UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 24: Virtual Memory and Dependable Memory

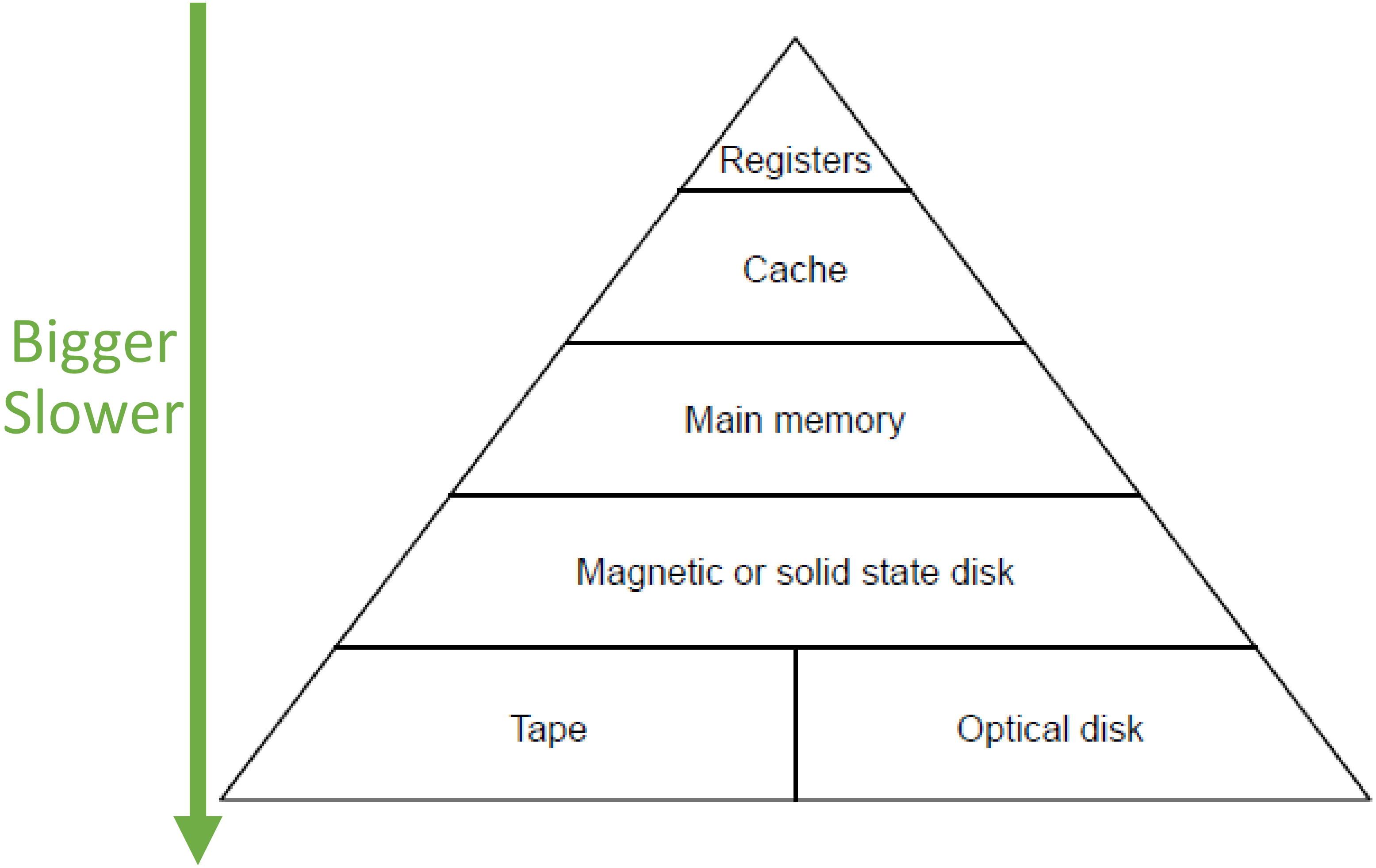
Taylor Johnson

Announcements and Outline

- Programming assignment 2 assigned, due 11/13 (tonight) by midnight

- Finish Cache Control, Coherence / Consistency
- Dependable and Virtual Memory

Memory Hierarchy



Bigger
Slower

Registers

Cache

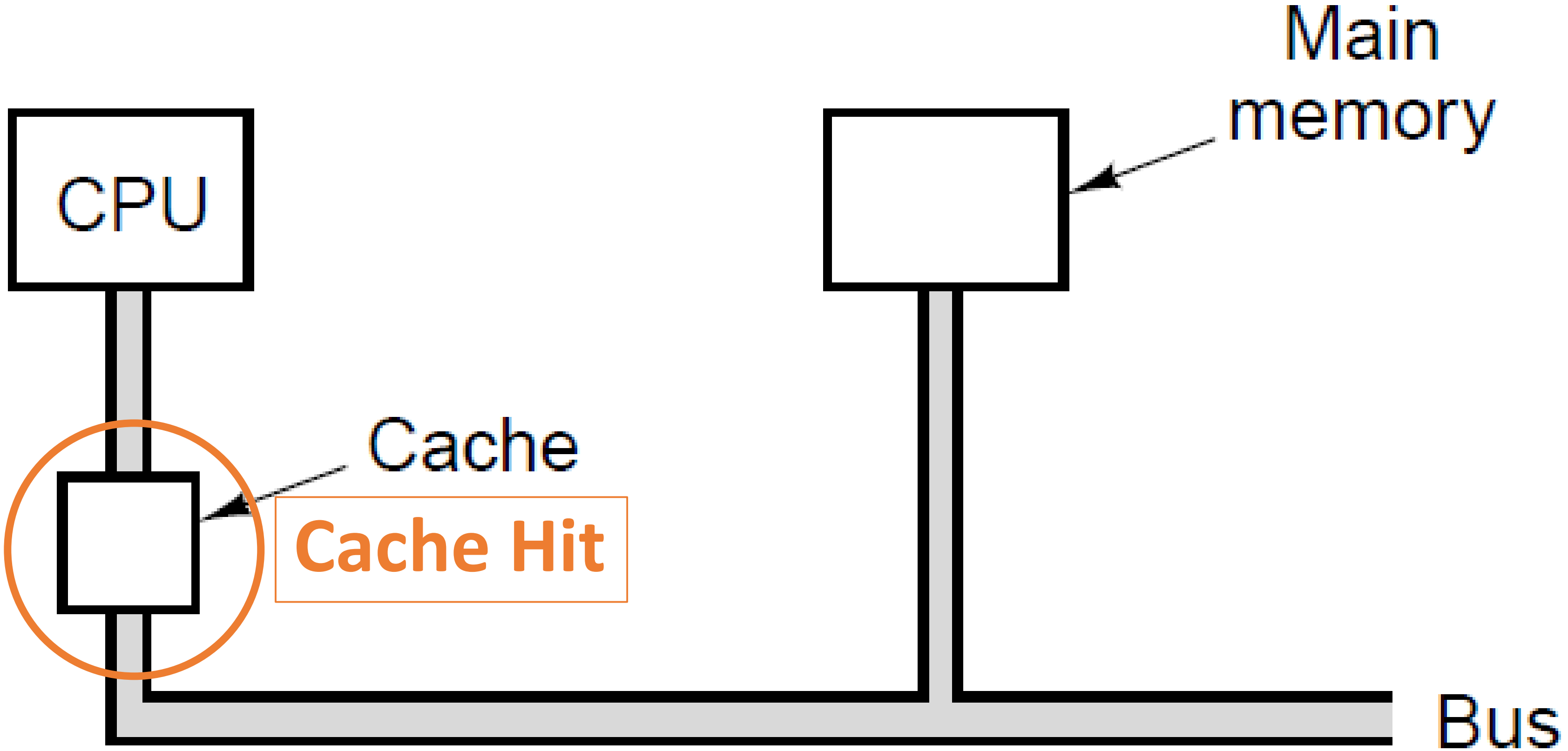
Main memory

Magnetic or solid state disk

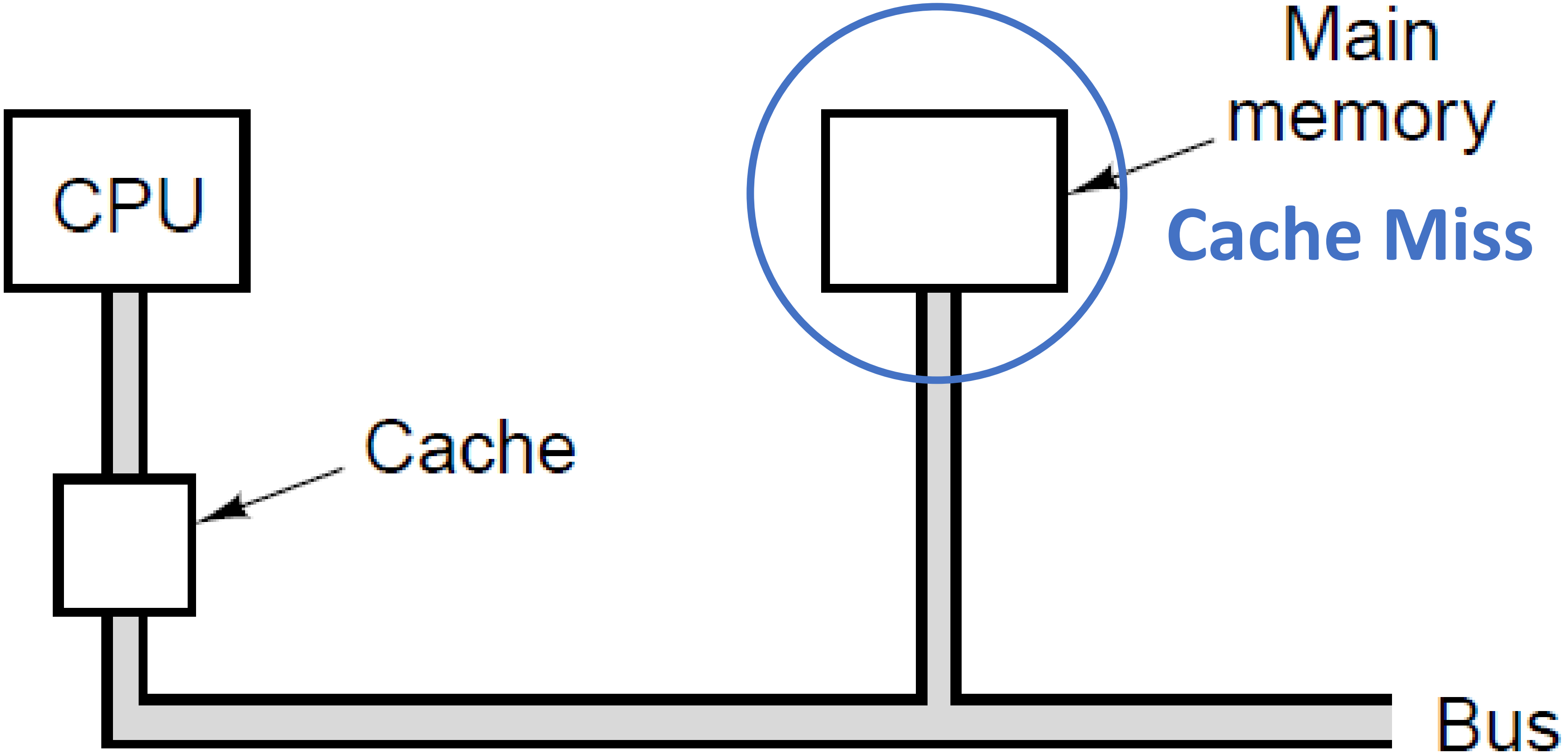
Tape

Optical disk

Cache Hit: find necessary data in cache



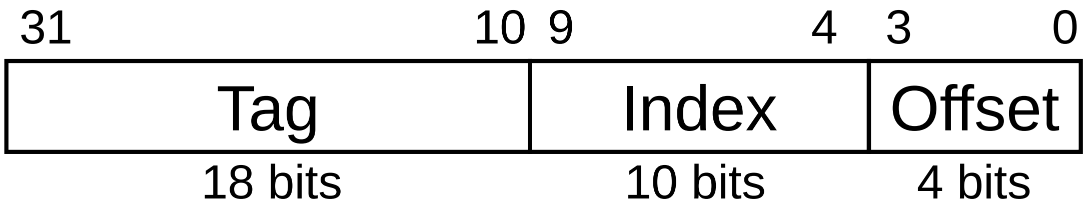
Cache Miss: have to get necessary data from main memory



Cache Control, Coherence / Consistency

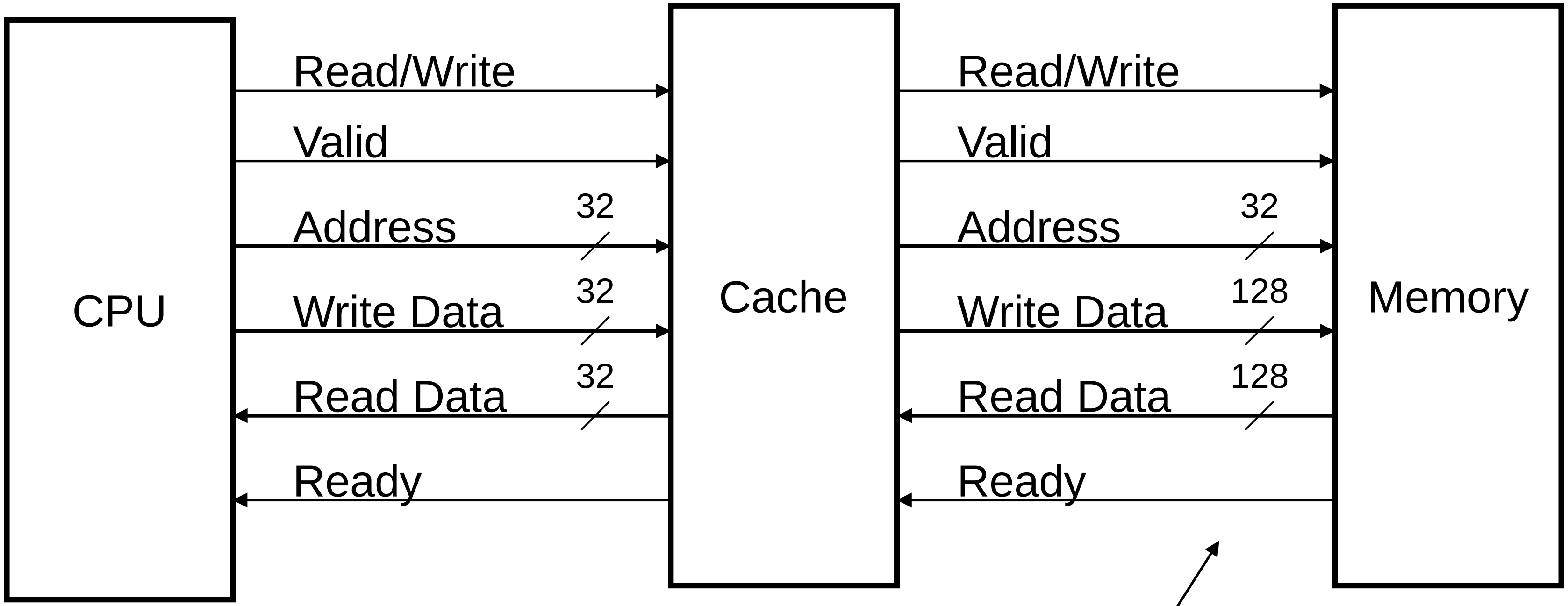
Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete



31 … 10	9 … 4	3 … 0
Tag	Index	Offset
18 bits	10 bits	4 bits

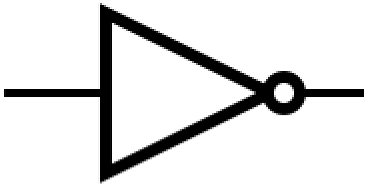
Interface Signals

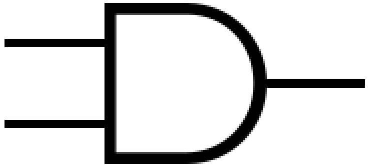


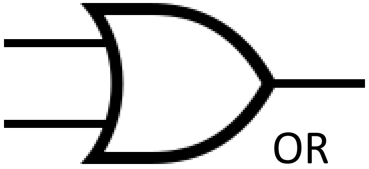
Multiple cycles per access

Digital Logic

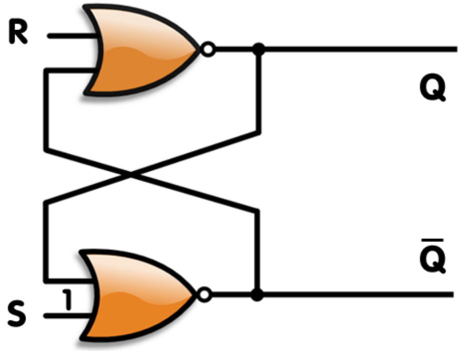
- Combinational Circuits
  - Stateless (“memoryless”)
  - “Combine” inputs only
  - Examples: AND gates, OR gates, NOT gates, adders (ALUs), multiplexors, …





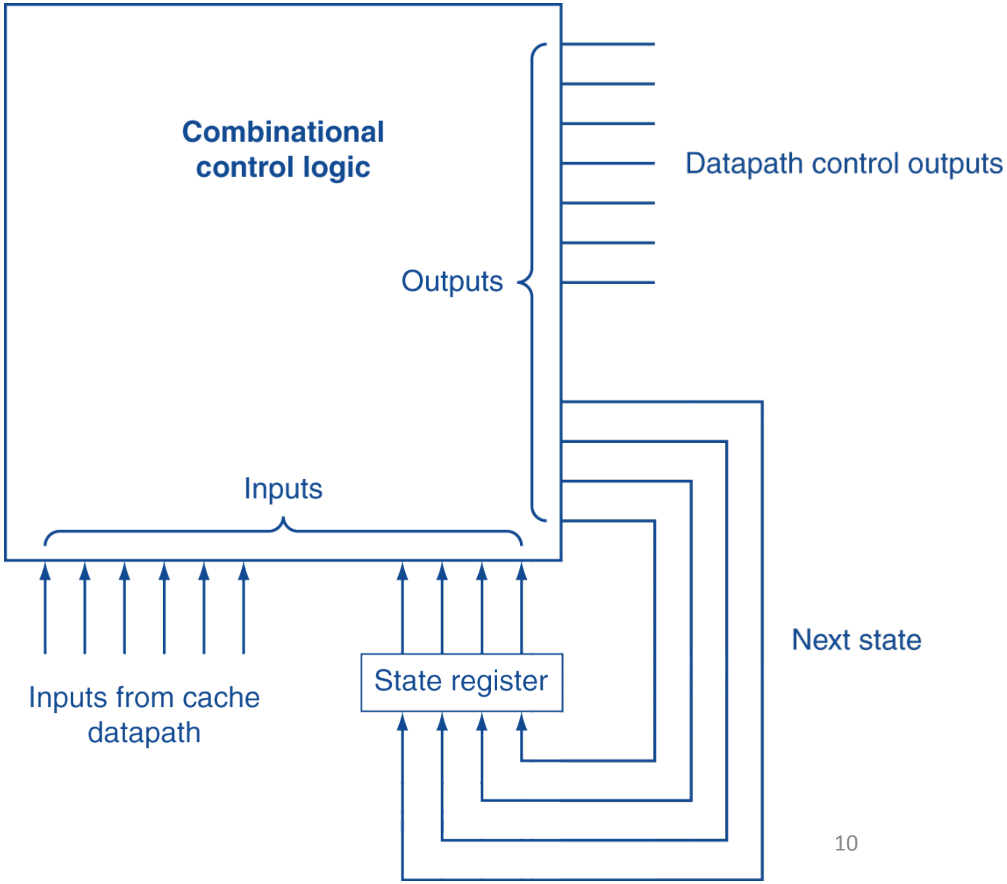


- Sequential Circuits
  - Stateful (“has memory”)
  - Combines inputs and state (memory)
  - Examples: memory, latches / flip-flops, shift registers

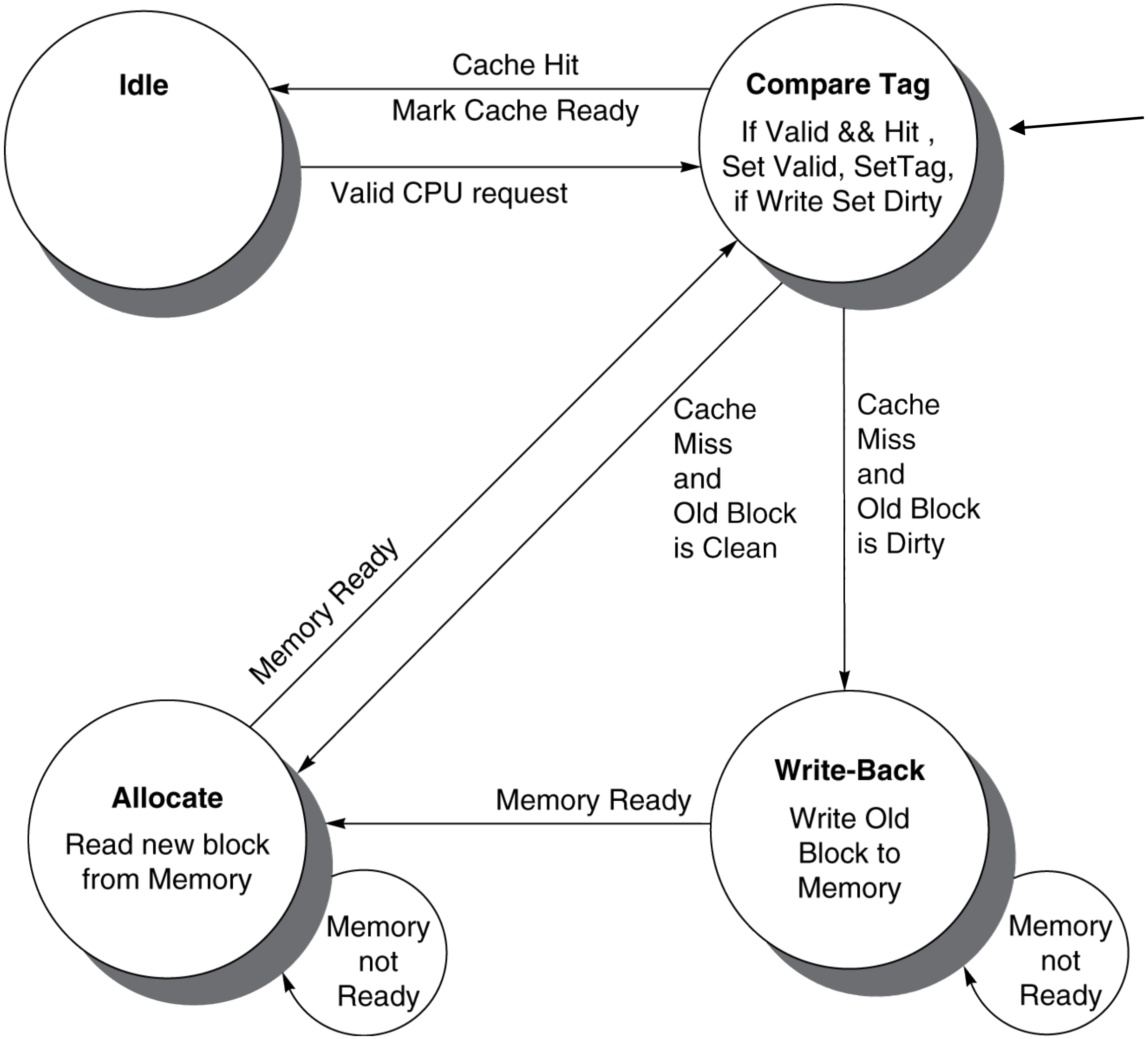


Finite State Machines

- Use a FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state = f_n (current state, current inputs)
- Control output signals = f_o (current state)



Cache Controller FSM



Could partition into separate states to reduce clock cycle time

Cache Coherence Problem

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1
4	CPU B reads X (cache hit)	1	**0**	1

Coherence Defined

- Informally: Reads return most recently written value
- More Formally:
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1		0
CPU B read X	Cache miss for X	1	1	1

Memory Consistency

- When are writes seen by other processors
  - “Seen” means a read returns the written value
  - Can’t be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    ⇒ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

ARM Cortex A-8 L1 Cache

- The L1 memory system consists of ***separate instruction and data caches*** in a ***Harvard*** arrangement. The L1 memory system provides the core with:
  - fixed line length of 64 bytes
  - support for 16KB or 32KB caches
  - two 32-entry fully associative ARMv7-A MMU
  - data array with parity for error detection
  - an instruction cache that is virtually indexed, IVIPT
  - a data cache that is physically indexed, PIPT
  - ***4-way set associative cache structure***
  - ***random replacement policy***
  - nonblocking cache behavior for Advanced SIMD code
  - blocking for integer code
  - MBIST
  - support for hardware reset of the L1 data cache valid RAM (clear valid bits)

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html

ARM Cortex A-8 L2 Cache

- The L2 memory system is tightly coupled to the L1 data cache and L1 instruction cache. The L2 memory system ***does not support hardware cache coherency***, therefore software intervention is required to maintain coherency in the system.
  - The key features of the L2 memory system include:
  - configurable cache size of 0KB, 128KB, 256KB, 512KB, and 1MB
  - fixed line length of 64 bytes
  - physically indexed and tagged
  - ***8-way set associative cache structure***
  - support for lockdown format C
  - configurable 64-bit or 128-bit wide AXI system bus interface with support for multiple outstanding requests
  - ***random replacement policy***
  - optional ECC or parity protection on the data RAM
  - optional parity protection on the tag RAM
  - MBIST
  - support hardware reset of the L2 unified cache valid RAM (clear valid bits)

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html

ARM Cache Control Register Functions (will not use in PAs)

Function	Description
Invalidate cache	Invalidates all cache data, including any dirty data.
Invalidate single entry using either index or modified virtual address	Invalidates a single cache line, discarding any dirty data.
Clean single data entry using either index or modified virtual address	Writes the specified DCache line to main memory if the line is marked valid and dirty. The line is marked as not dirty. The valid bit is unchanged.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/I1014942.html

ARM Cache Control Instructions (will not use in PAs)

Function/operation	Data format	Instruction
Invalidate ICache and DCache	SBZ	MCR p15, 0, <Rd>, c7, c7, 0
Invalidate ICache	SBZ	MCR p15, 0, <Rd>, c7, c5, 0
Invalidate ICache single entry (MVA)	MVA	MCR p15, 0, <Rd>, c7, c5, 1
Invalidate ICache single entry (Set/Way)	Set/Way	MCR p15, 0, <Rd>, c7, c5, 2
Prefetch ICache line (MVA)	MVA	MCR p15, 0, <Rd>, c7, c13, 1
Invalidate DCache	SBZ	MCR p15, 0, <Rd>, c7, c6, 0
Invalidate DCache single entry (MVA)	MVA	MCR p15, 0, <Rd>, c7, c6, 1
Invalidate DCache single entry (Set/Way)	Set/Way	MCR p15, 0, <Rd>, c7, c6, 2
Clean DCache single entry (MVA)	MVA	MCR p15, 0, <Rd>, c7, c10, 1
Clean DCache single entry (Set/Way)	Set/Way	MCR p15, 0, <Rd>, c7, c10, 2
Test and clean DCache	-	MRC p15, 0, <Rd>, c7, c10, 3
Clean and invalidate DCache entry (MVA)	MVA	MCR p15, 0, <Rd>, c7, c14, 1
Clean and invalidate DCache entry (Set/Way)	Set/Way	MCR p15, 0, <Rd>, c7, c14, 2
Test, clean, and invalidate DCache	-	MRC p15, 0, <Rd>, c7, c14, 3
Drain write buffer	SBZ	MCR p15, 0, <Rd>, c7, c10, 4

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/I1014942.html

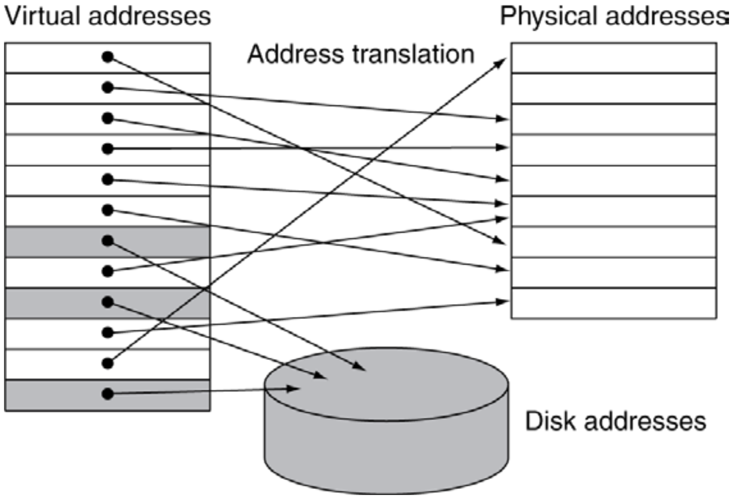
Virtual Memory

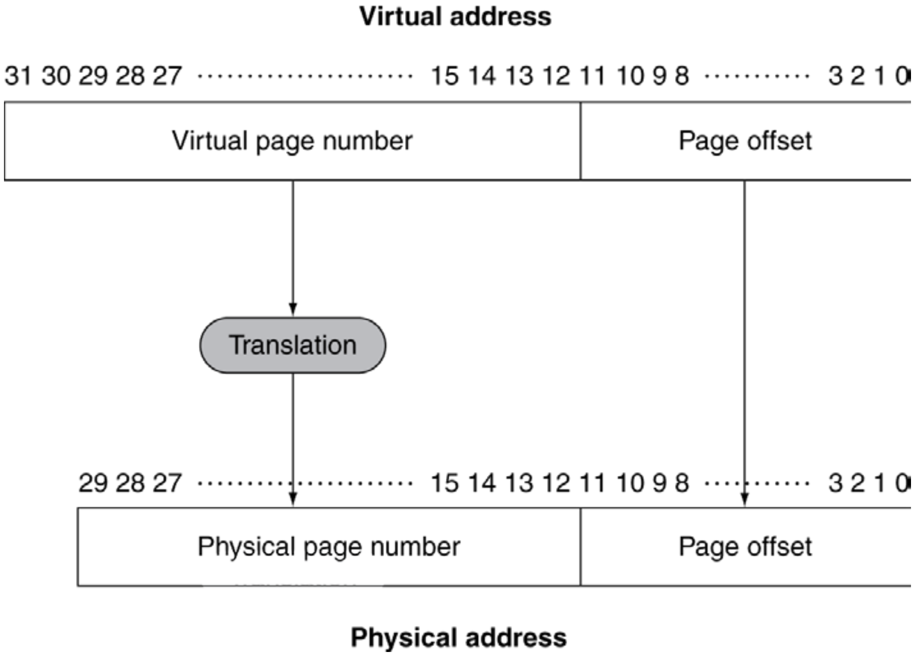
Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
  - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
  - Each gets a private virtual address space holding its frequently used code and data
  - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
  - VM “block” is called a page
  - VM translation “miss” is called a page fault
  - ***Memory management unit (MMU)***

Address Translation

- Fixed-size pages (e.g., 4K)





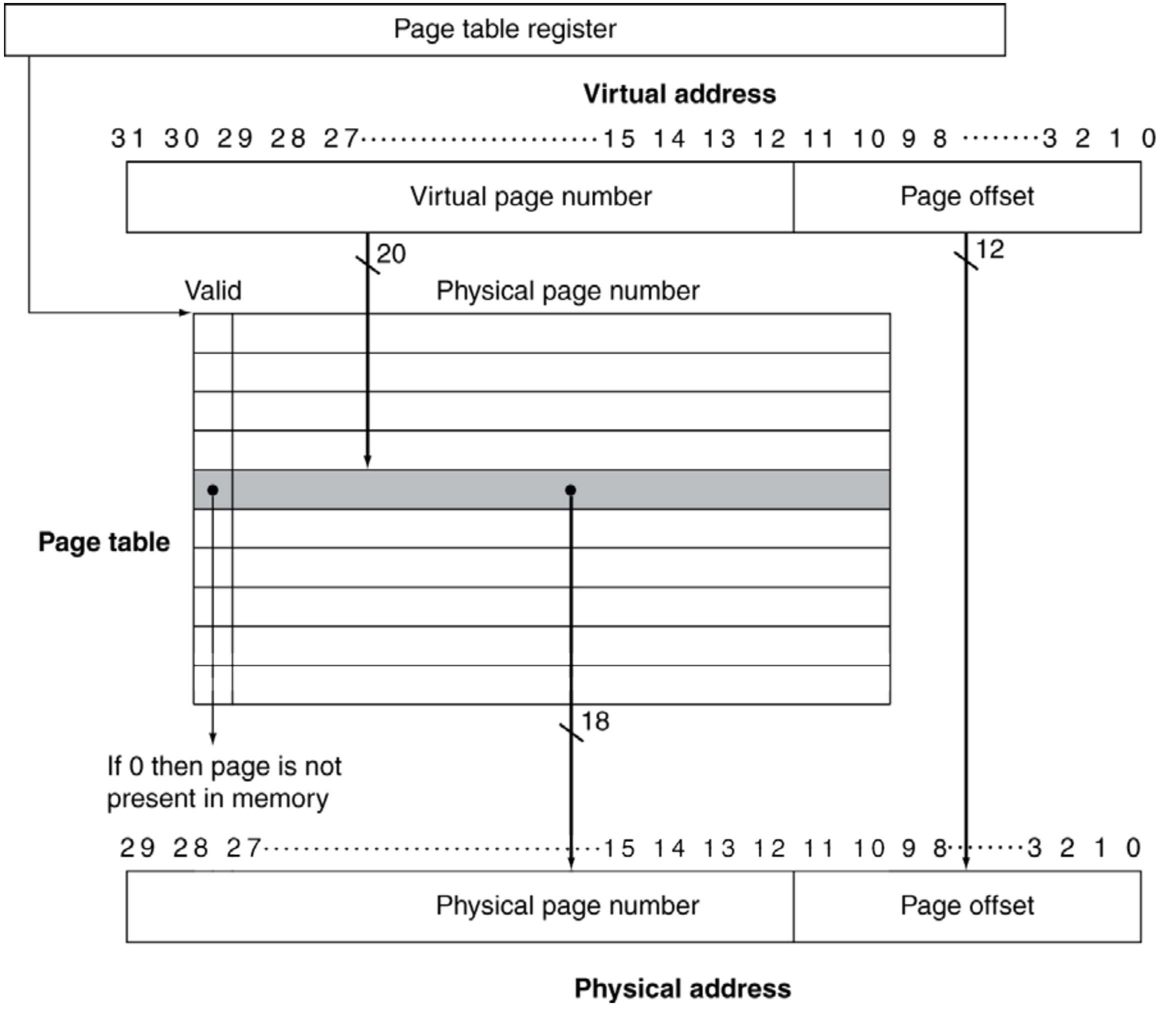
Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
  - Handled by OS code
- Try to minimize page fault rate
  - Fully associative placement
  - Smart replacement algorithms

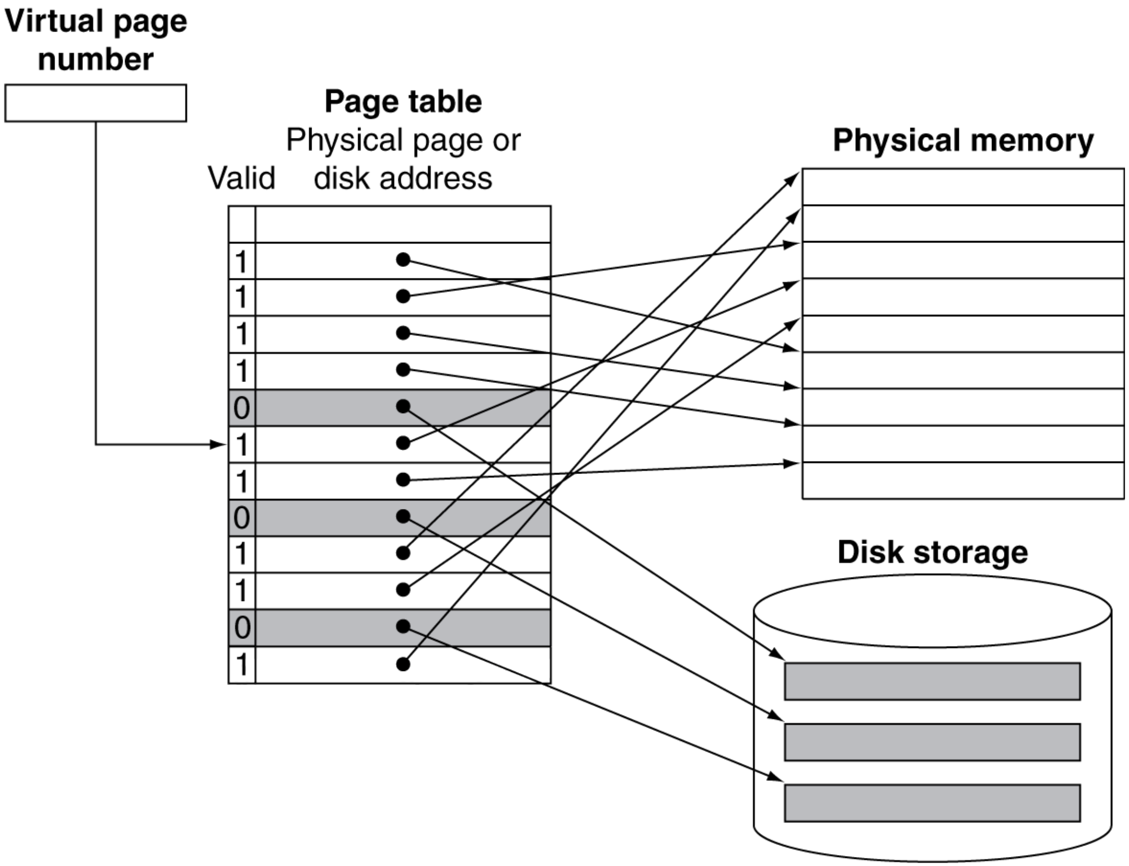
Page Tables

- PTE: Page Table Entry
- Stores placement information
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory
- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, …)
- If page is not present
  - PTE can refer to location in swap space on disk

Translation Using a Page Table



Mapping Pages to Storage



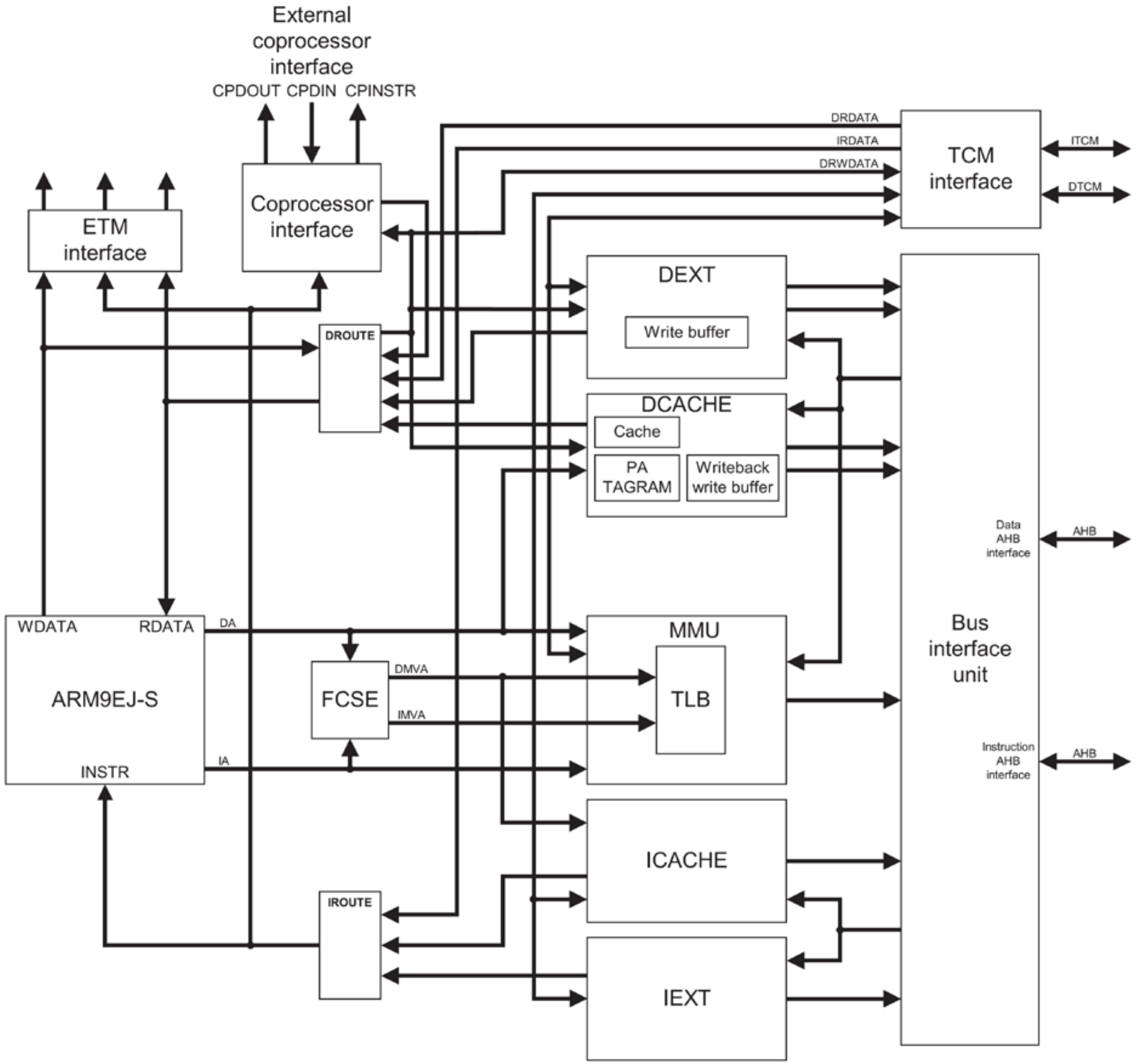
Replacement and Writes

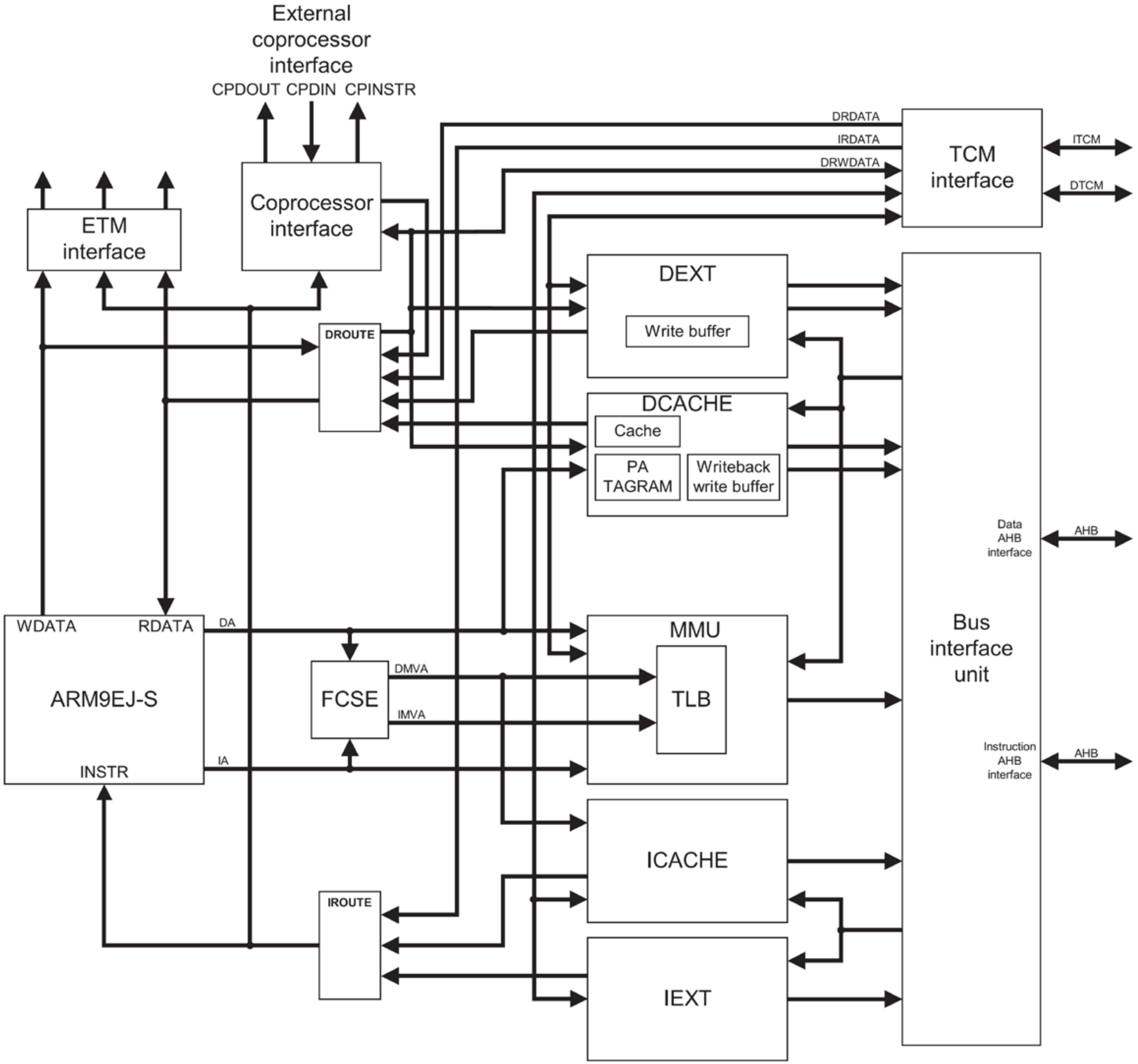
- To reduce page fault rate, prefer least-recently used (LRU) replacement
  - Recall LRU policy in associative cache replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page
  - Periodically cleared to 0 by OS
  - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
  - Block at once, not individual locations
  - Write through is impractical
  - Use write-back
  - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
- But access to page tables has good locality
  - So use a fast cache of PTEs ***within the CPU***
  - Called a ***Translation Look-aside Buffer (TLB)***
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

ARM Interface Organization



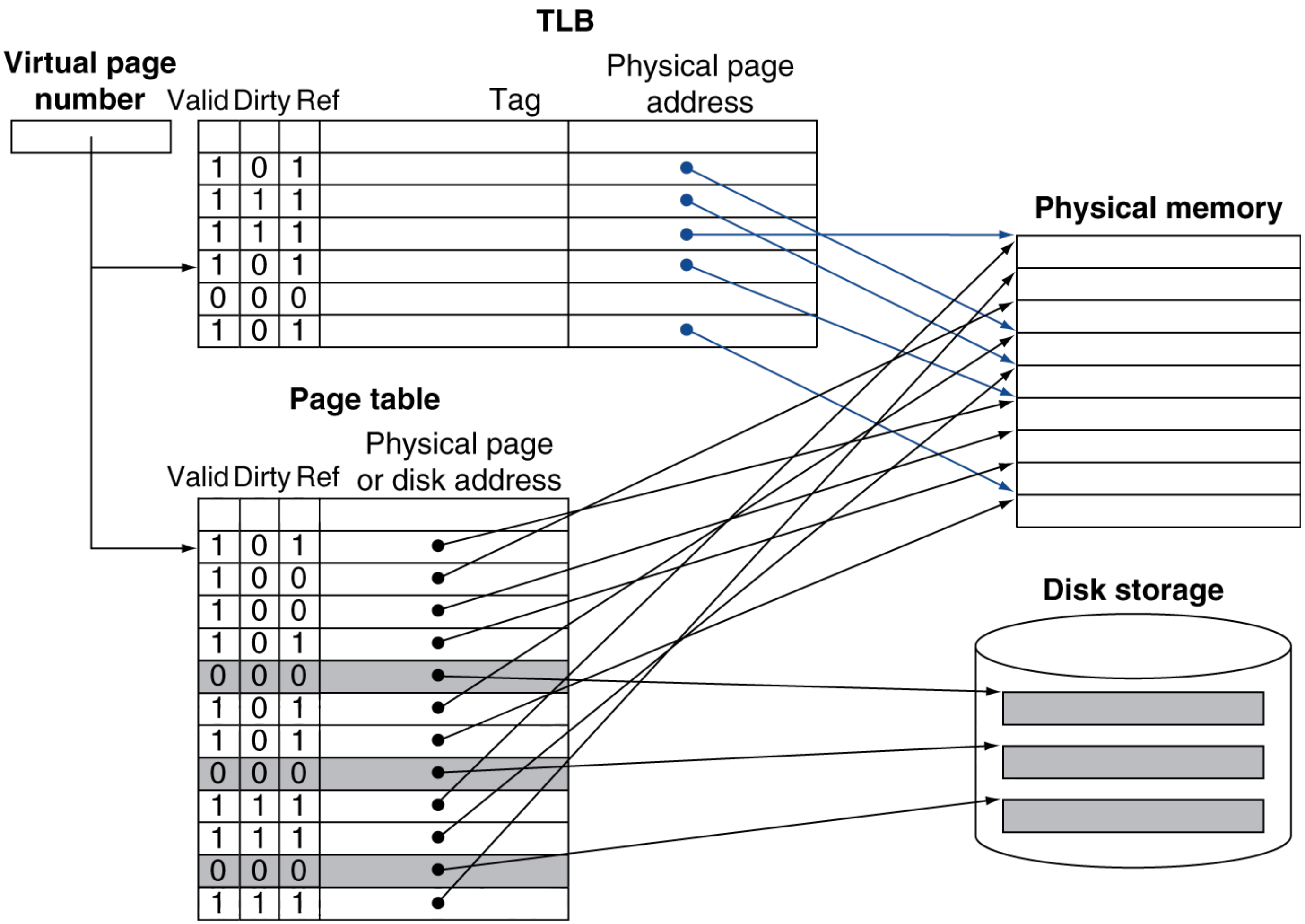

External coprocessor interface
CPDOUT
CPDIN
CPINSTR
Coprocessor interface
ETM interface
DRDATA
IRDATA
DRWDATA
TCM interface
ITCM
DTCM
DEXT
Write buffer
DROUTE
DCACHE
Cache
PA TAGRAM
Writeback write buffer
Data AHB interface
AHB
Bus interface unit
WDATA
RDATA
DA
ARM9EJ-S
INSTR
IA
FCSE
DMVA
IMVA
MMU
TLB
Instruction AHB interface
AHB
ICACHE
IROUTE
IEXT

ARM Cortex A-8 MMU

- MMU works with L1 and L2 memory system to translate virtual addresses to physical addresses
- It also controls accesses to and from external memory
- The processor implements the ARMv7-A MMU enhanced with Security Extensions features to provide address translation and access permission checks.
- The MMU controls table walk hardware that accesses translation tables in main memory. The MMU enables fine-grained memory system control through a set of virtual-to-physical address mappings and memory attributes held in instruction and data TLBs.
- The MMU features include the following:
  - full support for *Virtual Memory System Architecture version 7* (VMSAv7)
  - ***separate, fully-associative, 32-entry data and instruction TLBs***
  - support for 32 lockable entries using the lock-by-entry model
  - TLB entries that support 4KB, 64KB, 1MB, and 16MB pages
  - 16 domains
  - global and application-specific identifiers to prevent context switch TLB flushes
  - extended permissions check capability
  - round-robin replacement policy
  - CP15 TLB preloading instructions to enable locking of TLB entries.

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html

Fast Translation Using a TLB



TLB

Valid	Dirty	Ref	Tag	Physical page address
1	0	1		●
1	1	1		●
1	1	1		●
1	0	1		●
0	0	0		
1	0	1		●

Page table

Valid	Dirty	Ref	Physical page or disk address
1	0	1	●
1	0	0	●
1	0	0	●
1	0	1	●
0	0	0	●
1	0	1	●
1	0	1	●
0	0	0	●
1	1	1	●
1	1	1	●
0	0	0	●
1	1	1	●

TLB Misses

- If page is in memory
  - Load the PTE from memory and retry
  - Could be handled in hardware
    - Can get complex for more complicated page table structures
  - Or in software
    - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
  - OS handles fetching the page and updating the page table
  - Then restart the faulting instruction

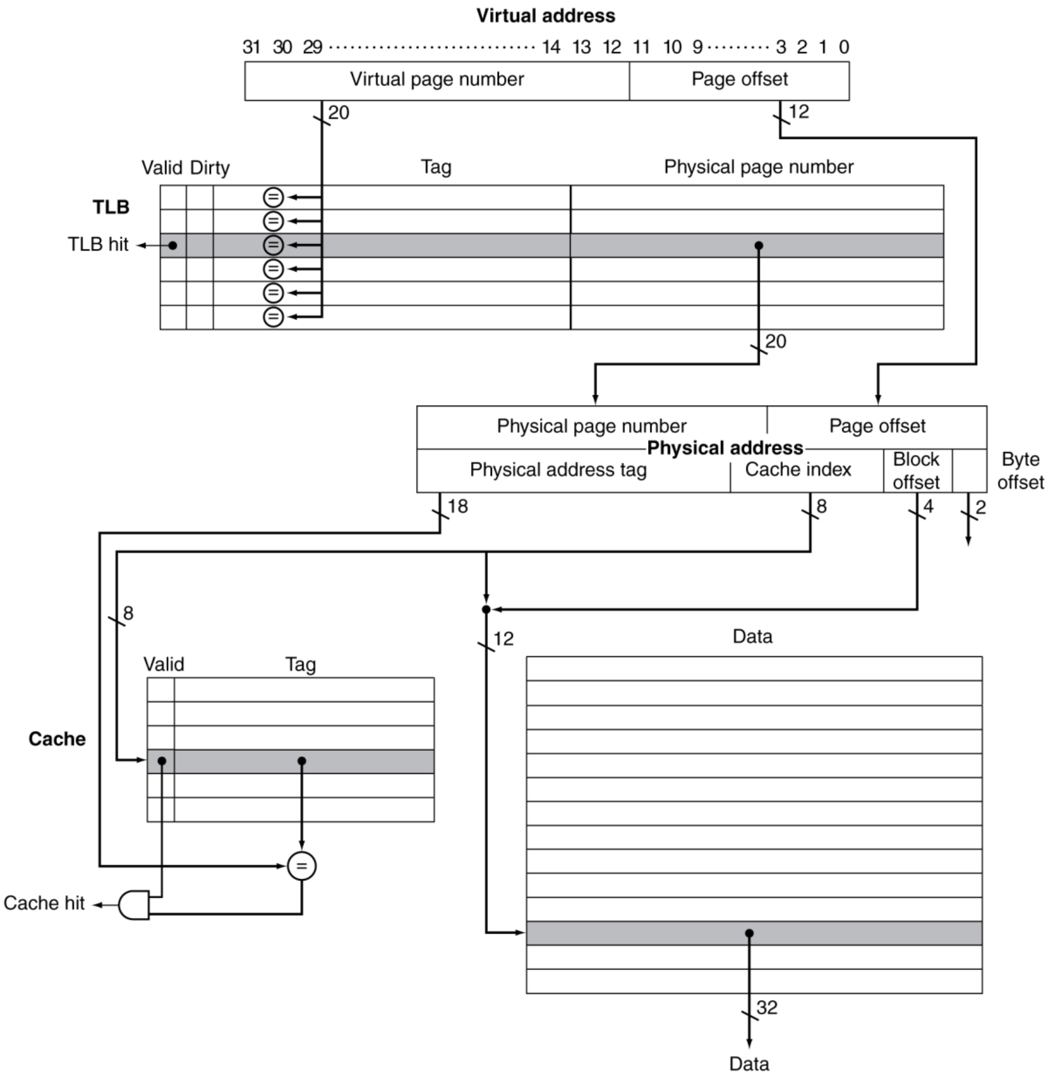
TLB Miss Handler

- TLB miss indicates
  - Page present, but PTE not in TLB
  - Page not preset
- Must recognize TLB miss before destination register overwritten
  - Raise exception
- Handler copies PTE from memory to TLB
  - Then restarts instruction
  - If page not present, page fault will occur

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction

TLB and Cache Interaction



- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address

Example ARM TLB Instructions (will not use in PAs)

Table 2.19. Register c8 TLB operations

ARMv4/ARMv5 operation	**ARM926EJ-S operation**	**Data**	**Instruction**
Invalidate TLB	Invalidate set-associative TLB	SBZ	MCR p15, 0, <Rd>, c8, c7, 0
Invalidate TLB single entry (MVA)	Invalidate single entry	MVA	MCR p15, 0, <Rd>, c8, c7, 1
Invalidate instruction TLB	Invalidate set-associative TLB	SBZ	MCR p15, 0, <Rd>, c8, c5, 0
Invalidate instruction TLB single entry (MVA)	Invalidate single entry	MVA	MCR p15, 0, <Rd>, c8, c5, 1
Invalidate data TLB	Invalidate set-associative TLB	SBZ	MCR p15, 0, <Rd>, c8, c6, 0
Invalidate data TLB single entry (MVA)	Invalidate single entry	MVA	MCR p15, 0, <Rd>, c8, c6, 1

[http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/Babfdfbh.html]

Memory Protection

- Different tasks can share parts of their virtual address spaces
  - But need to protect against errant access
  - Requires OS assistance
- Hardware support for OS protection
  - Privileged supervisor mode (aka kernel mode)
  - Privileged instructions
  - Page tables and other state information only accessible in supervisor mode
  - System call exception (e.g., syscall in MIPS)

Commonalities Between Memory Hierarchies

Cache = faster way to access larger main memory

Virtual memory = cache for storage (e.g., faster way to access larger secondary memory / storage)

Memory Hierarchy Big Picture

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy

Block Placement

- Determined by associativity
  - Direct mapped (1-way associative)
    - One choice for placement
  - n-way set associative
    - n choices within a set
  - Fully associative
    - Any location
- Higher associativity reduces miss rate
  - Increases complexity, cost, and access time

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
  - Reduce comparisons to reduce cost
- Virtual memory
  - Full table lookup makes full associativity feasible
  - Benefit in reduced miss rate

Replacement

- Choice of entry to replace on a miss
  - Least recently used (LRU)
    - Complex and costly hardware for high associativity
  - Random
    - Close to LRU, easier to implement
- Virtual memory
  - LRU approximation with hardware support

Write Policy

- Write-through
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer
- Write-back
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state
- Virtual memory
  - Only write-back is feasible, given disk write latency

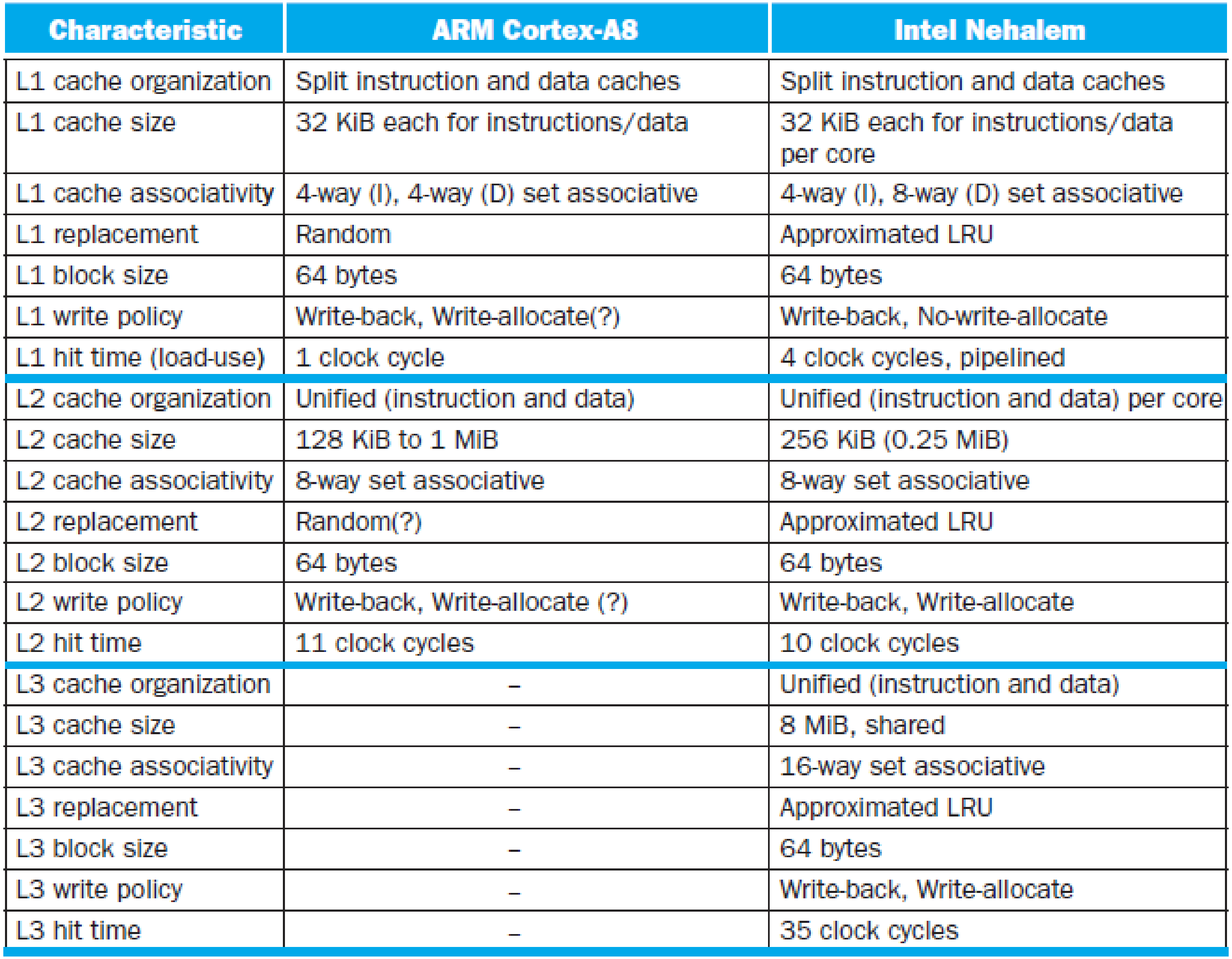
Sources of Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size

Cache Design Trade-offs

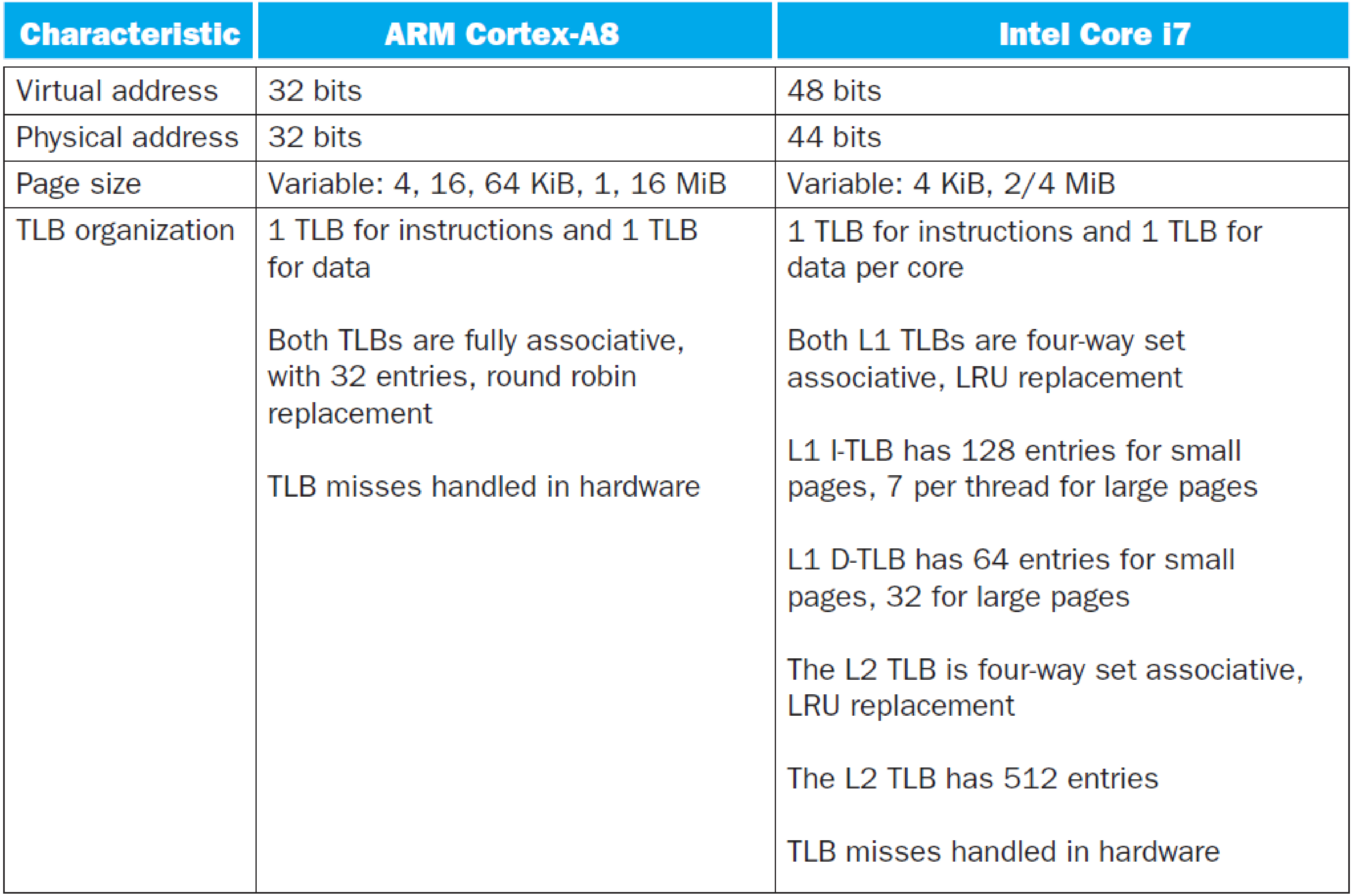
Design change	Effect on miss rate	Negative performance effect
Increase cache size	Decrease capacity misses	May increase access time
Increase associativity	Decrease conflict misses	May increase access time
Increase block size	Decrease compulsory misses	Increases miss penalty. For very large block size, may increase miss rate due to pollution.

Multilevel On-Chip Caches



Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

2-Level TLB Organization

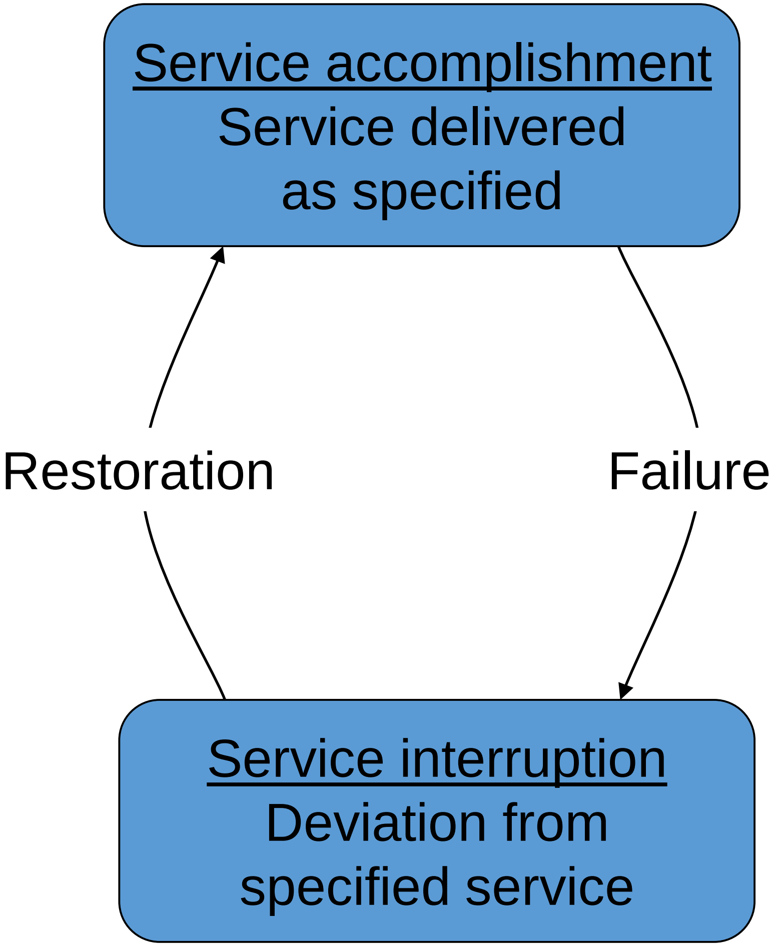


Characteristic	ARM Cortex-A8	Intel Core i7
Virtual address	32 bits	48 bits
Physical address	32 bits	44 bits
Page size	Variable: 4, 16, 64 KiB, 1, 16 MiB	Variable: 4 KiB, 2/4 MiB
TLB organization	1 TLB for instructions and 1 TLB for data Both TLBs are fully associative, with 32 entries, round robin replacement TLB misses handled in hardware	1 TLB for instructions and 1 TLB for data per core Both L1 TLBs are four-way set associative, LRU replacement L1 I-TLB has 128 entries for small pages, 7 per thread for large pages L1 D-TLB has 64 entries for small pages, 32 for large pages The L2 TLB is four-way set associative, LRU replacement The L2 TLB has 512 entries TLB misses handled in hardware

Dependable Memory

Dependability Measures, Error Correcting Codes, RAID, …

Dependability



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

Dependability Measures

- Reliability: mean time to failure (MTTF)
- Service interruption: mean time to repair (MTTR)
- Mean time between failures
  - MTBF = MTTF + MTTR
- Availability = MTTF / (MTTF + MTTR)
- Improving Availability
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

The Hamming SEC Code

- Hamming distance
  - Number of bits that are different between two bit patterns
- Minimum distance = 2 provides single bit error detection
  - E.g. parity code
- Minimum distance = 3 provides single error correction, 2 bit error detection

Encoding SEC

- To calculate Hamming code:
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:

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

Decoding SEC

- Value of parity bits indicates which bits are in error
  - Use numbering from encoding procedure
  - E.g.
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

SEC/DEC Code

- Add an additional parity bit for the whole word (p_n)
- Make Hamming distance = 4
- Decoding:
    - Let H = SEC parity bits
        - H even, p_n even, no error
        - H odd, p_n odd, correctable single bit error
        - H even, p_n odd, error in p_n bit
        - H odd, p_n even, double error occurred
- Note: ECC DRAM uses SEC/DEC with 8 bits protecting each 64 bits

Error Detection – Error Correction

- Memory data can get corrupted, due to things like:
  - Voltage spikes.
  - Cosmic rays.
- The goal in **error detection** is to come up with ways to tell if some data has been corrupted or not.
- The goal in **error correction** is to not only detect errors, but also be able to correct them.
- Both error detection and error correction work by attaching additional bits to each memory word.
- Fewer extra bits are needed for error detection, more for error correction.

Encoding, Decoding, Codewords

- Error detection and error correction work as follows:
- Encoding stage:
  - Break up original data into m-bit words.
  - Each m-bit original word is converted to an n-bit **codeword.**
- Decoding stage:
  - Break up encoded data into n-bit codewords.
  - By examining each n-bit codeword:
    - Deduce if an error has occurred.
    - Correct the error if possible.
    - Produce the original m-bit word.

Parity Bit

- Suppose that we have an *m*-bit word.
- Suppose we want a way to tell if a single error has occurred (i.e., a single bit has been corrupted).
  - No error detection/correction can catch an unlimited number of errors.
- Solution: represent each *m*-bit word using an (*m+1)*-bit codeword.
  - The extra bit is called **parity bit**.
- Every time the word changes, the parity bit is set so as to make sure that the number of 1 bits is even.
  - This is just a convention, enforcing an odd number of 1 bits would also work, and is also used.

Parity Bits - Examples

- Size of original word: *m* = 8.

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101		
00110000		
11100001		
01011110		

Parity Bits - Examples

- Size of original word: *m* = 8.

Original Word (8 bits)	Number of 1s in Original Word	Codeword (9 bits): Original Word + Parity Bit
01101101	5	011011011
00110000	2	001100000
11100001	4	111000010
01011110	5	010111101

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?
- How can a single bit be corrupted?

Parity Bit: Detecting A 1-Bit Error

- Suppose now that indeed the memory work has been corrupted in a single bit.
- How can we use the parity bit to detect that?
- How can a single bit be corrupted?
  - Either it was a 1 that turned to a 0.
  - Or it was a 0 that turned to a 1.
- Either way, the number of 1-bits either increases by 1 or decreases by 1, and **becomes odd**.
- The error detection code just has to check if the number of 1-bits is even.

Error Detection Example

- Size of original word: *m* = 8.
- Suppose that the error detection algorithm gets as input one of the bit patterns on the left column. What will be the output?

Input: Codeword (9 bits): Original Word + Parity Bit	Number of 1s	Error?
011001011		
001100000		
100001010		
010111110		

Error Detection Example

- Size of original word: *m* = 8.
- Suppose that the error detection algorithm gets as input one of the bit patterns on the left colum. What will be the output?

Input: Original Word + Parity Bit (9 bits)	Number of 1s	Error?
011001011	5	yes
001100000	2	no
100001010	3	yes
010111110	6	no

Parity Bit and Multi-Bit Errors

- What if two bits get corrupted?
- The number of 1-bits can:
  - remain the same, or
  - increase by 2, or
  - decrease by 2.
- In all cases, the number of 1-bits remains even.
- The error detection algorithm will not catch this error.
- That is to be expected, a single parity bit is only good for detecting a single-bit error.

More General Methods

- Up to the previous slide, we discussed a very simple error detection method, namely **using a single parity bit.**
- We know move on to more general methods, that possibly detect and/or correct multiple errors.
  - For that, we need multiple extra bits.
- Key parameters:
  - *m*: the number of bits in the original memory word.
  - *r*: the number of extra (also called *redundant*) bits.
  - *n*: the total number of bits per codeword: $n = m + r$.
  - *d*: the number of errors we want to be able to detect or correct.

Legal and Illegal Codewords

- Each *m*-bit original word corresponds to **only one** *n*-bit codeword.
- A codeword is called **legal** if an original m-bit word corresponds to that codeword.
- A codeword is called **illegal** if **no** original m-bit word corresponds to that codeword.
- How many possible original words are there?
- How many possible codewords are there?
- How many **legal** codewords are there? In other words, how many codewords are possible to observe if there are no errors?

Legal and Illegal Codewords

- Each *m*-bit original word corresponds to **only one** *n*-bit codeword.
- A codeword is called **legal** if an original m-bit word corresponds to that codeword.
- A codeword is called **illegal** if **no** original m-bit word corresponds to that codeword.
- How many possible original words are there? 2^m.
- How many possible codewords are there? 2^n.
- How many **legal** codewords are there? In other words, how many codewords are possible to observe if there are no errors? 2^m.

Legal and Illegal Codewords

- How many possible original words are there? 2^m.
- How many possible codewords are there? 2^n.
- How many **legal** codewords are there? In other words, how many codewords are possible to observe if there are no errors? 2^m.
- Therefore, most (2^n-2^m) codewords are illegal, and only show up in the case of errors.
- The set of legal codewords is called a **code**.

The Hamming Distance

- Suppose we have two codewords *A* and *B*.
- Each codeword is an *n*-bit binary pattern.
- We define the distance between A and B to be the number of bit positions where A and B differ.
- This is called the **Hamming distance**.
- One way to compute the Hamming distance:
  - Let *C* = EXCLUSIVE OR(*A, B*).
  - Hamming Distance(*A*, *B*) = number of 1-bits in *C*.
- Given a code (i.e., the set of legal codewords), we can find the pair of codewords with the smallest distance.
- We call this minimum distance the **distance of the code.**

Hamming Distance: Example

- What is the Hamming distance between these two patterns?

```
1 0 1 1 0 1 0 0 1 0 0 0
0 0 1 1 0 1 0 1 1 0 1 0
```

- How can we measure this distance?

Hamming Distance: Example

- What is the Hamming distance between these two patterns?

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

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

- How can we measure this distance?
  - Find all positions where the two bit patterns differ.
  - Count all those positions.
- Answer: the Hamming distance in the example above is 3.

Example: 2-Bit Error Detection

- Size of original word: *m = 3*.
- Number of redundant bits: *r = 3.*
- Size of codeword: *n = 6.*
- Construction:
  - 1 parity bit for bits 1, 2.
  - 1 parity bit for bits 1, 3.
  - 1 parity bit for bits 2, 3.
- You can manually verify that you cannot find any two codewords with Hamming distance 2 (just need to manually check 28 pairs).
- This is a code with distance 3.
- Any 2-bit error can be detected.

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Example: 2-Bit Error Detection

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?
001100	
101011	
110011	
011110	
111110	
101101	
010011	
011000	

- Suppose that the error detection algorithm takes as input bit patterns as shown on the right table.
- What will be the output? How is it determined?

Example: 2-Bit Error Detection

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?
001100	Yes
101011	Yes
110011	No
011110	No
111110	Yes
101101	No
010011	Yes
011000	Yes

- Suppose that the error detection algorithm takes as input bit patterns as shown on the right table.
- The output simply depends on whether the input codeword is a legal codeword, as listed on the left table.

Example: 1-Bit Error Correction

- Size of original word: *m = 3*.
- Number of redundant bits: *r = 3.*
- Size of codeword: *n = 6.*
- Construction:
  - 1 parity bit for bits 1, 2.
  - 1 parity bit for bits 1, 3.
  - 1 parity bit for bits 2, 3.
- You can manually verify that you cannot find any two codewords with Hamming distance 2 (just need to manually check 28 pairs).
- This is a code with distance 3.
- Any 1-bit error can be corrected.

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codeword	Output (original word)
110101			
101000			
110011			
011110			
000010			
101101			
001111			
000110			

- Suppose that the error detection algorithm takes as input bit patterns as shown on the right table.
- What will be the output? How is it determined?

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codeword	Output (original word)
110101	Yes	010101	010
101000	Yes	111000	111
110011	No	110011	110
011110	No	011110	011
000010	Yes	000000	000
101101	No	101101	101
001111	Yes	001011	001
000110	Yes	100110	100

- The error detection algorithm:
  - Finds the legal codeword that is most similar to the input.
  - If that legal codeword is not equal to the input, there was an error!
  - Outputs the original word that corresponds to that legal codeword.

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codewords	Output (original word)
001100			

- What happens in this case?

Example: 1-Bit Error Correction

Original Word	Codeword
000	000000
001	001011
010	010101
011	011110
100	100110
101	101101
110	110011
111	111000

Input Codeword	Error?	Most Similar Codewords	Output (original word)
001100	Yes	000000 011110 101101	More than 1 bit corrupted, cannot correct!

- No legal codeword is within distance 1 of the input codeword.
- 3 legal codewords are within distance 2 of the input codeword.
- More than 1 bit have been corrupted, the error has been detected, but cannot be corrected.

Significance of Code Distances

- To detect up to d single-bit errors, we need a code with Hamming distance at least d+1. Why?
- When does an error fail to get detected?

Significance of Code Distances

- To detect up to d single-bit errors, we need a code with Hamming distance at least d+1. Why?
- When does an error fail to get detected?
  - When, due to bad luck, the error changes a legal codeword to another legal codeword.
- With a code of distance d+1, what is the smallest number of single-bit errors that can change a legal codeword to another legal codeword?

Significance of Code Distances

- To detect up to d single-bit errors, we need a code with Hamming distance at least d+1. Why?
- When does an error fail to get detected?
  - When, due to bad luck, the error changes a legal codeword to another legal codeword.
- With a code of distance d+1, what is the smallest number of single-bit errors that can change a legal codeword to another legal codeword?
  - d+1.
- Thus, d or fewer single-bit errors are guaranteed to produce an illegal codeword, and thus will be detected.

Correcting *d* Single-Bit Errors

- To correct *d* or fewer single-bit errors, we need a code of distance at least *2d + 1*. Why?

Correcting *d* Single-Bit Errors

- To correct *d* or fewer single-bit errors, we need a code of distance at least *2d + 1*. Why?
- What would be a good algorithm to use for error correction, if we have a code of distance *2d + 1*?
- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
  - If no error has occurred, output = input.
- Steps:

Correcting *d* Single-Bit Errors

- To correct *d* or fewer single-bit errors, we need a code of distance at least $2d + 1$. Why?
- What would be a good algorithm to use for error correction, if we have a code of distance $2d + 1$?
- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
    - Comment: If no error has occurred, output = input.
- Steps:
    - Find, among the 2^m legal codewords, the most similar to the input.
    - Return that most similar codeword as output.

Correcting *d* Single-Bit Errors

- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
- Error correction algorithm:
  - Find, among the 2^m legal codewords, the most similar to the input.
  - Return that most similar codeword as output.
- If the distance of the code is 2d+1, why would this algorithm correct up to *d* single-bit errors?

Correcting *d* Single-Bit Errors

- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
- Error correction algorithm:
  - Find, among the 2^m legal codewords, the most similar to the input.
  - Return that most similar codeword as output.
- If the distance of the code is 2d+1, why would this algorithm correct up to *d* single-bit errors?
- Suppose we have a legal codeword A, that gets *d* or fewer single-bit errors, and becomes codeword B.
- What is the most similar legal codeword to B?

Correcting *d* Single-Bit Errors

- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
- Error correction algorithm:
  - Find, among the 2^m legal codewords, the most similar to the input.
  - Return that most similar codeword as output.
- If the distance of the code is 2d+1, why would this algorithm correct up to *d* single-bit errors?
- Suppose we have a legal codeword A, that gets *d* or fewer single-bit errors, and becomes codeword B.
- What is the most similar legal codeword to B?
- It has to be A.
  - The distance from B to A is at most *???*.
  - The distance from B to any other legal codeword is at least *???*.

Correcting *d* Single-Bit Errors

- Input: *n*-bit codeword (may be corrupted or not).
- Output: *n*-bit corrected codeword.
- Error correction algorithm:
  - Find, among the 2^m legal codewords, the most similar to the input.
  - Return that most similar codeword as output.
- If the distance of the code is 2d+1, why would this algorithm correct up to *d* single-bit errors?
- Suppose we have a legal codeword A, that gets *d* or fewer single-bit errors, and becomes codeword B.
- What is the most similar legal codeword to B?
- It has to be A.
  - The distance from B to A is at most *d*.
  - The distance from B to any other legal codeword is at least *d+1*.

Correcting a Single-Bit Error

- The previous approaches are not **constructive**.
- We didn't say anywhere:

Correcting a Single-Bit Error

- The previous approaches are not **constructive**.
- We didn't say anywhere:
  - How many extra bits we need to obtain a *d+1* distance code or a *2d+1* distance code.
  - How to actually define the codewords for such a code.
- Now we will explicitly define a method for correcting a single-bit error.

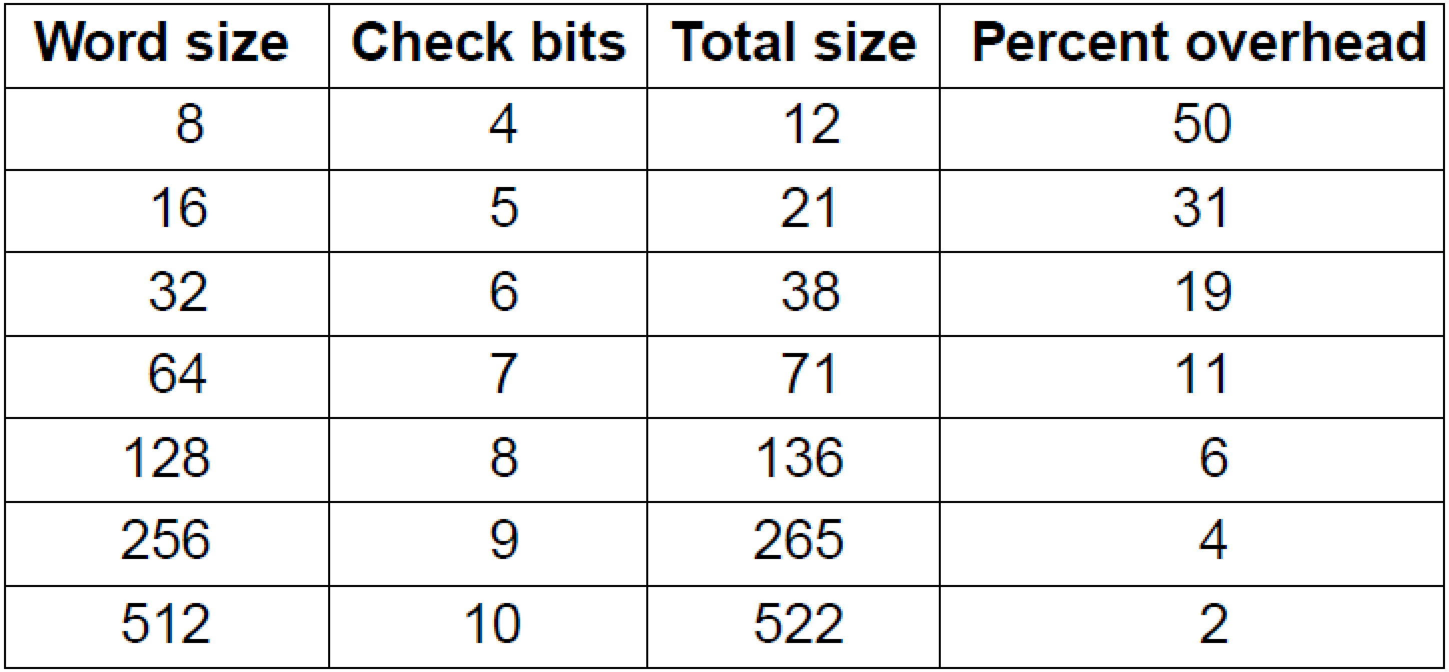
Correcting a Single-Bit Error

- Suppose that A is a legal *n-bit* codeword.
- Suppose that now A gets a single-bit-error, and becomes B.
- Given A, how many possible values are there for B?
  - n, one for every possible location of the bit that changed.
- Thus, to be able to correct single-bit errors, there must be at least *n+1* codewords (legal or illegal) that the error correction algorithm will map to codeword A:
  - A itself, and the *n* codewords that differ from A by a single bit.
- We have 2^m legal codewords, and we need at least *n+1* codewords for each legal codeword, thus we need at least $(n+1)2^m$ codewords.

Correcting a Single-Bit Error

- Thus, we have two equations, that we can solve:
  - *$(n+1)\ 2^m <= 2^n$*.
  - *$n = m + r$*.
- From the above equations, given *m* (the number of bits in the original memory word), we obtain:
  - a lower bound for *r* (the number of extra bits we need to add to each word).
  - a lower bound for *n* (the number of bits in each codeword).

Table of Bits Needed



Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Number of check bits for a code
that can correct a single error.

Hamming's Algorithm

- Hamming's Algorithm can correct a single-bit error.
- Suppose we have a 16-bit word.
  - Based on the previous equations (and table), we need 5 extra bits, for a total of 21 bits.
- Let's number these 21 bits as bit 1, bit 2, ..., bit 21.
  - We break from our usual convention, where numbering starts at 0.
- The five parity bits are placed at positions 1, 2, 4, 8, 16.
  - Positions corresponding to powers of 2.
- Each parity bit will check some (but not all) of the 21 bits.

Hamming's Algorithm

- The five parity bits are placed at positions *1, 2, 4, 8, 16*.
- Each parity bit will check some (but not all) of the 21 bits.
- Some bits may be checked by multiple parity bits.
- To determine which parity bits will check the bit at position *p*, we:
  - write *p* in binary. We need 5 digits. We get $d_5\ d_4\ d_3\ d_2\ d_1$.
  - For each d_i, if $d_i = 1$ then position *p* is checked by the parity bit at position 2^{i-1}.
- Example: position *18* is written in binary as *10010*.
- Since $d_5 = 1$, bit *18* is checked by parity bit *16* ($16 = 2^4$).
- Since $d_2 = 1$, bit *18* is checked by parity bit *2* ($2 = 2^1$).

Assigning Bits to Parity Bits

- By following the previous process for every single bit, we arrive at the following:
  - Parity bit 1 checks bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.
  - Parity bit 2 checks bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.
  - Parity bit 4 checks bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.
  - Parity bit 8 checks bits 8, 9, 10, 11, 12, 13, 14, 15.
  - Parity bit 16 checks bits 16, 17, 18, 19, 20, 21.
- Thus, each parity bit is set to 0 or 1, so as to ensure that the total number of 1-bits (among the bits that this parity bit checks) is even.

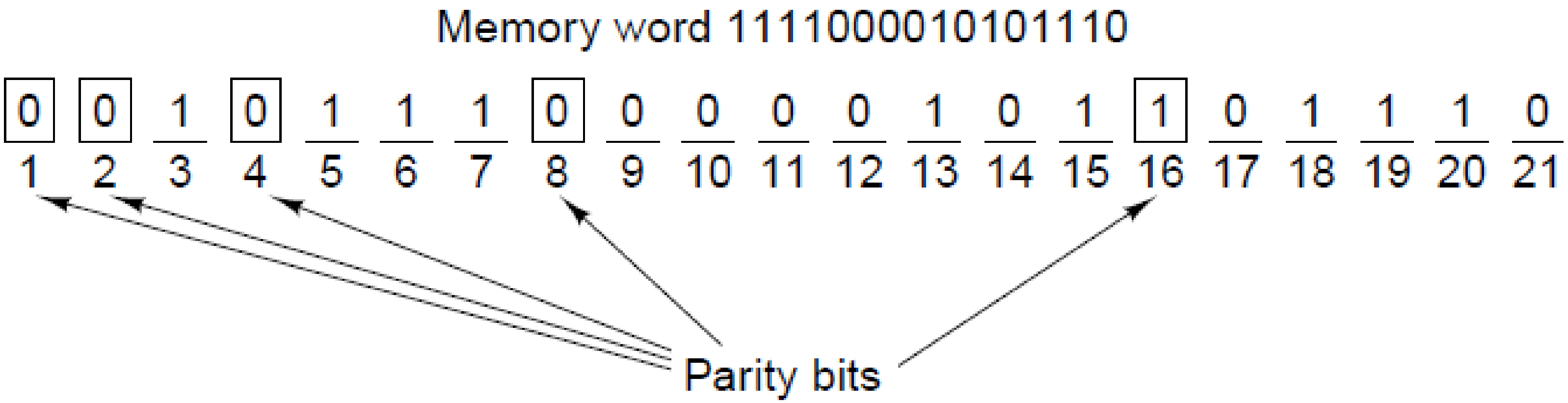
Correcting an Error

- Suppose now that a single-bit error has occurred.
- Will that be detected?
- Yes. One or more of the parity bits will be wrong.
  - What does this mean that a parity bit is wrong? It means that, among the bits that this parity bit checks, the total number of 1-bits is odd.
- How do we figure out the position of the error?
- We just need to add the positions of the parity bits that are wrong.

Proof That This Works?

- It is a bit complicated to get an elegant proof that Hamming's algorithm works.
- We can prove it by case-by-case examination.
- Pick any subset of the parity bits to be wrong. You can check manually that:
  - An error in the bit computed by Hamming's algorithm will lead to exactly that subset of parity bits to be wrong.
  - An error in any other bit will lead to a different subset of parity bits being wrong.

An Example Codeword



Memory word 1111000010101110

0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	1	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Parity bits

Construction of the Hamming code
for the memory word 1111000010101110 by
adding 5 check bits to the 16 data bits.

From Word to Codeword: Example 1

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value			1		1	1	1		1	0	0	0	1	0	1		0	1	1	1	0
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in original word = ??   Bit 1 value = ??
- Bit 2: number of 1s in original word = ??   Bit 1 value = ??
- Bit 4: number of 1s in original word = ??   Bit 1 value = ??
- Bit 8: number of 1s in original word = ??   Bit 1 value = ??
- Bit 16: number of 1s in original word = ??   Bit 1 value = ??

From Word to Codeword: Example 1

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	1	0	1	1	1	1	1	0	0	0	1	0	1	1	0	1	1	1	0
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in original word = 7. Bit 1 value = 1.
- Bit 2: number of 1s in original word = 6. Bit 2 value = 0.
- Bit 4: number of 1s in original word = 6. Bit 4 value = 0.
- Bit 8: number of 1s in original word = 3. Bit 8 value = 1.
- Bit 16: number of 1s in original word = 3. Bit 16 value = 1.

From Word to Codeword: Example 2

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value			0		1	0	1		0	1	1	1	0	0	0		1	0	0	0	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in original word = ?? Bit 1 value = ??
- Bit 2: number of 1s in original word = ?? Bit 1 value = ??
- Bit 4: number of 1s in original word = ?? Bit 1 value = ??
- Bit 8: number of 1s in original word = ?? Bit 1 value = ??
- Bit 16: number of 1s in original word = ?? Bit 1 value = ??

From Word to Codeword: Example 2

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	1	0	0	1	0	1	1	0	1	1	1	0	0	0	0	1	0	0	0	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in original word = 5. Bit 1 value = 1.
- Bit 2: number of 1s in original word = 3. Bit 2 value = 1.
- Bit 4: number of 1s in original word = 4. Bit 4 value = 0.
- Bit 8: number of 1s in original word = 3. Bit 8 value = 1.
- Bit 16: number of 1s in original word = 2. Bit 16 value = 0.

Error Correction: Example 1

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = ??
- Bit 2: number of 1s in codeword = ??
- Bit 4: number of 1s in codeword = ??
- Bit 8: number of 1s in codeword = ??
- Bit 16: number of 1s in codeword = ??

Error Correction: Example 1

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = 6. OK
- Bit 2: number of 1s in codeword = 5. ERROR
- Bit 4: number of 1s in codeword = 4. OK
- Bit 8: number of 1s in codeword = 2. OK
- Bit 16: number of 1s in codeword = 5. ERROR

Position of error:

Error Correction: Example 1

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	0	1	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = 6. OK
- Bit 2: number of 1s in codeword = 5. ERROR
- Bit 4: number of 1s in codeword = 4. OK
- Bit 8: number of 1s in codeword = 2. OK
- Bit 16: number of 1s in codeword = 5. ERROR

Position of error:

16+2 = 18

Error Correction: Example 2

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	1	1	1	0	0	1	0	0	0	1	1	0	1	0	0	0	0	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = ??
- Bit 2: number of 1s in codeword = ??
- Bit 4: number of 1s in codeword = ??
- Bit 8: number of 1s in codeword = ??
- Bit 16: number of 1s in codeword = ??

Error Correction: Example 2

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	1	1	1	0	0	1	0	0	0	1	1	0	1	0	0	0	0	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = 6. OK
- Bit 2: number of 1s in codeword = 2. OK
- Bit 4: number of 1s in codeword = 7. ERROR
- Bit 8: number of 1s in codeword = 4. OK
- Bit 16: number of 1s in codeword = 2. OK

Position of error:

Error Correction: Example 2

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Value	1	0	1	1	1	0	0	1	0	0	0	1	1	0	1	0	0	0	0	1	1
Bit 1 checks	*		*		*		*		*		*		*		*		*		*		*
Bit 2 checks		*	*			*	*			*	*			*	*			*	*		
Bit 4 checks				*	*	*	*					*	*	*	*					*	*
Bit 8 checks								*	*	*	*	*	*	*	*						
Bit 16 checks																*	*	*	*	*	*

- Bit 1: number of 1s in codeword = 6. OK
- Bit 2: number of 1s in codeword = 2. OK
- Bit 4: number of 1s in codeword = 7. ERROR
- Bit 8: number of 1s in codeword = 4. OK
- Bit 16: number of 1s in codeword = 2. OK

Position of error:

Bit 4

Summary

- Memory hierarchy
  - Caches
  - Main memory
  - Disk / storage
    - Virtual memory
  - Dependable memory: error-correcting codes

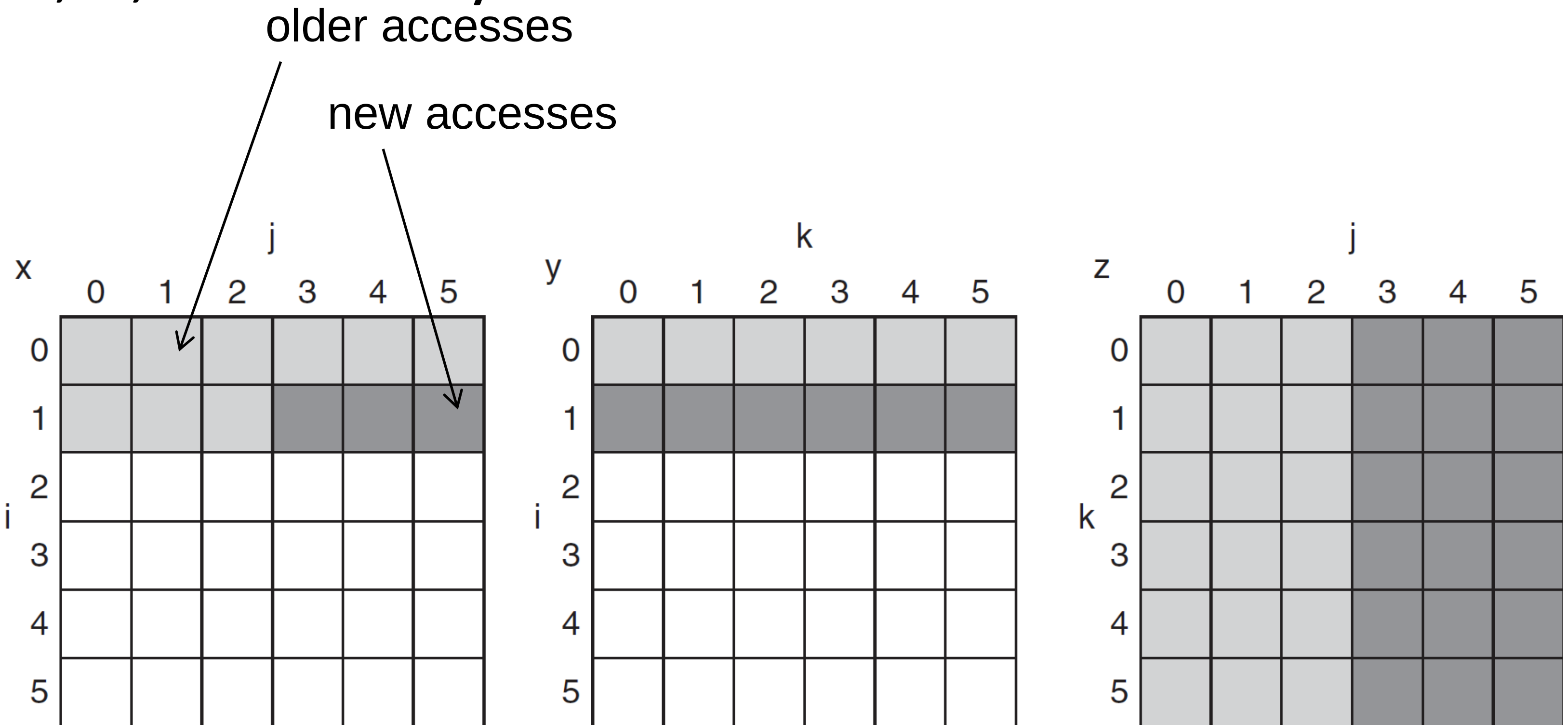
Software Optimization via Blocking

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];
  for( int k = 0; k < n; k++ )
    cij += A[i+k*n] * B[k+j*n];
  C[i+j*n] = cij;
}
```

DGEMM Access Pattern

- C, A, and B arrays



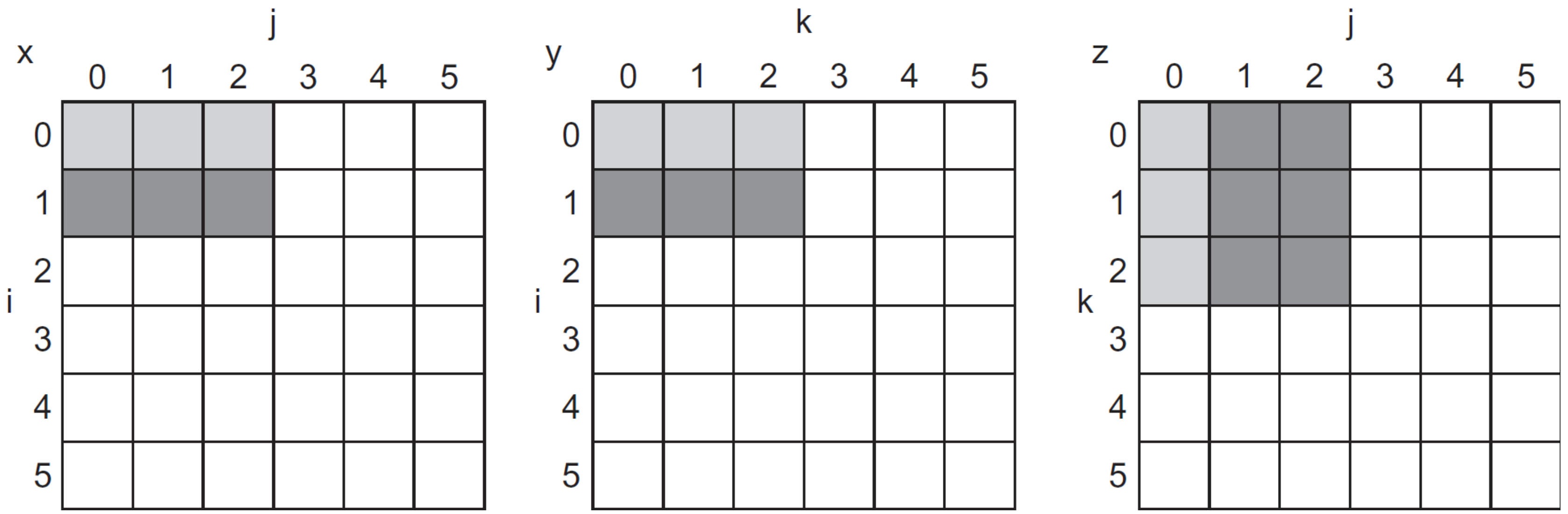
older accesses

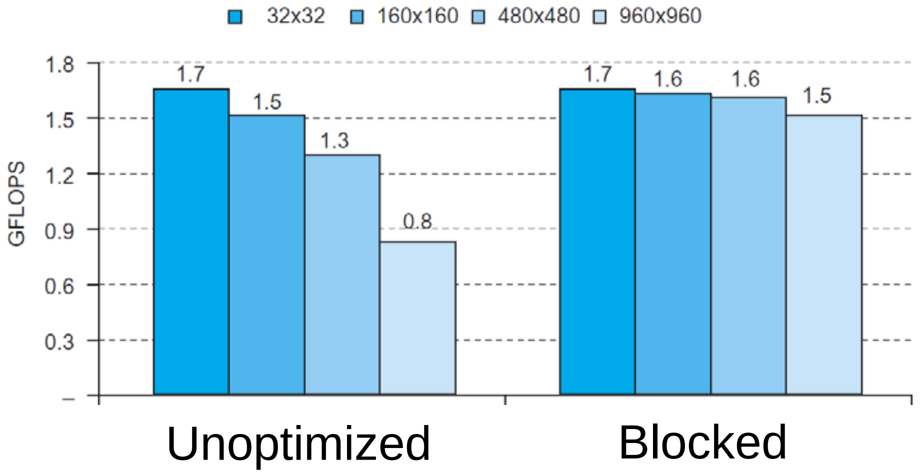
new accesses

Cache Blocked DGEMM

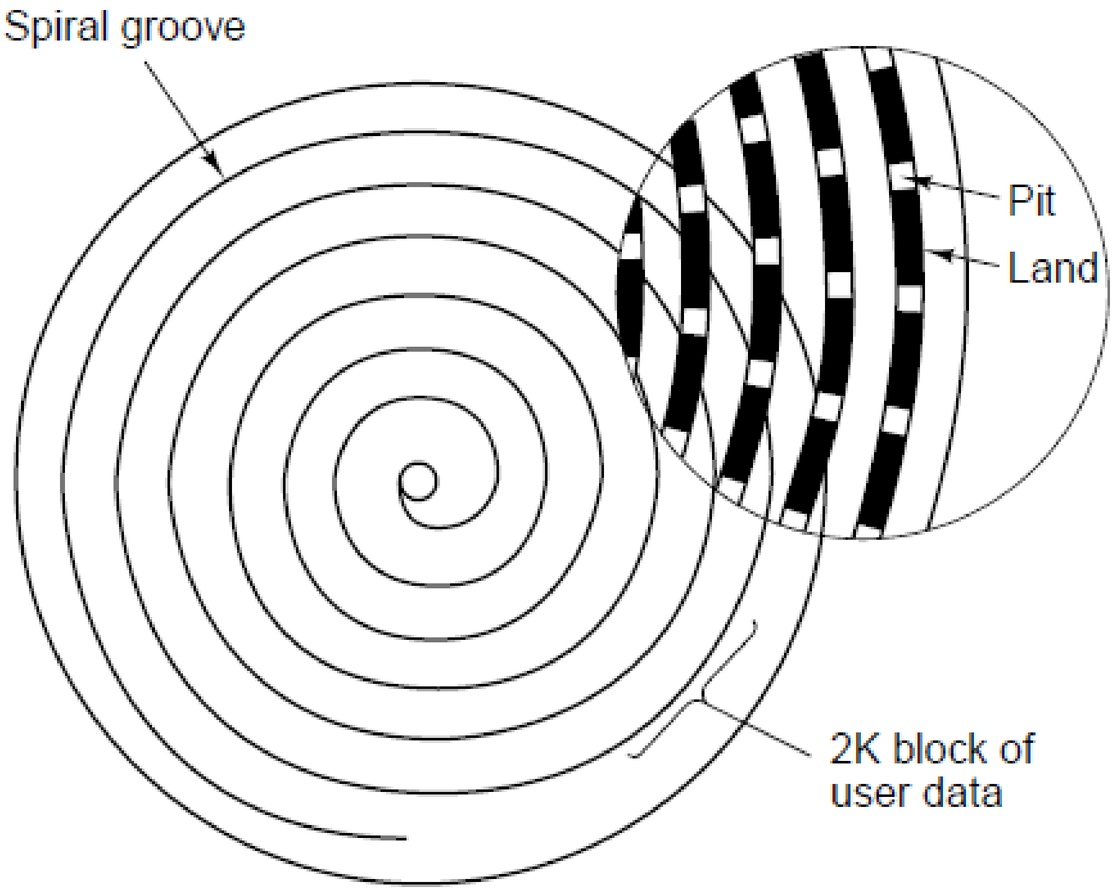
```
#define BLOCKSIZE 32
void do_block (int n, int si, int sj, int sk, double *A, double
*B, double *C)
{
 for (int i = si; i < si+BLOCKSIZE; ++i)
  for (int j = sj; j < sj+BLOCKSIZE; ++j)
  {
   double cij = C[i+j*n];/* cij = C[i][j] */
   for( int k = sk; k < sk+BLOCKSIZE; k++ )
    cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
   C[i+j*n] = cij;/* C[i][j] = cij */
  }
 }
 void dgemm (int n, double* A, double* B, double* C)
 {
  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
   for ( int si = 0; si < n; si += BLOCKSIZE )
    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
     do_block(n, si, sj, sk, A, B, C);
 }
```

Blocked DGEMM Access Pattern

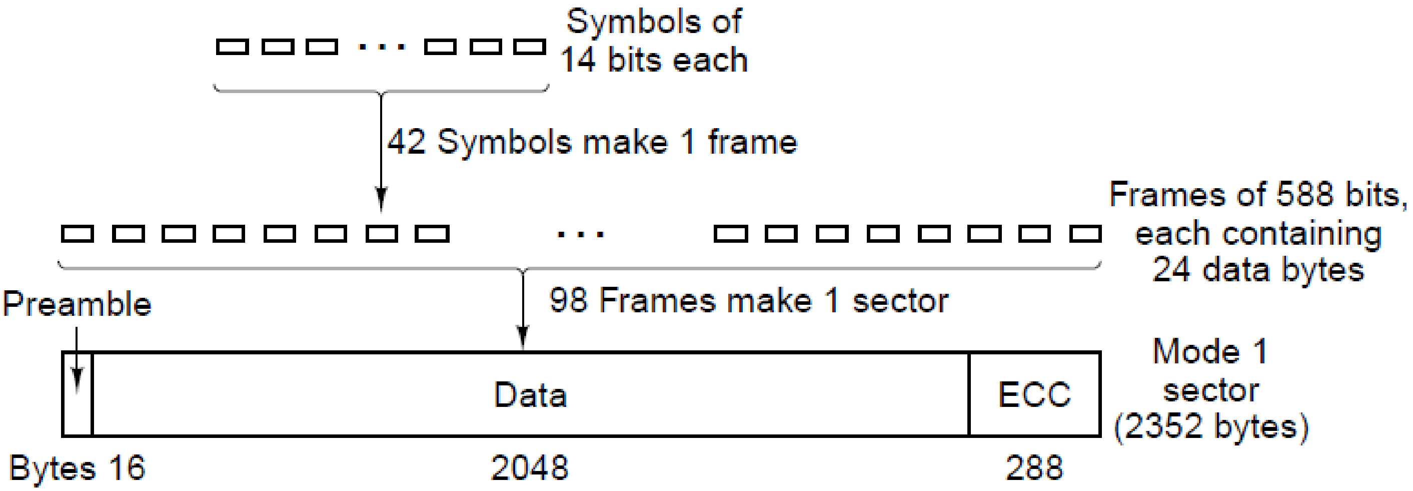




CDs

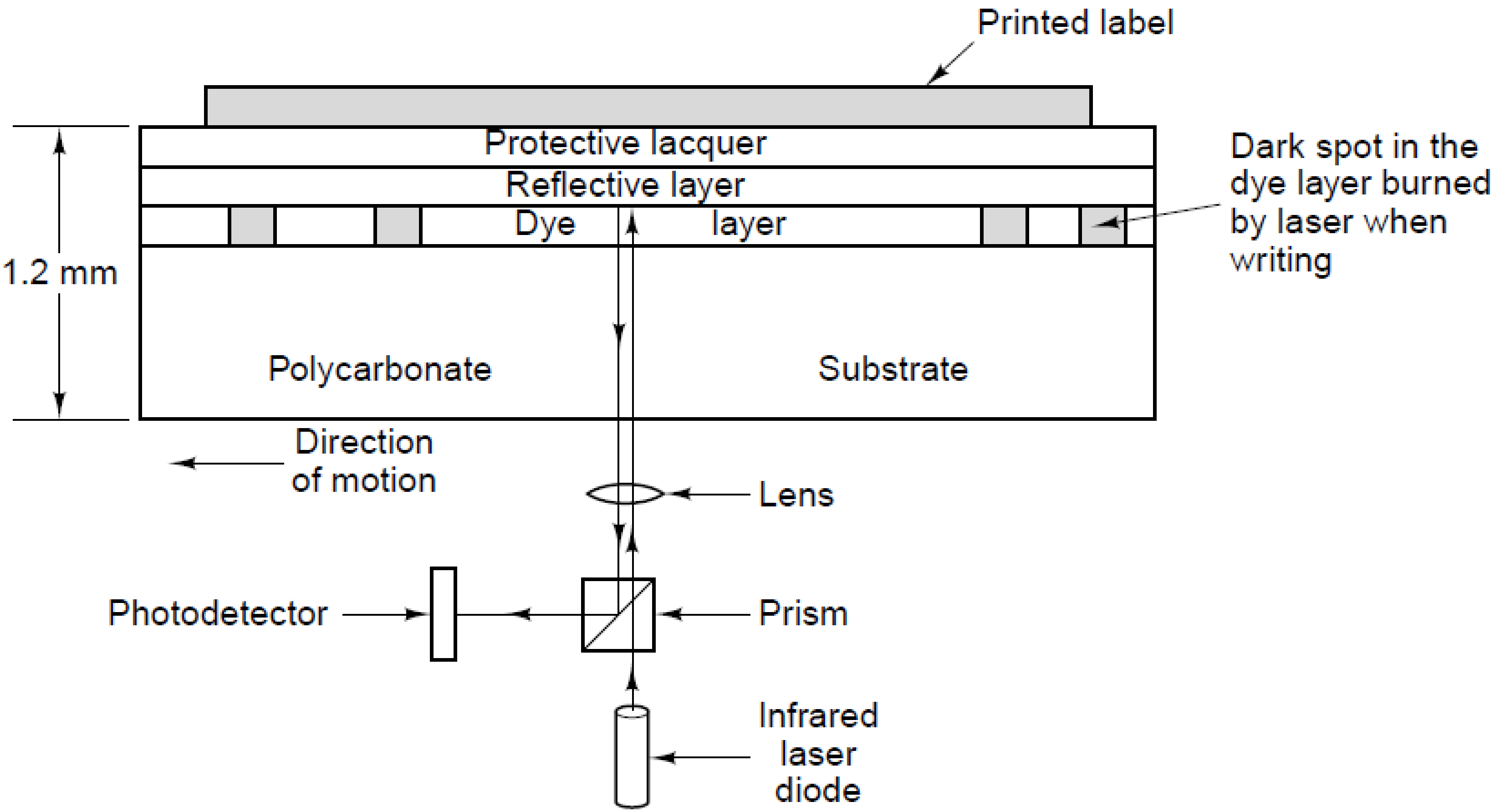


CDs



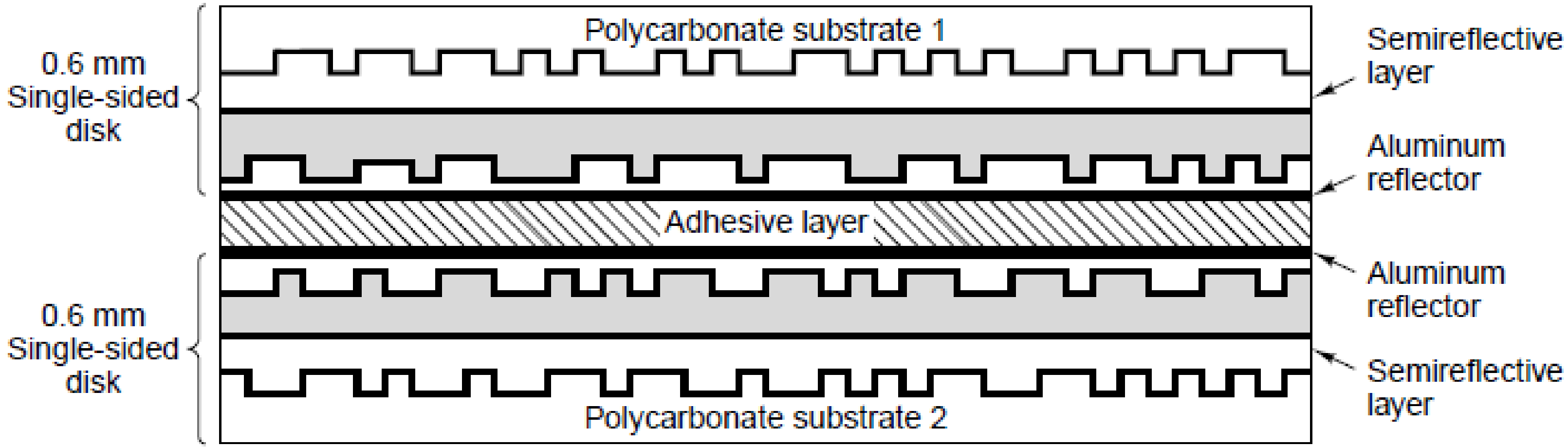
- Mode 1
  - 16 bytes preamble, 2048 bytes data, 288 bytes error-correcting code
  - Single Speed CD-ROM: 75 sectors/sec, so data rate: 75*2048=153,600 bytes/sec
  - 74 minutes audio CD: Capacity: 74*60*153,600=681,984,000 bytes ~=650 MB
- Mode 2
  - 2336 bytes data for a sector, 75*2336=175,200 bytes/sec

CD-R

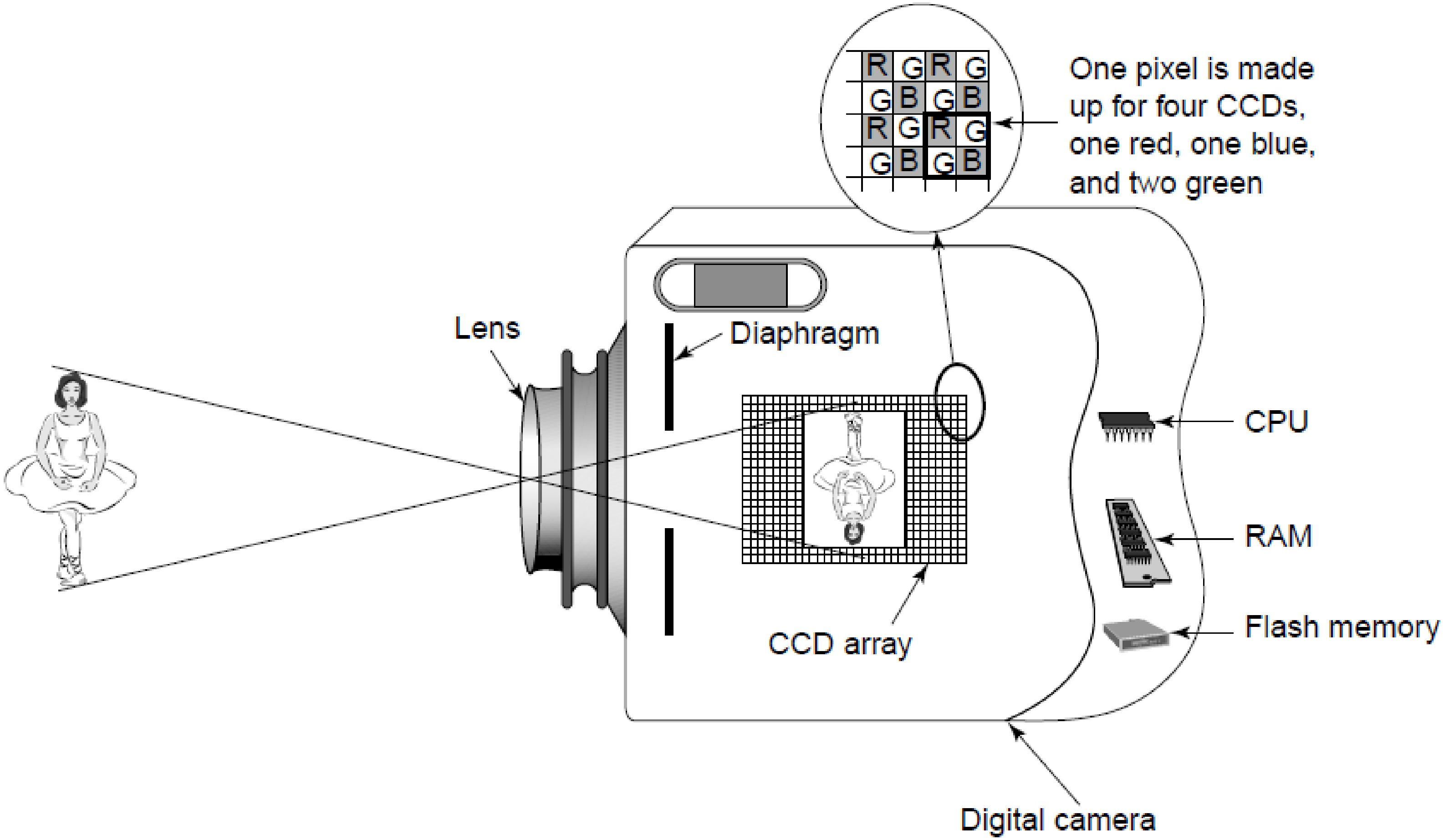

Printed label
Protective lacquer
Reflective layer
Dye layer
Dark spot in the dye layer burned by laser when writing
1.2 mm
Polycarbonate
Substrate
Direction of motion
Lens
Photodetector
Prism
Infrared laser diode

DVDs

- Single-sided, single-layer (4.7 GB)
- Single-sided, dual-layer (8.5 GB)
- Double-sided, single-layer (9.4 GB)
- Double-sided, dual-layer (17 GB)



Storing Images



Optical Disks

- Disks in this family include:
  - CDs, DVDs, Blu-ray disks.
- The basic technology is similar, but improvements have led to higher capacities and speeds.
- Optical disks are much slower than magnetic drives.
- These disks are a cheap option for write-once purposes.
  - Great for mass distribution of data (software, music, movies).
- CD capacity: 650-700MB.
  - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
  - Minimum data rate: 1.4MB/sec.
- Blu-ray capacity: 25GB-50GB.
  - Minimum data rate: 4.5MB/sec.

Optical Disk Capacities

- CD capacity: 650-700MB.
  - Minimum data rate: 150KB/sec.
- DVD capacity: 4.7GB to 17GB.
  - Minimum data rate: 1.4MB/sec.
  - Single-sided, single-layer: 4.7GB.
  - Single-sided, dual-layer: 8.5GB.
  - Double-sided, single-layer: 9.4GB.
  - Double-sided, dual-layer: 17GB.
- Blu-ray capacity: 25GB-50GB.
  - Minimum data rate: 4.5MB/sec.
  - Single-sided: 25GB.
  - Double-sided: 50GB.

Magnetic Disks

- Consists of one or more platters with magnetizable coating
- Disk head containing induction coil floats just over the surface
- When a positive or negative current passes through head, it magnetizes the surface just beneath the head, aligning the magnetic particles face right or left, depending on the polarity of the drive current
- When head passes over a magnetized area, a positive or negative current is induced in the head, making it possible to read back the previously stored bits
- Track
  - Circular sequence of bits written as disk makes complete rotation
  - Sector: Each track is divided into some sector with fixed length

Classical Hard Drives: Magnetic Disks

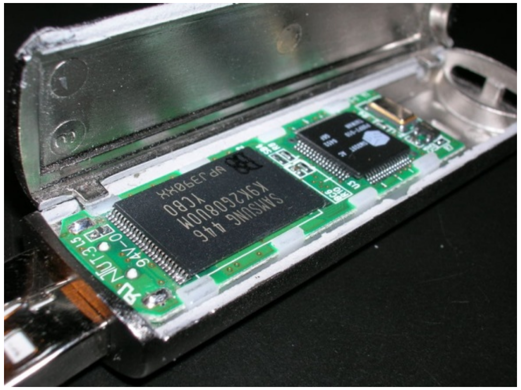
- A magnetic disk is a disk, that spins very fast.
  - Typical rotation speed: 5400, 7200, 10800 RPMs.
  - RPMs: rotations per minute.
  - These translate to 90, 120, 180 rotations per second.
- The disk is divided into rings, that are called **tracks**.
- Data is read by the **disk head**.
  - The head is placed at a specific radius from the disk center.
  - That radius corresponds to a specific track.
  - As the disk spins, the head reads data from that track.

Solid-State Drives

- A solid-state drive (SSD) is NOT a spinning disk. It is just cheap memory.
- Compared to hard drives, SSDs have two to three times faster speeds, and **~100nsec access time.**
- Because SSDs have no mechanical parts, they are well-suited for mobile computers, where motion can interfere with the disk head accessing data.
- Disadvantage #1: price.
  - Magnetic disks: pennies/gigabyte.
  - SSDs: one to three dollars/gigabyte.
- Disadvantage #2: failure rate.
  - A bit can be written about 100,000 times, then it fails.

Flash Storage

- Nonvolatile semiconductor storage
  - 100× – 1000× faster than disk
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

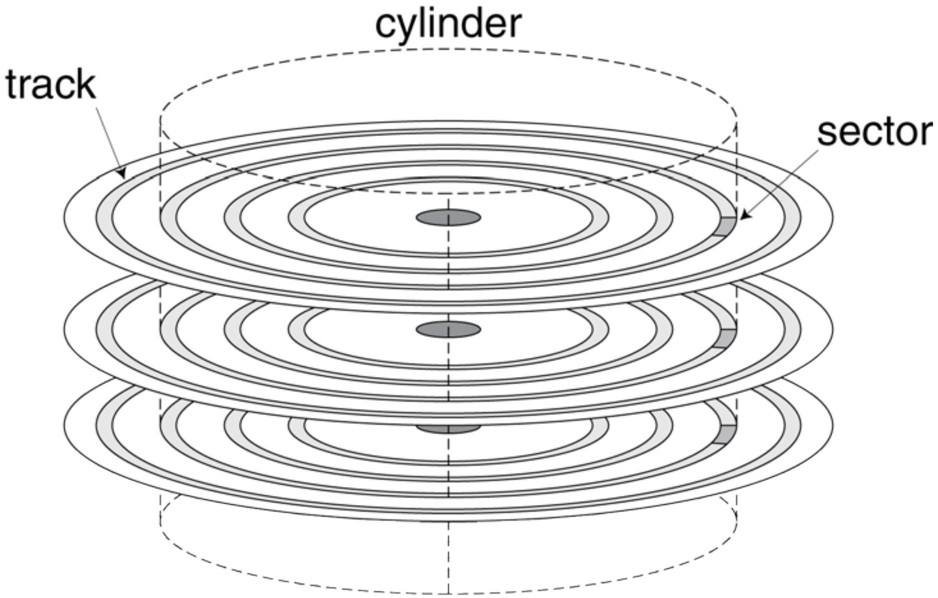


Flash Types

- 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 (bits/area), but block-at-a-time access
  - Cheaper per GB
  - Used for USB keys, media storage, …
- Flash bits wears out after 1000’s of accesses
  - Not suitable for direct RAM or disk replacement
  - Wear leveling: remap data to less used blocks

Disk Storage

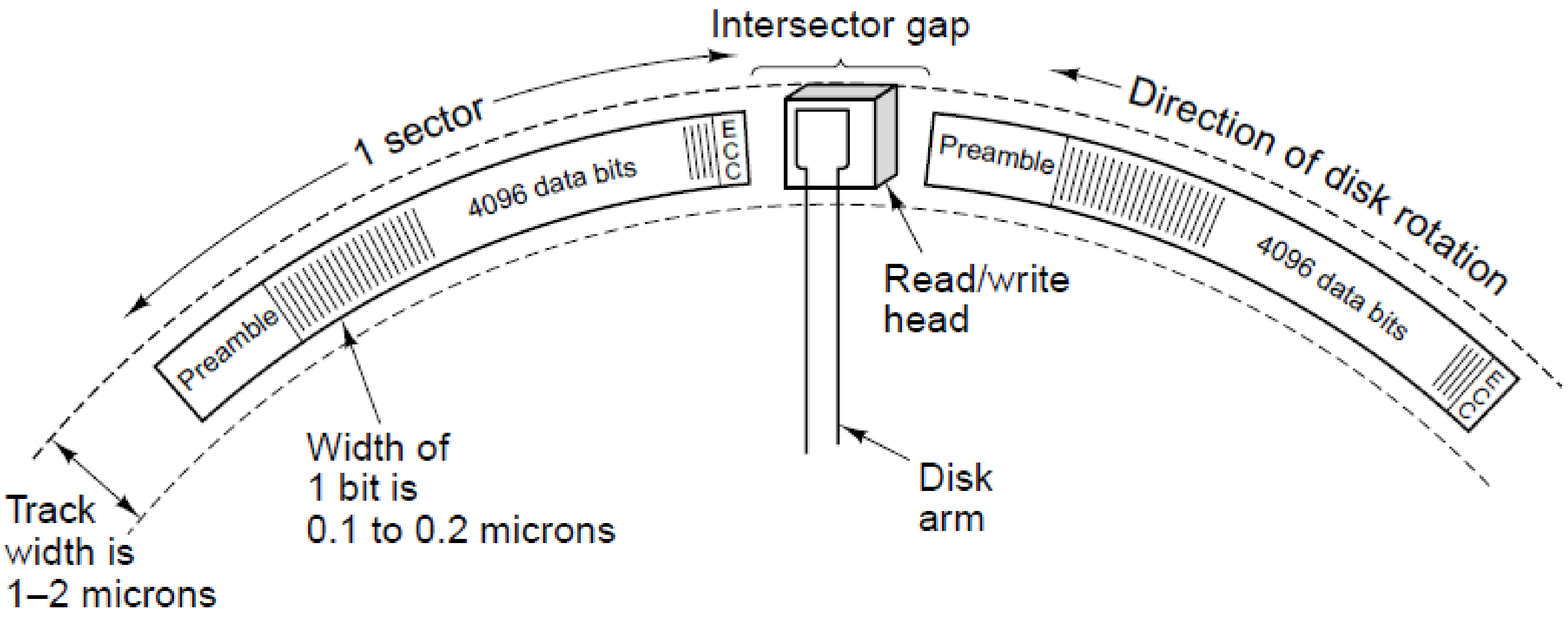
- Nonvolatile, rotating magnetic storage



Disk Tracks and Sectors

- A track can be 0.2µm wide.
  - We can have 50,000 tracks per cm of radius.
  - About 125,000 tracks per inch of radius.
- Each track is divided into fixed-length **sectors**.
  - Typical sector size: 512 bytes.
- Each sector is preceded by a **preamble**. This allows the head to be synchronized before reading or writing.
- In the sector, following the data, there is an error-correcting code.
- Between two sectors there is a small **intersector gap**.

Visualizing a Disk Track



A portion of a disk track. Two sectors are illustrated.

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: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

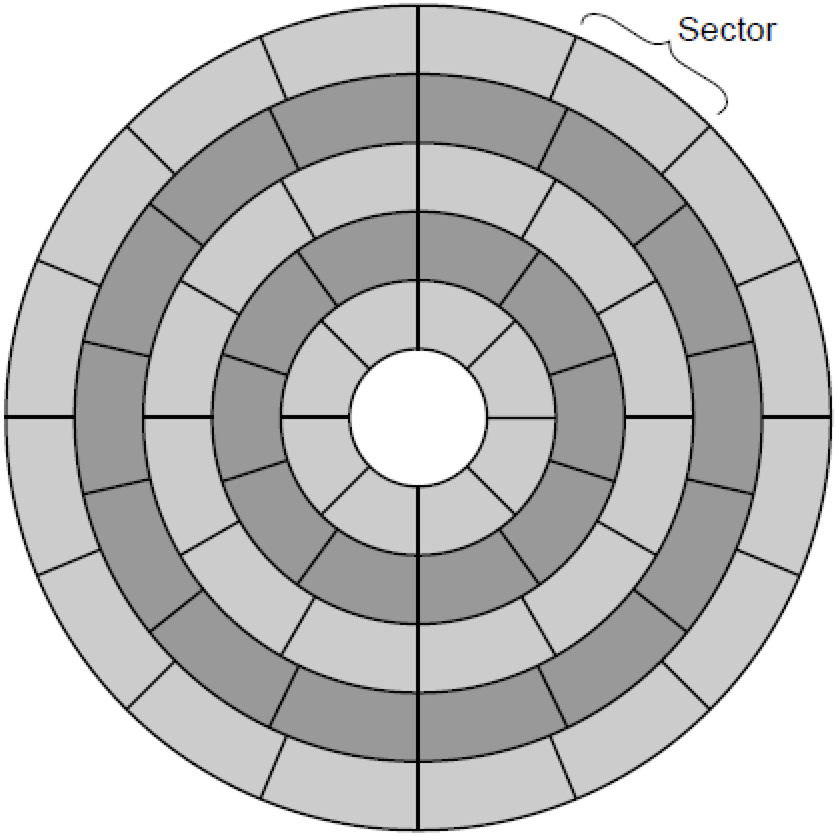
Disk Access Example

- 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
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.2ms
- If actual average seek time is 1ms
  - Average read time = 3.2ms

Disk Performance Issues

- Manufacturers quote average seek time
  - Based on all possible seeks
  - Locality and OS scheduling lead to smaller actual average seek times
- Smart disk controller allocate physical sectors on disk
  - Present logical sector interface to host
  - SCSI, ATA, SATA
- Disk drives include caches
  - Prefetch sectors in anticipation of access
  - Avoid seek and rotational delay

Magnetic Disk Sectors



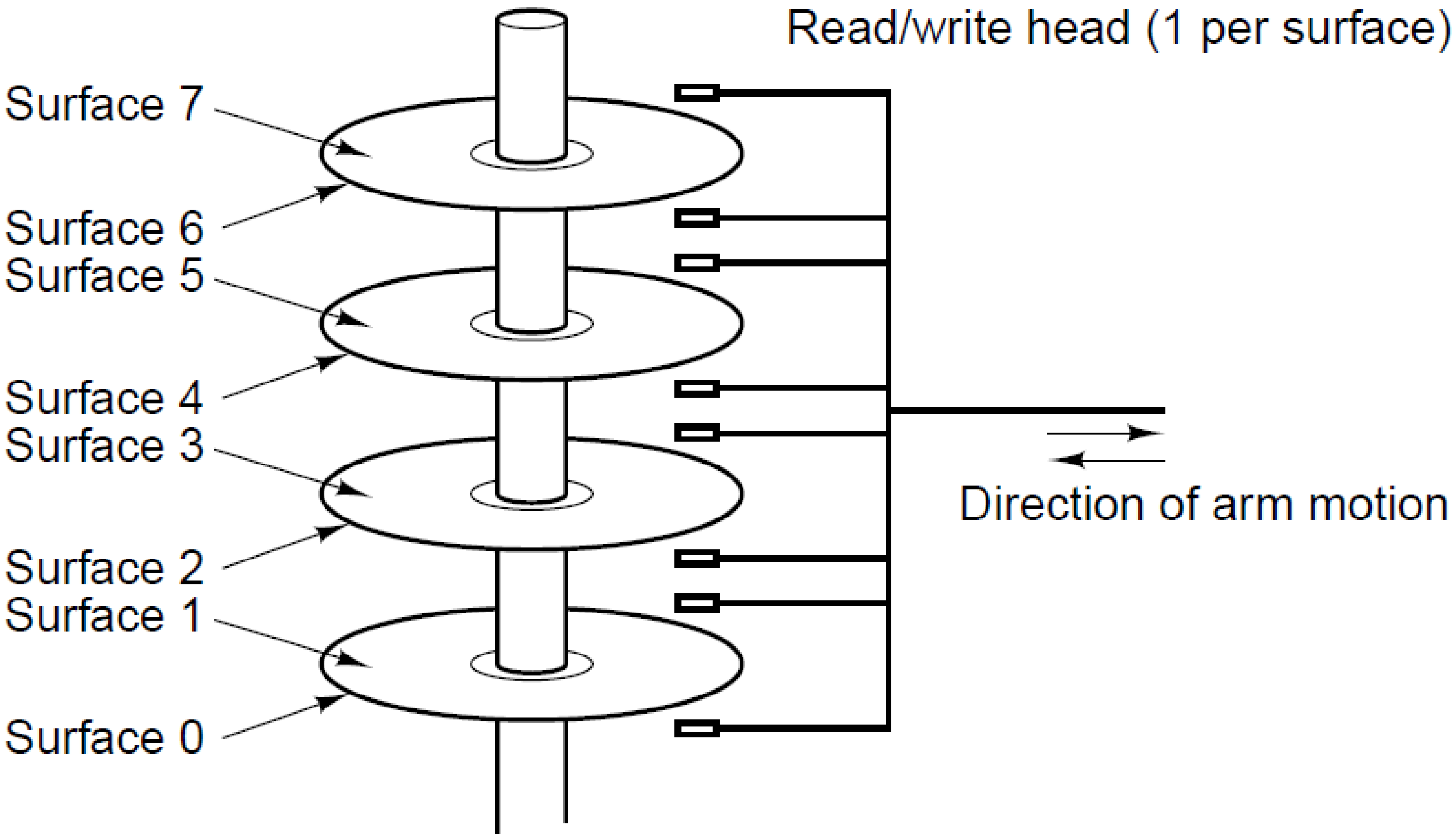
Measuring Disk Capacity

- Disk capacity is often advertized in unformatted state.
- However, <u>**formatting**</u> takes away some of this capacity.
  - Formatting creates preambles, error-correcting codes, and gaps.
- The formatted capacity is typically about 15% lower than unformatted capacity.

Multiple Platters

- A typical hard drive unit contains multiple platters, i.e., multiple actual disks.
- These platters are stacked vertically (see figure).
- Each platter stores information on both surfaces.
- There is a separate arm and head for each surface.

Magnetic Disk Platters



Cylinders

- The set of tracks corresponding to a specific radial position is called a **cylinder**.
- Each track in a cylinder is read by a different head.

Data Access Times

- Suppose we want to get some data from the disk.
- First, the head must be placed on the right track (i.e., at the right radial distance).
  - This is called **seek**.
  - Average seek times are in the 5-10 msec range.
- Then, the head waits for the disk to rotate, so that it gets to the right sector.
  - Given that disks rotate at 5400-10800 RPMs, this incurs an average wait of 3-6 msec. This is called **rotational latency**.
- Then, the data is read. A typical rate for this stage is 150MB/sec.
  - So, a 512-byte sector can be read in ~3.5 μsec.

Measures of Disk Speed

- **Maximum Burst Rate**: the rate (number of bytes per sec) at which the head reads a sector, **once the had has started seeing the first data bit**.
  - This excludes seeks, rotational latencies, going through preambles, error-correcting codes, intersector gaps.
- **Sustained Rate**: the actual average rate of reading data over several seconds, that includes all the above factors (seeks, rotational latencies, etc.).

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read ???/sec?

Worst Case Speed

- Rarely advertised, but VERY IMPORTANT to be aware of if your software accesses the hard drive: **the worst case speed**.
- What scenario gives us the worst case?
  - Read random positions, one byte at a time.
  - To read each byte, we must perform a seek, wait for the rotational latency, go through the sector preamble, etc.
- If this whole process takes about 10 msec (which may be a bit optimistic), we can only read 100 bytes/sec.
  - More than a million times slower than the maximum burst rate.

Worst Case Speed

- Reading a lot of non-contiguous small chunks of data kills magnetic disk performance.
- When your programs access disks a lot, it is important to understand how disk data are read, to avoid this type of pitfall.

Disk Controller

- The disk controller is a chip that controls the drive.
  - Some controllers contain a full CPU.
- Controller tasks:
  - Execute commands coming from the software, such as:
    - READ
    - WRITE
    - FORMAT (writing all the preambles)
  - Control the arm motion.
  - Detect and correct errors.
  - Buffer multiple sectors.
  - Cache sectors read for potential future use.
  - Remap bad sectors.

IDE and SCSI Drives

- IDE and SCSI drives are the two most common types of hard drives on the market.
- Just be aware that:
  - IDE drives are cheaper and slower.
    - Newer IDE drives are also called serial ATA or SATA.
  - SCSI drives are more expensive and faster.
- Most inexpensive computers use IDE drives.

RAID

- RAID stands for *Redundant Array of Inexpensive Disks.*
- RAID arrays are simply sets of disks, that are visible as a single unit by the computer.
  - Instead of a single drive accessible via a drive controller, the whole RAID is accessible via a RAID controller.
  - Since a RAID can look as a single drive, software accessing disks does not need to be modified to access a RAID.
- Depending on their type (we will see several types), RAIDs accomplish one (or both) of the following:
  - Speed up performance.
  - Tolerate failures of entire drive units.

RAID for Faster Speed

- Disk performance has not improved as dramatically as CPU performance.
- In the 1970s, average seek times on minicomputer disks were 50-100 msec.
- Now they have improved to 5-10 msec.
- The slow gains in performance have motivated people to look into ways to gain speed via parallel processing.

RAID-0

- RAID level 0: Improves speed via **striping**.
  - When a write request comes in, data is broken into strips.
  - Each strip is written to a different drive, in round-robin fashion.
  - Thus, multiple strips are written in parallel, effectively leading to faster speed, compared to using a single drive.
- Effect: most files are stored in a distributed manner: with different pieces of them stored on each drive of the RAID.
- When reading a file, the different pieces (strips) are read again in parallel, from all drives.

RAID-0 Example

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???
- What is the best case scenario, in which performance will be the best, compared to a single disk?

- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: ???
  - The read performance of RAID-0 is: ???

RAID-0 Example

- Suppose we have a RAID-0 system with 8 disks.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing large chunks of data, so striping can be exploited.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-0 is: 8 times faster than a single disk.
  - The read performance of RAID-0 is: 8 times faster than a single disk.
- What is the best case scenario, in which performance will be the best, compared to a single disk?
  - Reading/writing many small, unrelated chunks of data (e.g., a single byte at a time). Then, striping cannot be used.
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-0 is: the same as that of a single disk.
  - The read performance of RAID-0 is: the same as that of a single disk.

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is ??? hours?

RAID-0: Pros and Cons

- RAID-0 works the best for large read/write requests.
- RAID-0 speed deteriorates into that of a single drive if the software asks for data in chunks of one strip (or less) at a time.
- How about reliability? A RAID-0 is **less** reliable, and more prone to failure than that of a single drive.
  - Suppose we have a RAID with four drives.
  - Each drive has a mean time to failure of 20,000 hours.
  - Then, the RAID has a mean time to failure that is only 5000 hours.
- RAID-0 is not a "true" RAID, no drive is redundant.

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is:
  - The read performance is:
  - Reliability is:

RAID-1

- In RAID-1, we need to have an even number of drives.
- For each drive, there is an identical copy.
- When we write data, we write it to both drives.
- When we read data, we read from either of the drives.
- NO STRIPING IS USED.
- Compared to a single disk:
  - The write performance is: twice as slow.
  - The read performance is: the same.
  - Reliability is: far better, drive failure is not catastrophic.

The Need for RAID-5.

- RAID-0: great for performance, bad for reliability.
  - striping, but no redundant data.
- RAID-1: bad for performance, great for reliability.
  - redundant data, no striping
- RAID-2, RAID-3, RAID-4: have problems of their own.
  - You can read about them in the textbook if you are curious, but they are not very popular.
- RAID-5: great for performance, great for reliability.
  - both redundant data and striping.

RAID-5

- Data is striped for writing.
- If we have N disks, we can process N-1 data strips in parallel.
- For every N-1 data strips, we create an Nth strip, called **parity strip**.
  - The k-th bit in the parity strip ensures that there is an even number of 1-bits in position k in all N strips.
- If any strip fails, its data can be recovered from the other N-1 strips.
- This way, the contents of an entire disk can be recovered.

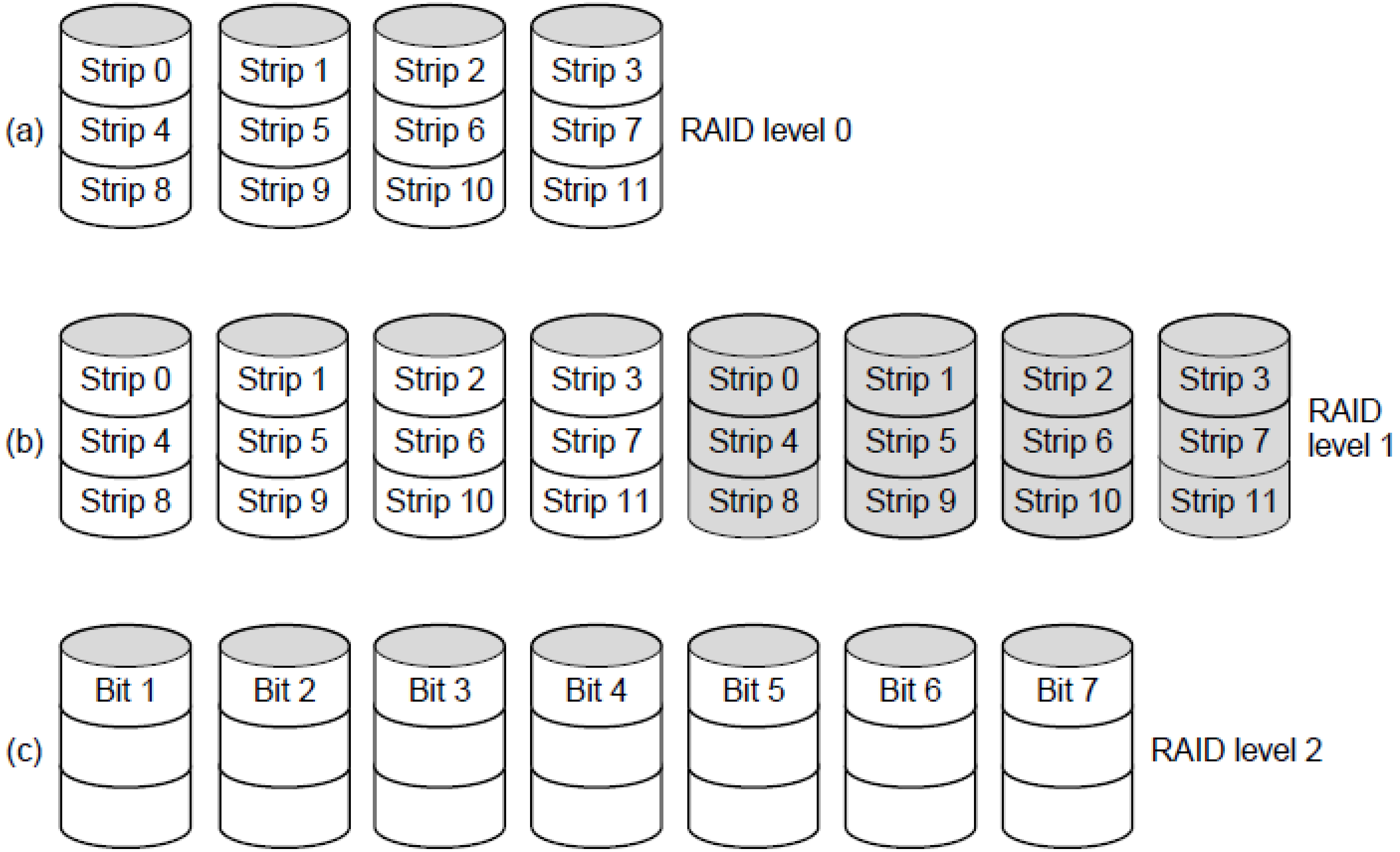
RAID-5 Example

- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: ???
  - The read performance of RAID-5 is: ???

RAID-5 Example

- Suppose we have a RAID-5 system with 8 disks.
- Compared to a single disk, in the **best** case:
  - The write performance of RAID-5 is: 7 times faster than a single disk. (writes non-parity data on 7 disks simultaneously).
  - The read performance of RAID-5 is: 7 times faster than a single disk. (reads non-parity data on 7 disks simultaneously).
- Compared to a single disk, in the **worst** case:
  - The write performance of RAID-5 is: the same as that of a single disk.
  - The read performance of RAID-5 is: the same as that of a single disk.
  - Why? Because striping is not useful when reading/writing one byte at a time.

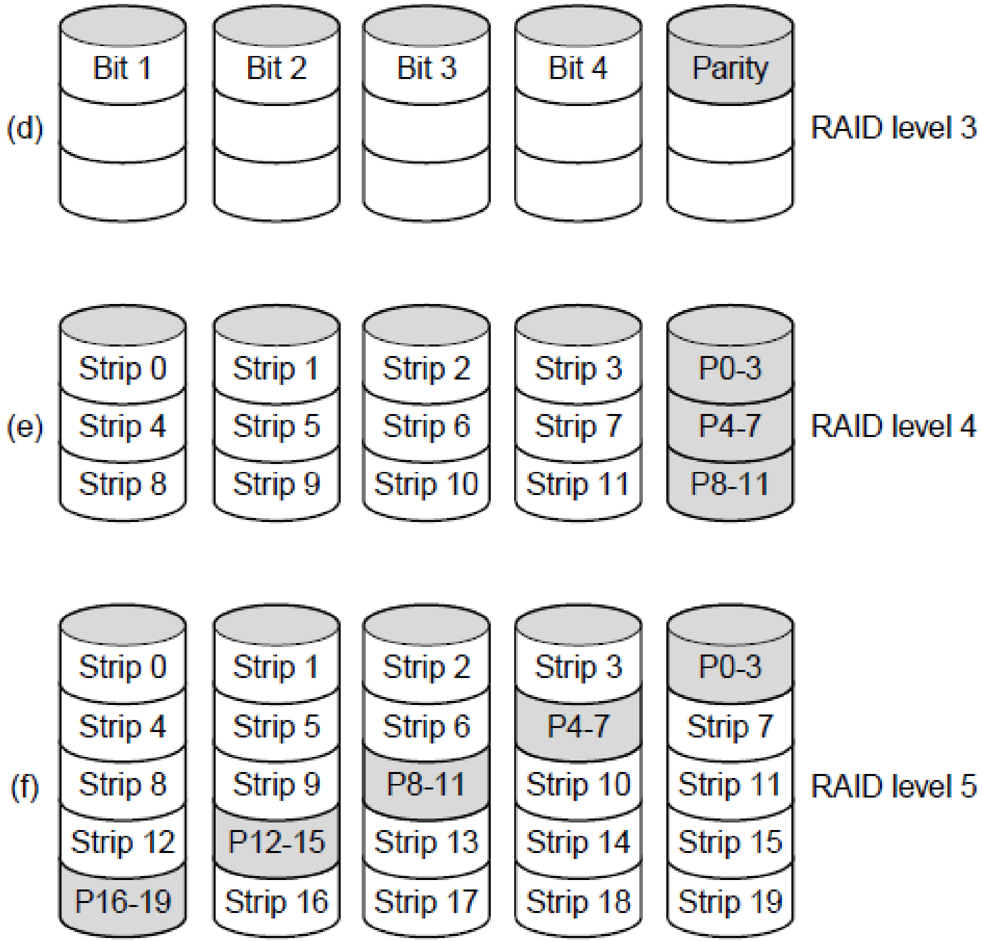
RAID-0, RAID-1, RAID-2



RAID levels 0 through 5. Backup and parity drives are shown shaded.

UNIVERSITY OF TEXAS ARLINGTON

RAID-3, RAID-4, RAID-5



RAID levels 0 through 5. Backup and parity drives are shown shaded.