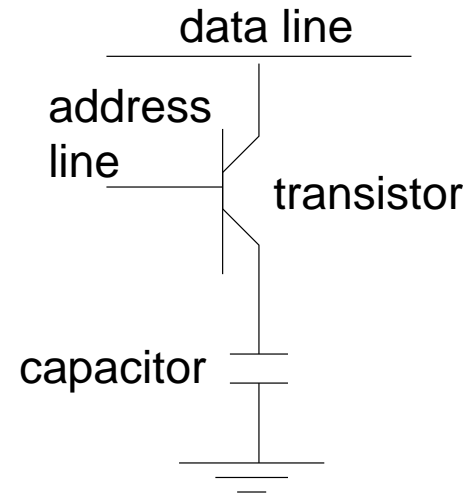


Memory Types

- SRAM (Static Random Access Memory)
 - basis is the D latch [Tanembaum, fig 3-23]
 - can hold 1 bit
 - requires 11 transistors (can be done in 6)
 - state persists providing power is on
- DRAM (Dynamic RAM)
 - requires 1 capacitor and 1 transistor!
 - capacitor stores just 40,000 electrons!
 - has a time constant of 10^{-3} sec
 - hence requires refresh (typically 10-100ms)
- comparison ([O'H&Bryant, fig 6.2])

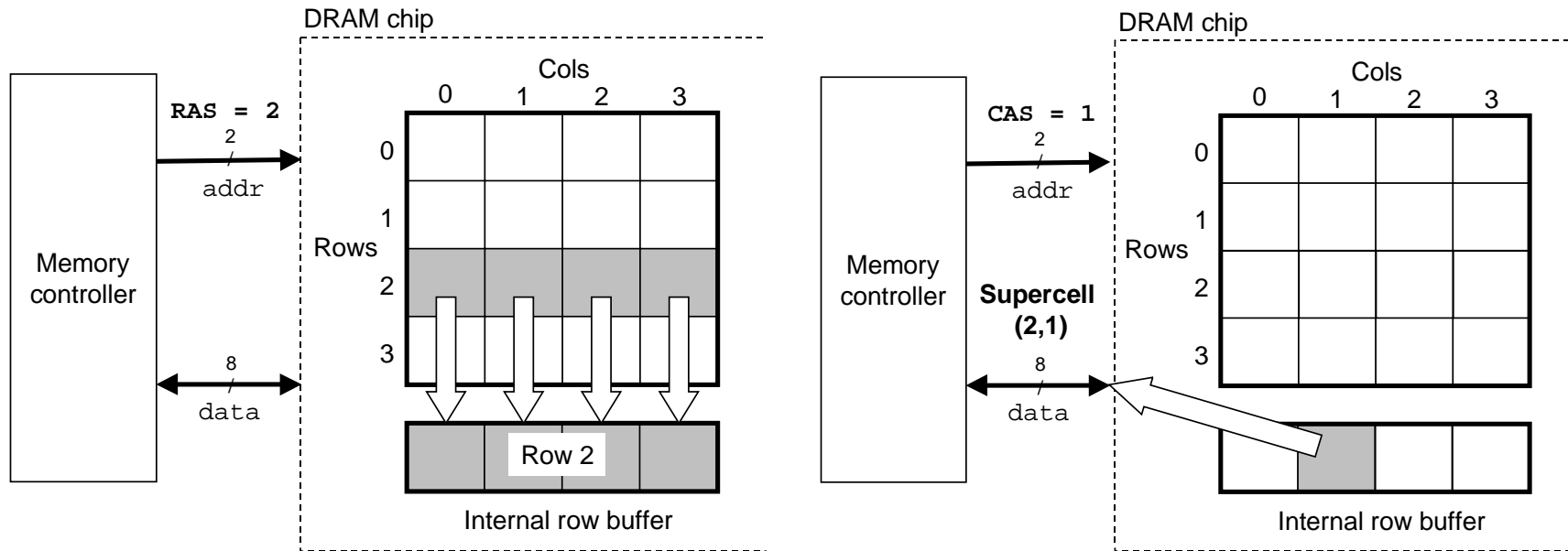


	transistors / bit	rel. access time	persistent	sensitive	rel. cost	use
SRAM	6	1×	yes	no	100×	?
DRAM	1	10×	no	yes	1×	?

- Read Only Memory (ROM) – non-volatile (state persists if power is off)
 - programs (e.g. the BIOS) stored in ROM are termed firmware

Organization and Access of (DRAM) Memory Chip

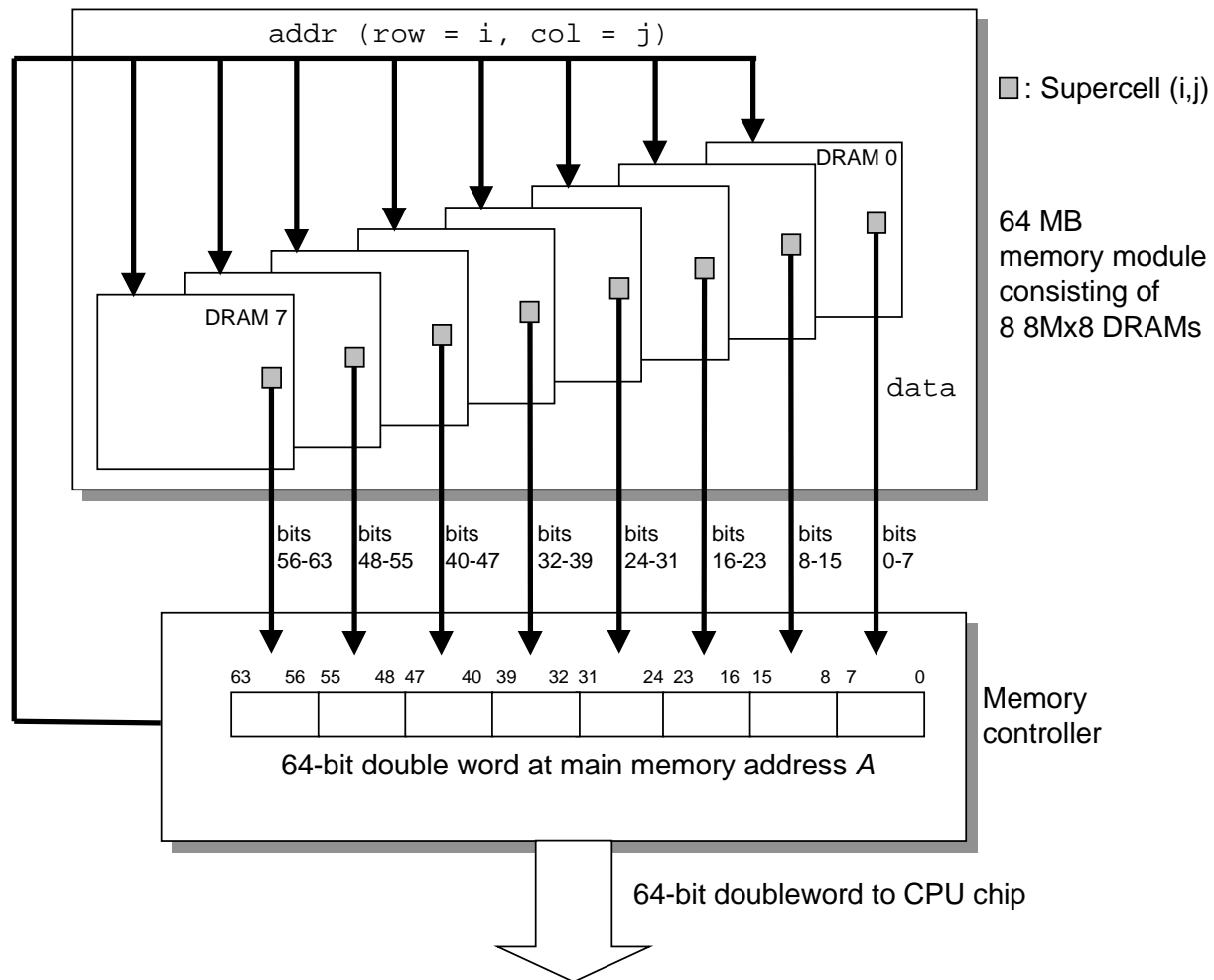
([O'H&Bryant, fig 6.4])



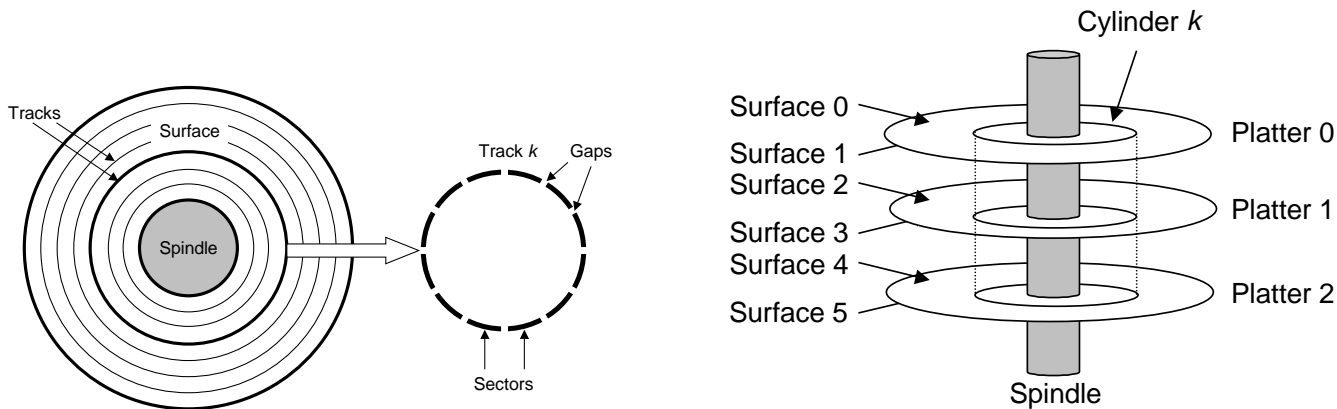
1. select row address (RAS = 2)
 2. select column address (CAS = 1)
 3. access bit
- why is address bus 2 bits wide?
 - why not a single index (0–15) per bit?

The Memory Module and Controller

Dual Inline Memory Module (DIMM, 168 pins; [O'H&Bryant, fig 6.5])



Disks: Anatomy, Capacity and Operation



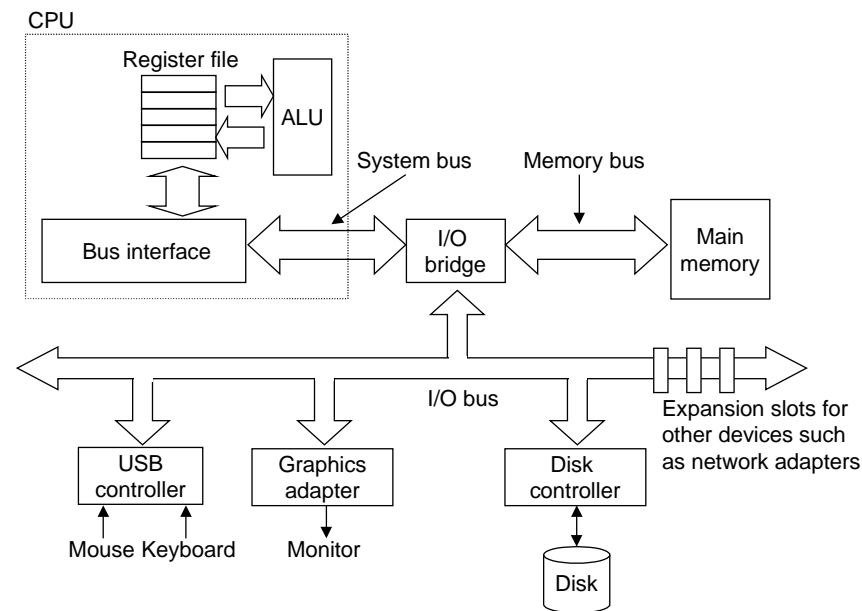
[O'H&Bryant, fig 6.9]: single platter

multiple platters

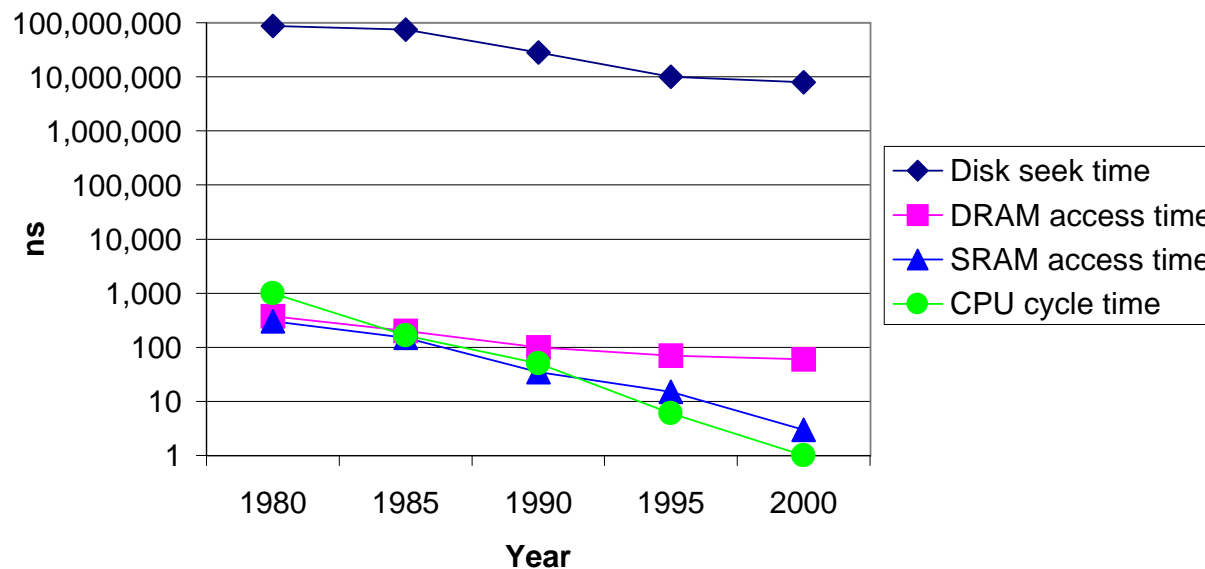
- coated with magnetic material; all tracks are equidistant from spindle
- typically sectors are 512 bytes, tracks have 400 sectors
- capacity = $\frac{\#bytes}{sector} \cdot \frac{avg. \#sectors}{track} \cdot \frac{\#tracks}{surface} \cdot \frac{\#surfaces}{platter} \cdot \frac{\#platters}{disk}$
- operation: read/write head connected to actuator arm
 - arm is 10^{-4} mm above surface and moves at 80km/h!
 - hence disks come in airtight containers

Disk Access Times and Configuration

- total access time is made up from:
 - seek time: move arm to track (typically 9 ms)
 - rotation latency: wait till sector reaches head (typically, with a 7200 RPM rotation rate, 4 ms)
 - transfer time per sector (typically, with 400 tracks/sector, 0.02 ms (25 MB/s))
- what dominates this? comparison with SRAM and DRAM?
- disk configuration [O'H&Bryant, fig 6.11] (will consider access details later under Module 0)



Trends in Memory Technologies



[O'H&Bryant, fig 6.16]:

The Take Home Message:

- different storage technologies have different prices and performance
- price / performance of different technologies is changing at different rates
- DRAM and disk access are lagging CPU cycle times
- how do we bridge the processor memory performance gap? (the 'memory wall')

Caching in the Broader Context

- recall the memory hierarchy [O’H&Bryant, fig 6.21]
- requires spatial and/or temporal locality of data accesses to be effective
 - data organized into ‘blocks’ to reduce management overheads
- this can be extended beyond a single computer system [O’H&Bryant, fig 6.23]:

type	what	where	latency (cycles)	managed by
registers	4-byte word	on-chip	0	compiler
TLB	address translation	on-chip	(0)	MMU / OS
L1 cache	32-byte block	on-chip	1	hardware
L2 cache	64-byte block	off-chip	10	hardware
virtual memory	(8KB) page	main memory	100	hardware / OS
buffer cache	parts of files	main memory	100	OS
network buffer cache	parts of files	main memory	10,000,000	AFS/NSF client
browser cache	web pages	local disk	10,000,000	web browser
web cache	web pages	remote server disks	1,000,000,000	web proxy server

Memory Systems

- in modern computer systems, we need **fast** memory access for **large** memories
- can achieve by: cost (per bit) - speed tradeoff:
 1. caches memory low
 2. wide memory access moderate
 3. faster technology (?) high
- effective CPU speed has improved faster than memory access speed because ... ?
- recall the two main memory technologies:

	cost/bit:	access time:	used in:
SRAM (Static RAM)	high	low ($\approx 5\text{ns}$)	off-chip caches
DRAM (Dynamic RAM)	low	high ($\approx 50\text{ns}$)	most main memories (low pinout)
- recall memory access involves the stages:
 1. select row address
 2. select column address
 3. (R/W) access selected bit(s)

(these can however be pipelined)

Cache Memory

- idea: data that is “*currently most needed*” is brought into a (smaller) faster memory
- observation: memory accesses in *most* programs exhibit:
 - temporal locality: if access address X , likely to access X again soon
 - spatial locality: if access address X , likely to access $X+1$ soon
 - ⇒ cache organized into lines (blocks) of L bytes ($L = 2^l$, e.g. $4 \leq l \leq 9$)
 - ✓ blocked memory accesses (faster) & less control info needed (per byte)
 - × redundant memory traffic if only ever use 1 byte per line
 - ★ e.g. pointer chasing (large address ‘strides’ guaranteed!)

if so, yields good cost-speed tradeoff

- cache hit rates: % of (word) accesses in program when data is in cache
 - need to be **high** (e.g. $> 95\%$) for good performance
 - only possible if accesses have a sufficient locality
- **problem**: keeping consistency of data cache & main memory:
 - when `store` instruction is executed, the relevant line is updated 1st;
when does main memory get updated?

Direct-Mapped Caches

- idea: for a cache of size $C' = 2^{c-l}$ lines,
all addresses with same a_1 are mapped to the same cache line

$$X = \begin{array}{|c|c|c|} \hline 31 & c & c-1 & l & l-1 & 0 \\ \hline & a_0 & & a_1 & & a_2 \\ \hline \end{array}$$

- ✓ easy to implement & low chip area / byte (note: cache must store value of a_0)
 \Rightarrow large C' possible (better performance)
- ✗ cache conflicts: 2 (or more) words from memory map to the same cache line
 can make large, often *unpredictable*, performance losses
- alternative: K-way set associative caches
 - like a direct-mapped cache of size C'/K , but every line extended to a set of K lines
 addresses with same a_1 can map into any of the K lines in the set
 - typically $K = 1, 2, 4, 6, 8$ (issue: replacement policy)
 - ✓ reduces chance of conflicts by factor of K
 - ✗ some extra cost / byte (more complex H/W needed)

Virtual Memory Concerns

- program addresses are virtual; H/W translates these into physical addresses
 - as for caches, this is (largely) transparent to the programmer
- on PeANUt, how many memory references are needed to execute a load instruction?
- real machines have (in their MMUs) a translation lookaside buffer (TLB) of T entries
 - effectively a 'cache' of recent VM translations
 - typically $T = 64 \Rightarrow$ a working set of > 64 pages causes TLB misses
 - ◆ usually causes an O.S. trap to access the page tables; costs $10^2..10^3$ cycles!
 - typically, TLBs are fully associative with an LRU replacement policy
- a working set of T pages is needed to minimize TLB misses
- good locality can also minimize page faults (the swap space is at the very bottom of the memory hierarchy!)
- list (at least) 6 similarities between caching and virtual memory