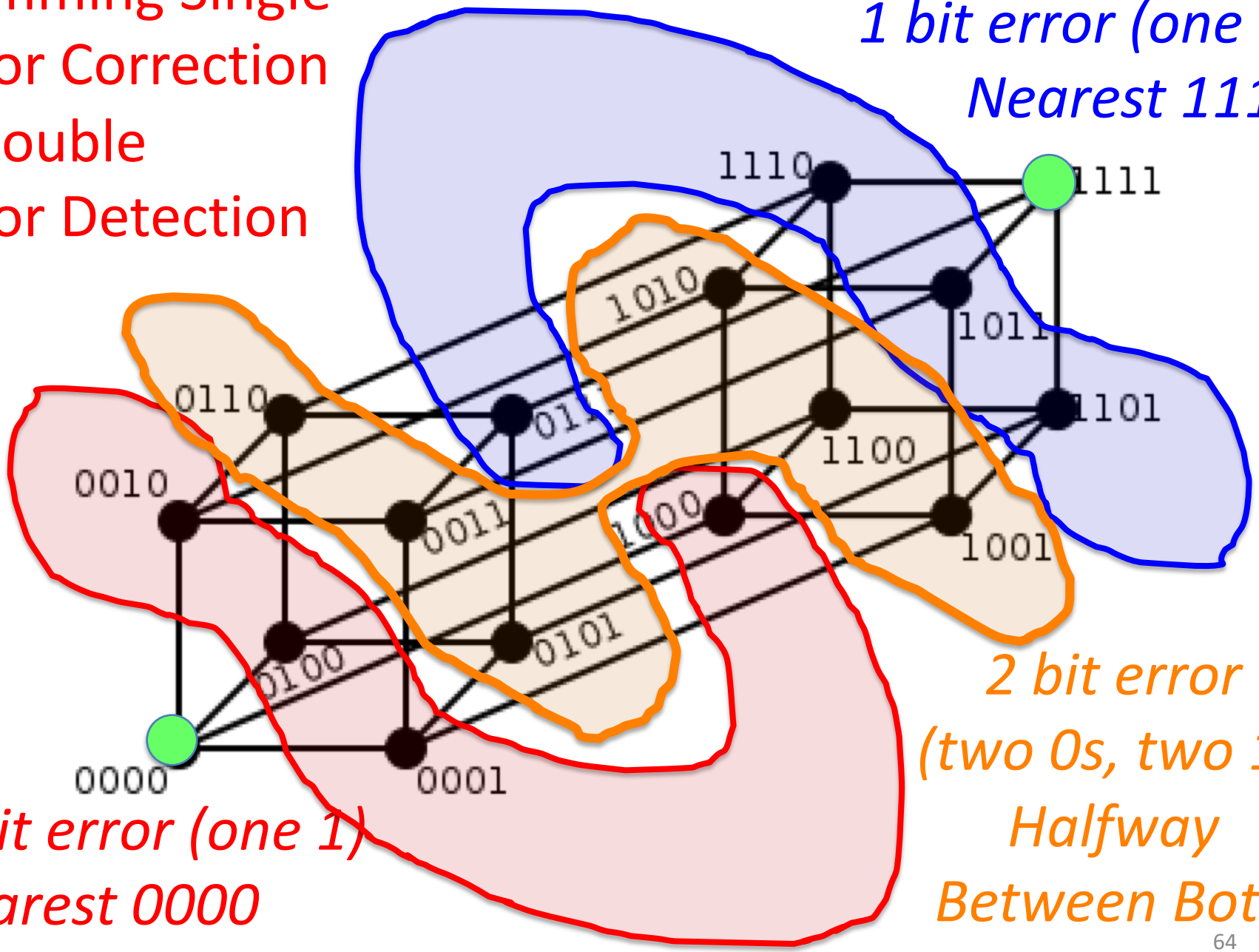
  H≠0 p_4=0, double error occurred (even parity if 2 errors=> p_4=0)

Typical modern codes in DRAM memory systems:

64-bit data blocks (8 bytes) with 72-bit code words (9 bytes).

Hamming Distance = 4

Hamming Single Error Correction + Double Error Detection

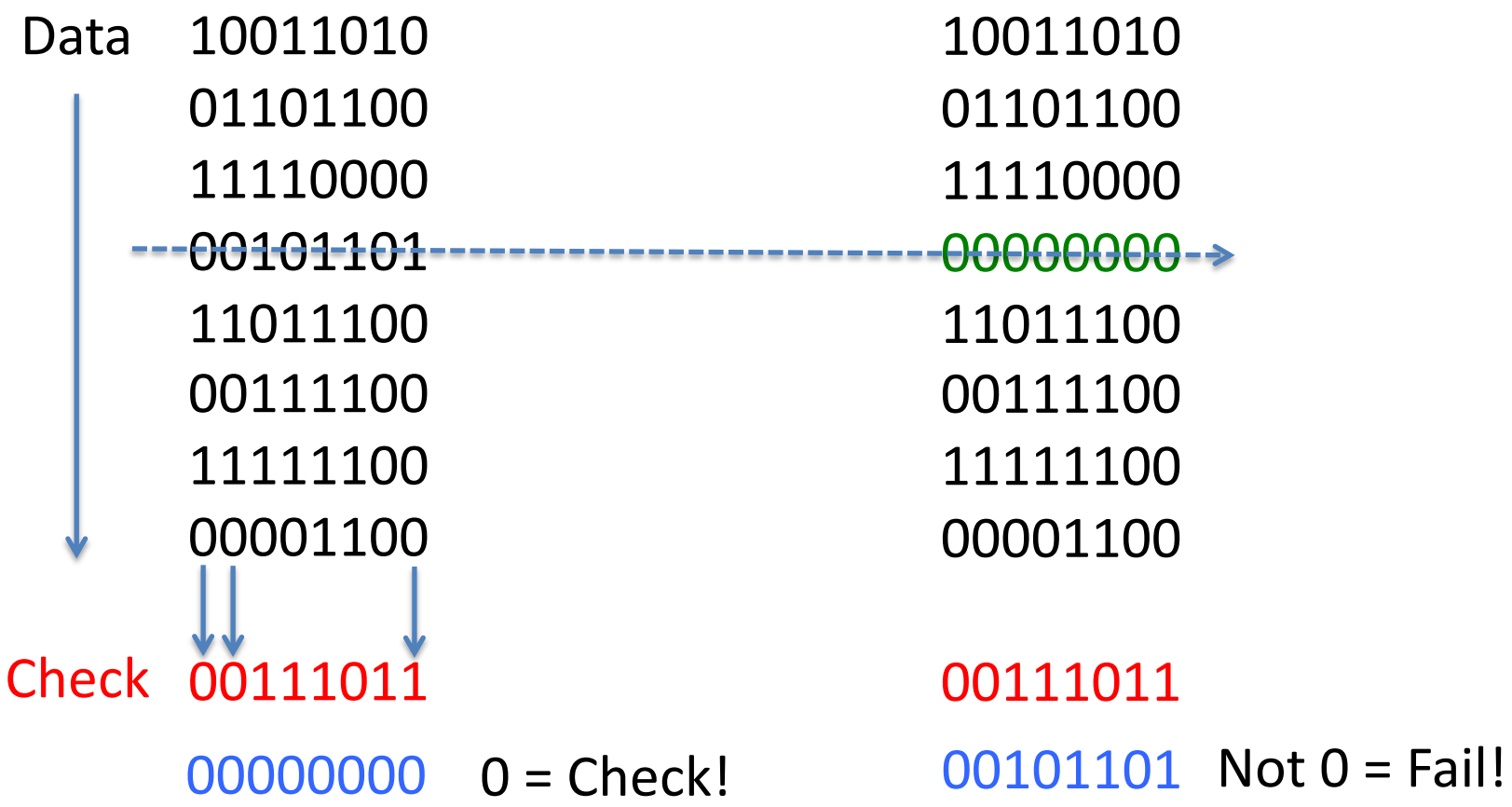


What if More Than 2-Bit Errors?

- Network transmissions, disks, distributed storage common failure mode is bursts of bit errors, not just one or two bit errors
  - Contiguous sequence of *B* bits in which first, last and any number of intermediate bits are in error
  - Caused by impulse noise or by fading in wireless
  - Effect is greater at higher data rates

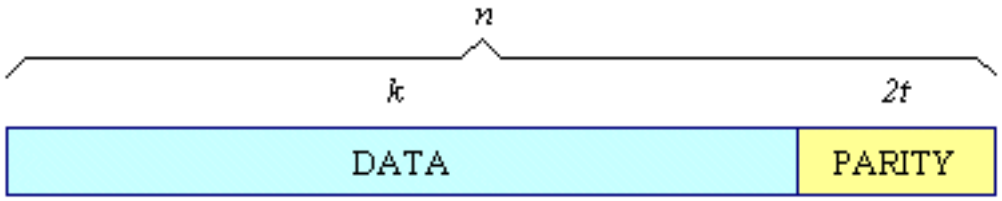
Cyclic Redundancy Check

Simple example: Parity Check Block



Cyclic Redundancy Check

- Parity codes not powerful enough to detect long runs of errors (also known as *burst errors*)
- Better Alternative: *Reed-Solomon Codes*
  - Used widely in CDs, DVDs, Magnetic Disks
  - RS(255,223) with 8-bit symbols: each codeword contains 255 code word bytes (223 bytes are data and 32 bytes are parity)



  - For this code: n = 255, k = 223, s = 8, 2t = 32, t = 16
  - Decoder can correct any errors in up to 16 bytes anywhere in the codeword

Cyclic Redundancy Check

14 data bits 3 check bits 17 bits total

```
11010011101100 000 <--- input right padded by 3 bits
1011               <--- divisor
01100011101100 000 <--- result
 1011              <--- divisor
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000 <--- skip leading zeros
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
-----------------
00000000000000 100 <--- remainder
```

3 bit CRC using the polynomial $x^3 + x + 1$
(divide by 1011 to get remainder)

Didn’t finish lecture!

- Read and understand:
  - the rest of the lecture material!
  - P&H 5.5 “Dependable Memory Hierarchy”
- Will be part of the final!