

Memory Systems

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

- two main memory technologies:

	cost/bit:	access time:	used in:
SRAM (Static RAM)	high	low (\approx 10ns)	off-chip caches
DRAM (Dynamic RAM)	low	high (\approx 50ns)	most memories (low pinout)

- 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

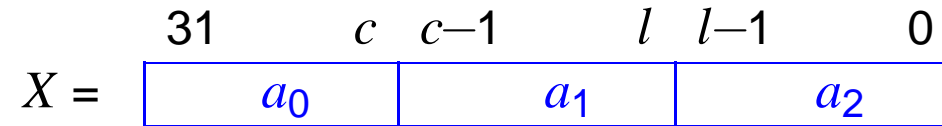
- idea: data that is “*currently most needed*” is brought into a (smaller) faster memory
- observation: memory accesses in *most* programs exhibit:
 - temporal locality: if access address X , likely to access X again soon
 - spatial locality: if access address X , likely to access $X+1$ soon
 - ⇒ cache 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 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



- ✓ 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
 - implemented as a set of K direct-mapped caches of size C'
 addresses with same a_1 can map into the corresp. line in *any* of the K sets
 - typically $K = 1, 2, 4, 5, 6, 8$ (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
 - 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