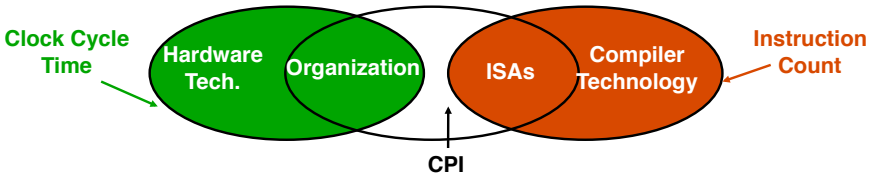
An important idea...

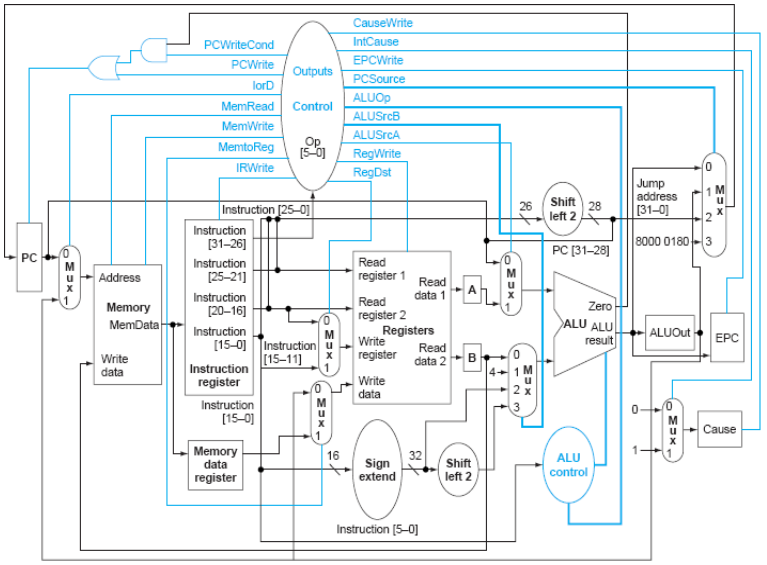
A common denominator

$$\frac{Instructions}{\text{Pr}ogram} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} = \frac{Seconds}{\text{Pr}ogram} = CPU\ time$$

- We can see CPU performance dependent on:
  - Clock rate, CPI, and instruction count
- CPU time is directly proportional to all 3:
  - Therefore an x % improvement in any one variable leads to an x % improvement in CPU performance
- But, everything usually affects everything:

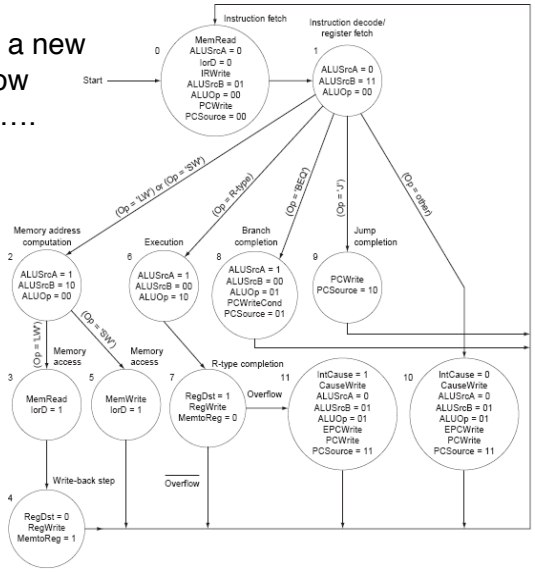


Understand flow through datapath

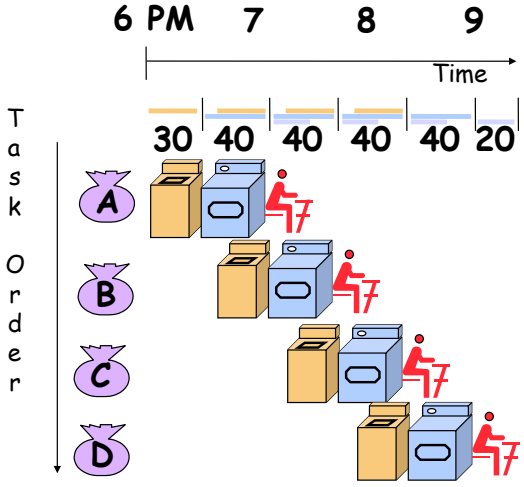


Understand how FSM translates to CCs

Or, if you add a new instruction, how FSM affected….



Pipelining Lessons (laundry example)



- Multiple tasks operating simultaneously
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Also, need time to "fill" and "drain" the pipeline.

More technical detail

- Book's approach to draw pipeline timing diagrams…
  - Time runs left-to-right, in units of stage time
  - Each "row" below corresponds to distinct initiation
  - Boundary b/t 2 column entries: pipeline register
    - (i.e. hamper)
  - Look at columns to see what stage is doing what

0	1	2	3	4	5	6
Wash 1	Dry 1	Fold 1	Pack 1			
	Wash 2	Dry 2	Fold 2	Pack 2		
		Wash 3	Dry 3	Fold 3	Pack 3	
			Wash 4	Dry 4	Fold 4	Pack 4
				Wash 5	Dry 5	Fold 5
					Wash 6	Dry 6

Time for N initiations to complete: NT + (S-1)T
Throughput: Time per initiation = T + (S-1)T/N → T!

The hazards of pipelining

- Pipeline hazards prevent next instruction from executing during designated clock cycle
- There are 3 classes of hazards:
  - Structural Hazards:
    - Arise from resource conflicts
    - HW cannot support all possible combinations of instructions
  - Data Hazards:
    - Occur when given instruction depends on data from an instruction ahead of it in pipeline
  - Control Hazards:
    - Result from branch, other instructions that change flow of program (i.e. change PC)

Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…
- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction
- Ignoring overhead and assuming stages are balanced:

$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ stall\ cycles\ per\ instruction}$$

Data hazards

- These exist because of pipelining
- Why do they exist???
  - Pipelining changes order or read/write accesses to operands
  - Order differs from order seen by sequentially executing instructions on unpipelined machine
- Consider this example:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

All instructions after ADD use result of ADD

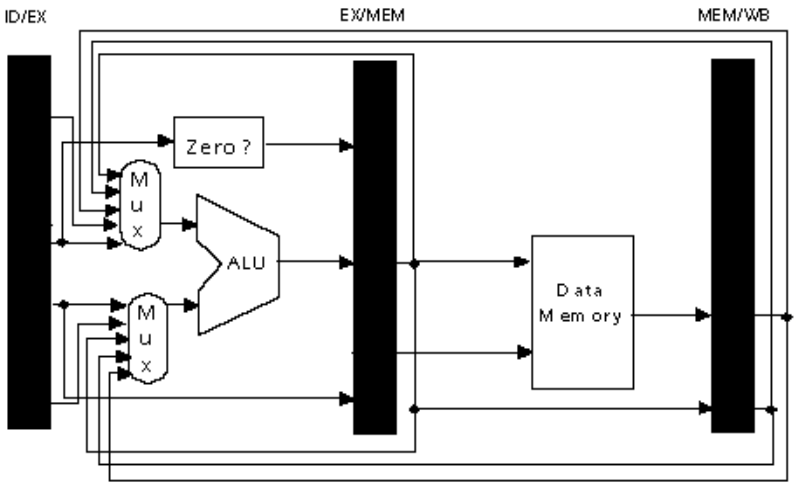
ADD writes the register in WB but SUB needs it in ID.

This is a data hazard

Forwarding

- Problem illustrated on previous slide can actually be solved relatively easily – with forwarding
- In this example, result of the ADD instruction not really needed until after ADD actually produces it
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes! Hence this slide!
- Generally speaking:
  - Forwarding occurs when a result is passed directly to functional unit that requires it.
  - Result goes from output of one unit to input of another

HW Change for Forwarding



Hazards vs. Dependencies

- dependence: fixed property of instruction stream
  - (i.e., program)
- hazard: property of program and processor organization
  - implies potential for executing things in wrong order
    - potential only exists if instructions can be simultaneously "in-flight"
    - property of dynamic distance between instructions vs. pipeline depth
- For example, can have RAW dependence with or without hazard
  - depends on pipeline

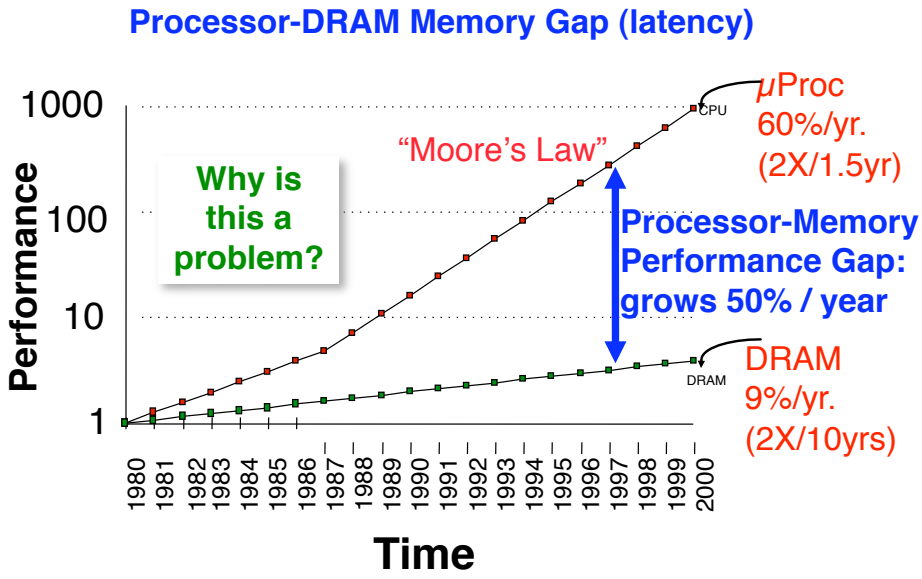
Branch/Control Hazards

- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers
- Also need to consider hazards involving branches:
  - Example:
    - 40: beq $1, $3, $28 # ($28 gives address 72)
    - 44: and $12, $2, $5
    - 48: or $13, $6, $2
    - 52: add $14, $2, $2
    - 72: lw $4, 50($7)
- How long will it take before the branch decision takes effect?
  - What happens in the meantime?

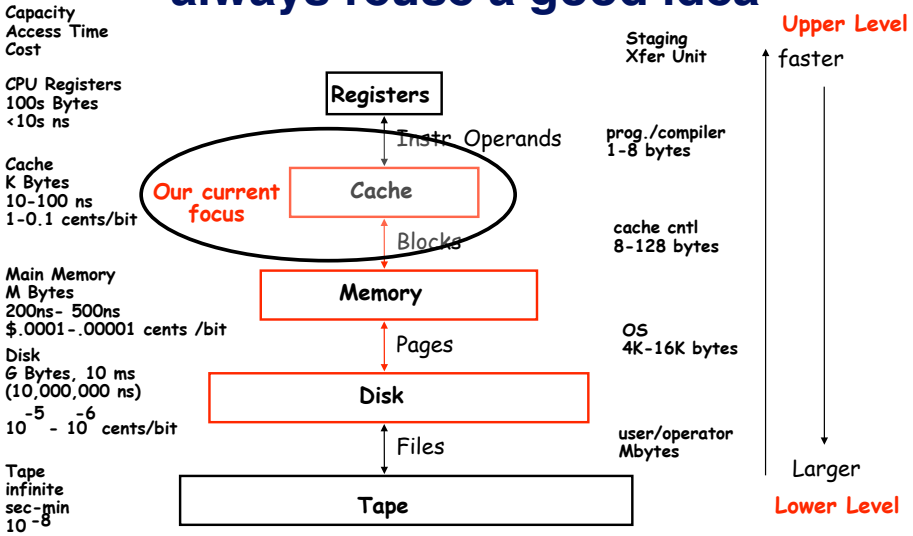
Branch Prediction

- Prior solutions are "ugly"
- Better (& more common): guess in IF stage
  - Technique is called "branch predicting"; needs 2 parts:
    - "Predictor" to guess where/if instruction will branch (and to where)
    - "Recovery Mechanism": i.e. a way to fix your mistake
  - Prior strategy:
    - Predictor: always guess branch never taken
    - Recovery: flush instructions if branch taken
  - Alternative: accumulate info. in IF stage as to…
    - Whether or not for any particular PC value a branch was taken next
    - To where it is taken
    - How to update with information from later stages

Is there a problem with DRAM?

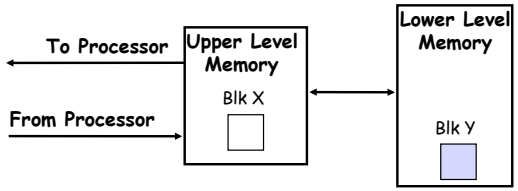


The Full Memory Hierarchy "always reuse a good idea"



Terminology Summary

- Hit: data appears in block in upper level (i.e. block X in cache)
  - Hit Rate: fraction of memory access found in upper level
  - Hit Time: time to access upper level which consists of
    - RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (i.e. block Y in memory)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Extra time to replace a block in the upper level +
    - Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264)



Average Memory Access Time

$$AMAT = HitTime + (1 - h) \times MissPenalty$$

- Hit time: basic time of every access.
- Hit rate (h): fraction of access that hit
- Miss penalty: extra time to fetch a block from lower level, including time to replace in CPU

Where can a block be placed in a $?

- 3 schemes for block placement in a cache:
  - Direct mapped cache:
    - Block (or data to be stored) can go to only 1 place in cache
    - Usually: (Block address) MOD (# of blocks in the cache)
  - Fully associative cache:
    - Block can be placed anywhere in cache
  - Set associative cache:
    - "Set" = a group of blocks in the cache
    - Block mapped onto a set & then block can be placed anywhere within that set
    - Usually: (Block address) MOD (# of sets in the cache)
    - If n blocks, we call it n-way set associative

How is a block found in the cache?

- Block offset field selects data from block
  - (i.e. address of desired data within block)
- Index field selects a specific set
- Tag field is compared against it for a hit

- Could we compare on more of address than the tag?
  - Not necessary; checking index is redundant
    - Used to select set to be checked
    - Ex.: Address stored in set 0 must have 0 in index field
  - Offset not necessary in comparison –entire block is present or not and all block offsets must match

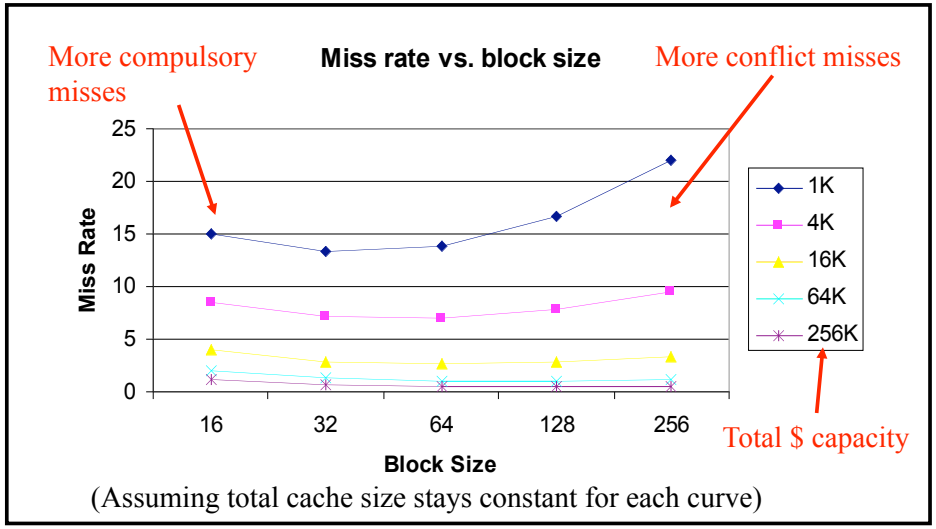
Reducing cache misses

- Obviously, we want data accesses to result in cache hits, not misses –this will optimize performance

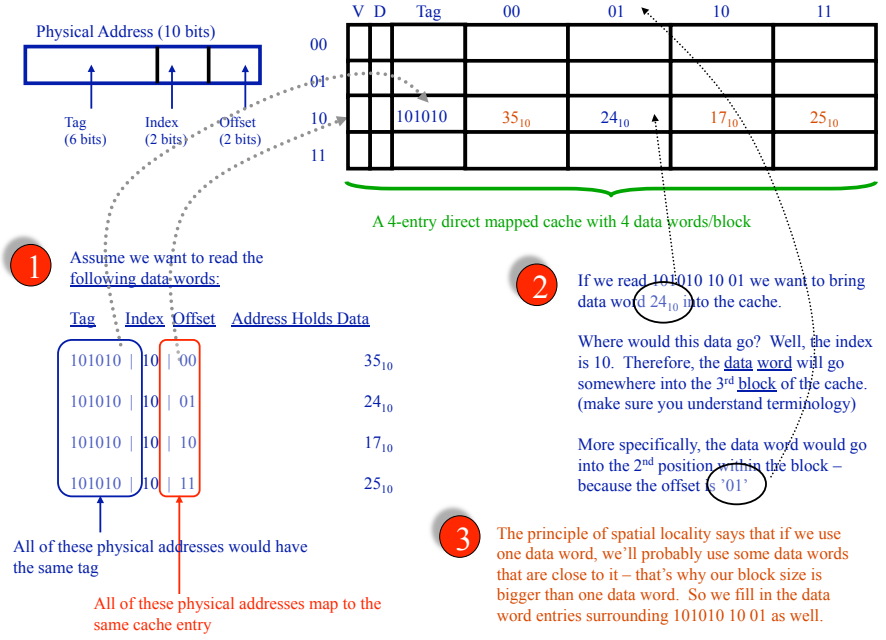
- Start by looking at ways to increase % of hits….

- …but first look at 3 kinds of misses!
  - Compulsory misses:
    - Very 1st access to cache block will not be a hit –the data's not there yet!
  - Capacity misses:
    - Cache is only so big. Won't be able to store every block accessed in a program – must swap out!
  - Conflict misses:
    - Result from set-associative or direct mapped caches
    - Blocks discarded/retrieved if too many map to a location

Optimizing cache design



(Assuming total cache size stays constant for each curve)



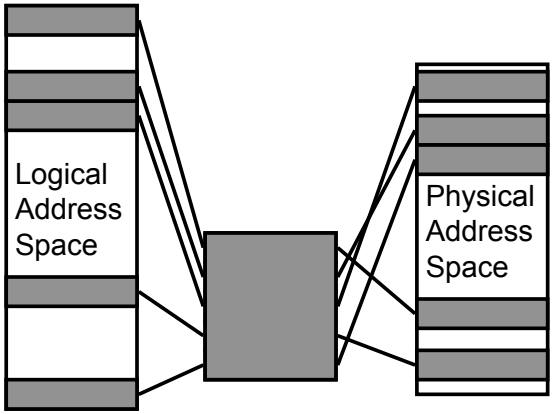
Second-level caches

- This will of course introduce a new definition for average memory access time:
  - Hit timeL1 + Miss RateL1 * Miss PenaltyL1
  - Where, Miss PenaltyL1 =
    - Hit TimeL2 + Miss RateL2 * Miss PenaltyL2
    - So 2nd level miss rate measure from 1st level cache misses…
- A few definitions to avoid confusion:
  - Local miss rate:
    - # of misses in the cache divided by total # of memory accesses to the cache – specifically Miss RateL2
  - Global miss rate:
    - # of misses in the cache divided by total # of memory accesses generated by the CPU – specifically -- Miss RateL1 * Miss RateL2

Virtual Memory

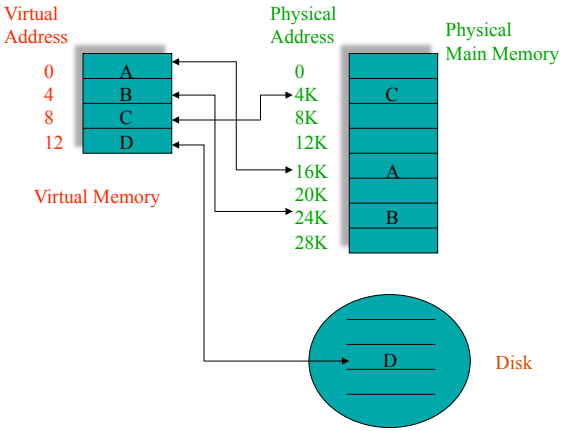
- Some facts of computer life…
  - Computers run lots of processes simultaneously
  - No full address space of memory for each process
    - Physical memory expensive and not dense - thus, too small
  - Must share smaller amounts of physical memory among many processes
- Virtual memory is the answer!
  - Divides physical memory into blocks, assigns them to different processes
    - Compiler assigns data to a "virtual" address.
      - VA translated to a real/physical somewhere in memory
    - Allows program to run anywhere; where is determined by a particular machine, OS
      - + Business: common SW on wide product line (w/o VM, sensitive to actual physical memory size)

Virtual address space greater than Logical address space



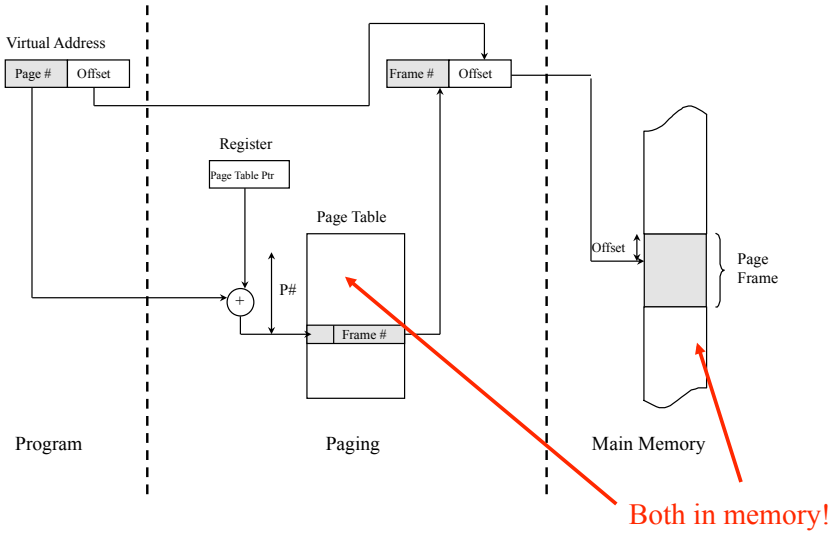
The gist of virtual memory

- **Relieves problem of making a program that was too large to fit in physical memory – well…fit!**
- **Allows program to run in any location in physical memory**
  - **Really useful as you might want to run same program on lots machines…**

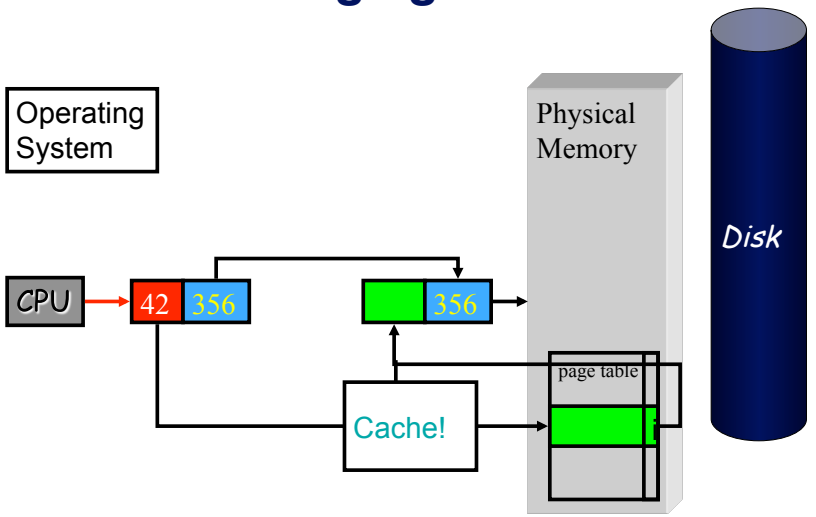


Logical program is in contiguous VA space; here, pages: A, B, C, D;
(3 are in main memory and 1 is located on the disk)

Review: Address Translation



Paging/VM

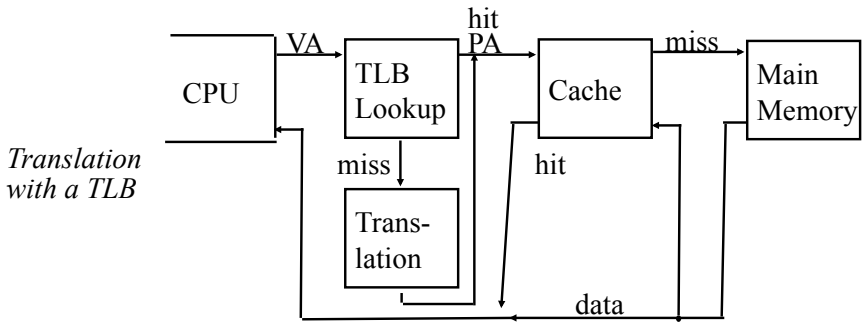


Special-purpose cache for translations
Historically called the TLB: Translation Lookaside Buffer

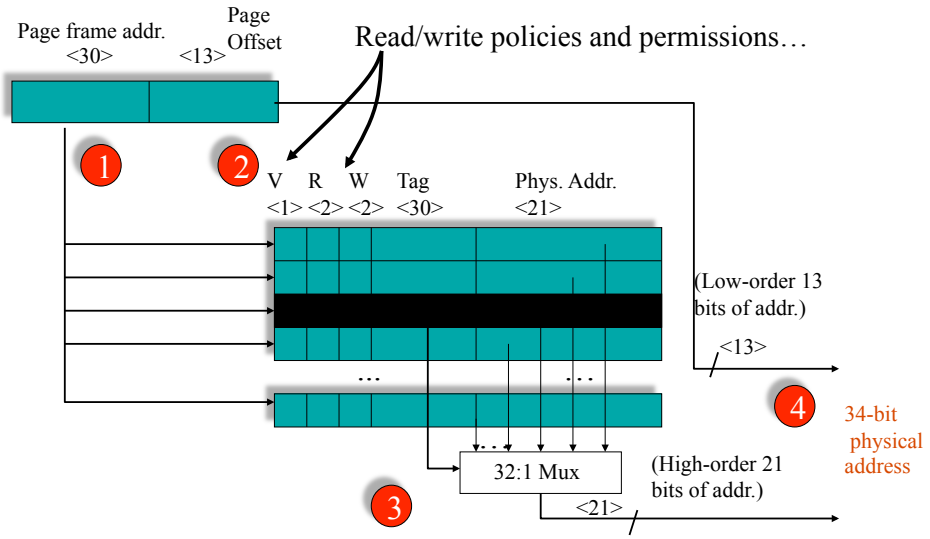
Review: Translation Cache

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



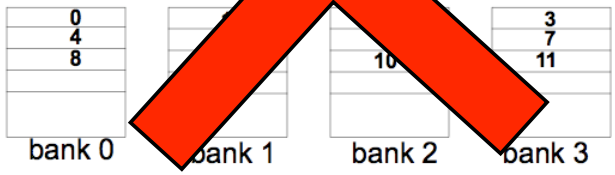
An example of a TLB



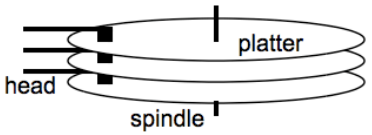
Bandwidth: Simple Interleaving/Banking

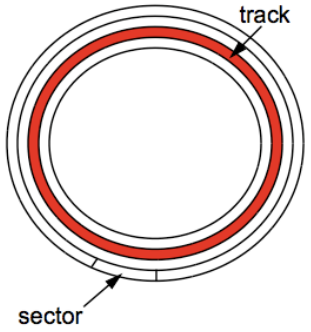
use multiple DRAMs, exploit their aggregate bandwidth

- each DRAM called a bank
  - not true: collection of DRAM called a bank
- M 32-bit banks
- simple interleaving: address lines
- word A in bank (A % M)
  - e.g., M=4, A=9: bank 1



Disk Parameters





- 1–20 *platters* (data on both sides)
  - magnetic iron-oxide coating
  - 1 read/write head per side
- 500–2500 *tracks* per platter
- 32–128 *sectors* per track
  - sometimes fewer on inside tracks
- 512–2048 *bytes* per sector
  - usually fixed number of bytes/sector
  - data + ECC (parity) + gap
- 4–24GB total
- 3000–10000 RPM

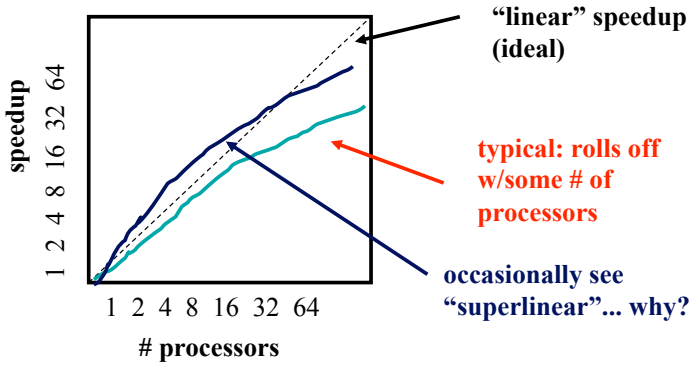
Disk Performance Example

- parameters
  - 3600 RPM ⇒ 60 RPS (may help to think in units of tracks/sec)
  - avg seek time: 9ms
  - 100 sectors per track, 512 bytes per sector
  - controller + queuing delays: 1ms
- Q: average time to read 1 sector (512 bytes)?
  - $rate_{transfer}$ = 100 sectors/track * 512 B/sector * 60 RPS = 2.4 MB/s
  - $t_{transfer}$ = 512 B / 2.4 MB/s = 0.2ms
  - $t_{rotation}$ = .5 / 60 RPS = 8.3ms
  - t_{disk} = 9ms (seek) + 8.3ms (rotation) + 0.2ms (xfer) + 1ms = 18.5ms
  - $t_{transfer}$ is only a small component! counter-intuitive?
  - end of story? no! $t_{queuing}$ not fixed (gets longer with more requests)

Speedup

metric for performance on latency-sensitive applications

- **Time(1) / Time(P) for P processors**
  - **note: must use the best *sequential* algorithm for Time(1) -- the parallel algorithm may be different.**



More examples with Amdahl's Law

- **Sequential part can limit speedup**
- **Example: 100 processors, 90× speedup?**
  - **$T_{new} = T_{parallelizable}/100 + T_{sequential}$**
  - $$\text{Speedup} = \frac{1}{(1-F_{parallelizable}) + F_{parallelizable}/100} = 90$$
  - **Solving: $F_{parallelizable} = 0.999$**
- **Need sequential part to be 0.1% of original time**

Scaling Example

- **Workload: sum of 10 scalars, and 10 × 10 matrix sum**
  - **Speed up from 10 to 100 processors**
- **Single processor: Time = (10 + 100) × t_{add}**
- **10 processors**
  - **Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}**
  - **Speedup = 110/20 = 5.5 (55% of potential)**
- **100 processors**
  - **Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}**
  - **Speedup = 110/11 = 10 (10% of potential)**
- **Assumes load can be balanced across processors**