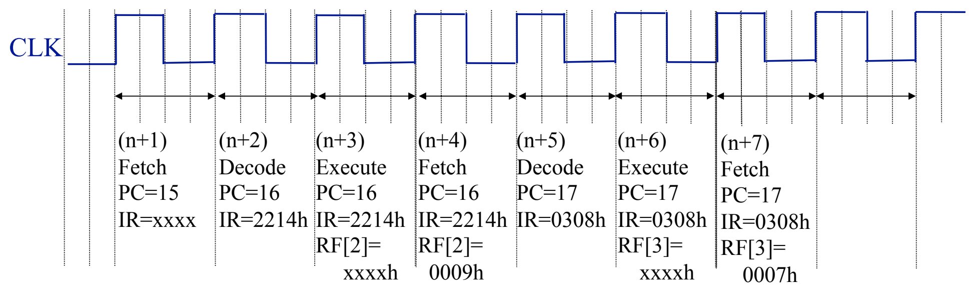
Lecture 28-29 Course Review

Clock cycles: Understand in multi-cycle, pipeline

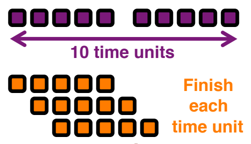
Q1: D[8] = D[8] + RF[1] + RF[4]

…	
I[15]: Add R2, R1, R4	RF[1] = 4
I[16]: MOV R3, 8	RF[4] = 5
I[17]: Add R2, R2, R3	D[8] = 7
…	


CLK
(n+1) Fetch PC=15 IR=xxxx
(n+2) Decode PC=16 IR=2214h
(n+3) Execute PC=16 IR=2214h RF[2]= xxxxh
(n+4) Fetch PC=16 IR=2214h RF[2]= 0009h
(n+5) Decode PC=17 IR=0308h
(n+6) Execute PC=17 IR=0308h RF[3]= xxxxh
(n+7) Fetch PC=17 IR=0308h RF[3]= 0007h

Common (and good) performance metrics

- latency: response time, execution time
  - good metric for fixed amount of work (minimize time)
- throughput: bandwidth, work per time, "performance"
  - = (1 / latency) when there is NO OVERLAP
  - > (1 / latency) when there is overlap
    - in real processors there is always overlap
  - good metric for fixed amount of time (maximize work)

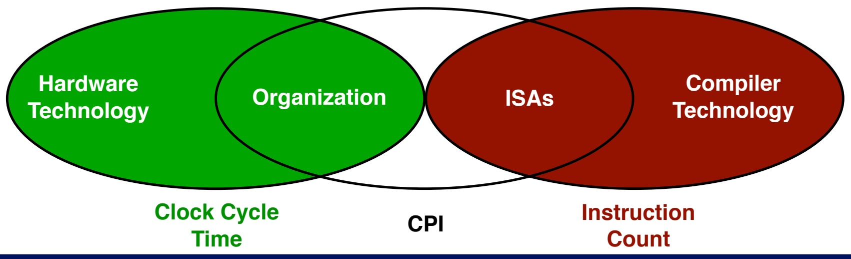


- comparing performance
  - A is N times faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = N
  - A is X% faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = 1 + X/100

CPU time (the "best" metric)

$$\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} = \frac{Seconds}{Program} = CPU\ time$$

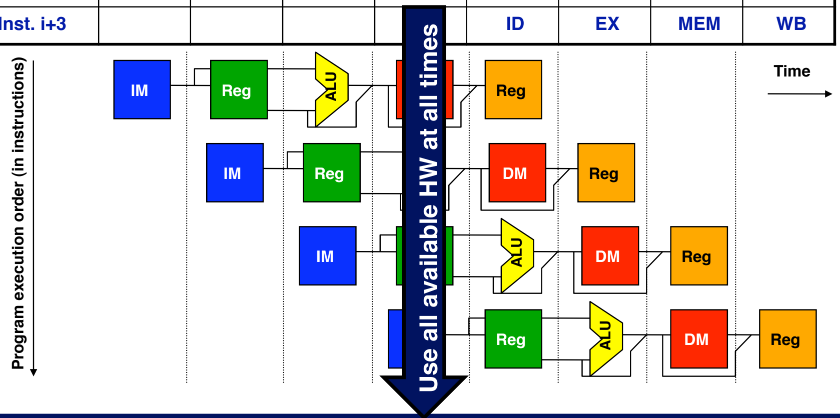
- We can see CPU performance dependent on:
  - Clock rate, CPI, and instruction count
- CPU time is directly proportional to all 3:
  - Therefore an x % improvement in any one variable leads to an x % improvement in CPU performance
- But, everything usually affects everything:



Recap: Pipelining improves throughput

Clock Number

Inst. #	1	2	3	4	5	6	7	8
Inst. i	IF	ID	EX	MEM	WB			
Inst. i+1		IF	ID	EX	MEM	WB		
Inst. i+2			IF	ID	EX	MEM	WB	
Inst. i+3				IF	ID	EX	MEM	WB


Program execution order (in instructions)
Time
IM
Reg
ALU
DM
Reg
Use all available HW at all times

Recap: pipeline math

- If times for all S stages are equal to T:
  - Time for one initiation to complete still ST
  - Time between 2 initiates = T not ST
  - Initiations per second = 1/T

Key to improving performance: "parallel" execution

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

- Pipelining: Overlap multiple executions of same sequence
  - Improves THROUGHPUT, not the time to perform a single operation

Recap: 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:

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

- If no stalls, speedup equal to # of pipeline stages in ideal case

Stalls occur because of hazards!

Recap: Structural hazards

- 1 way to avoid structural hazards is to duplicate resources
  - i.e.: An ALU to perform an arithmetic operation and an adder to increment PC
- If not all possible combinations of instructions can be executed, structural hazards occur
- Most common instances of structural hazards:
  - When some resource not duplicated enough
- Pipeline stalls result of hazards, CPI increased from the usual "1"

Recap: 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 un-pipelined 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

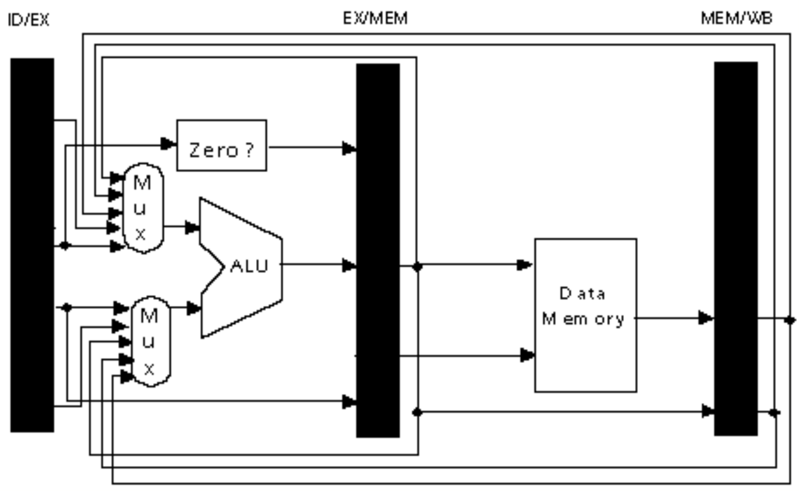
ADD writes the register in WB but SUB needs it in ID.

This is a data hazard

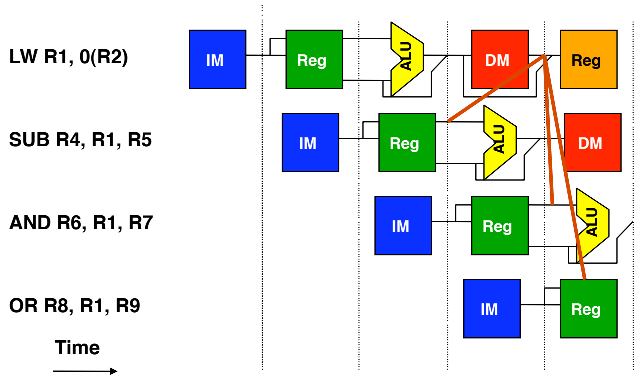
Recap: Forwarding & data hazards

- 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

Recap: HW change for forwarding



Recap: Forwarding doesn't always work



Load has a latency that forwarding can't solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can't get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

Recap: 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

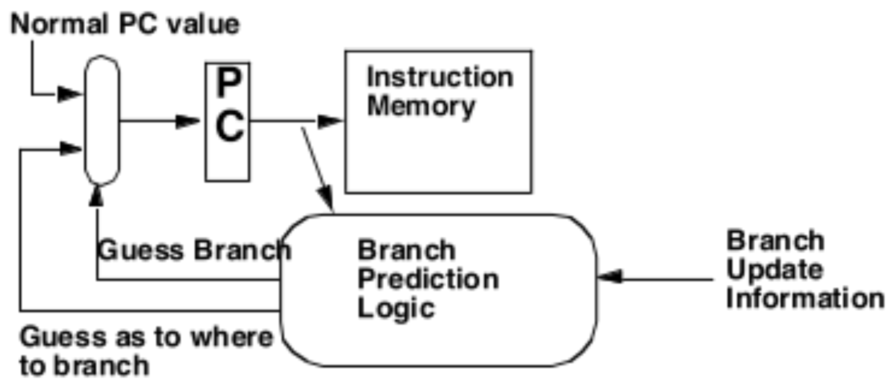
Branch/Control Hazards

- So far, we've limited discussion of hazards to:
  - Arithmetic/logic operations
  - Data transfers
- Also need to consider hazards involving branches:
  - Example:

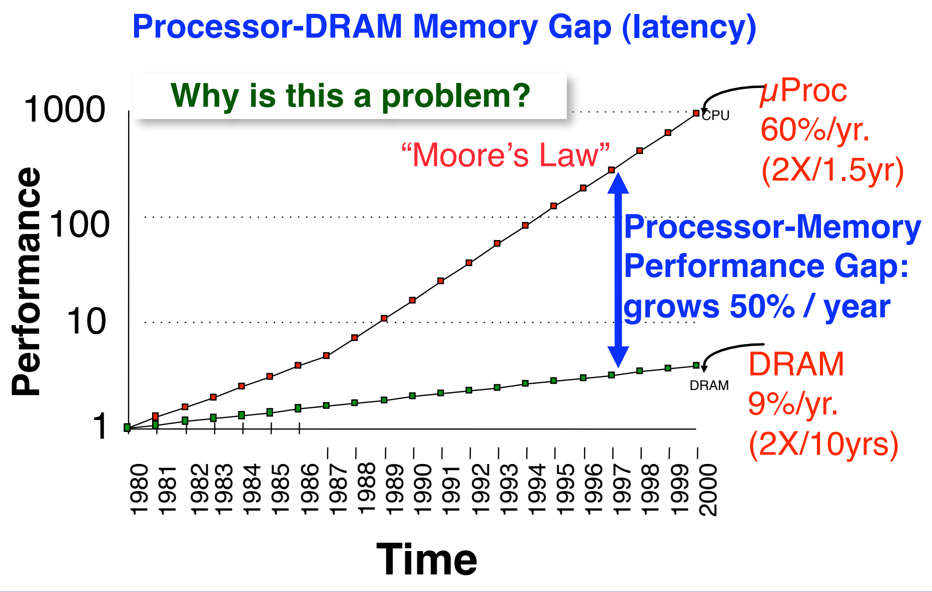
```
40: beq    $1, $3, $28        # ($28 gives address 72)
44: and    $12, $2, $5
48: or     $13, $6, $2
52: add    $14, $2, $2
72: lw     $4, 50($7)
```

- How long will it take before the branch decision takes effect?
  - What happens in the meantime?

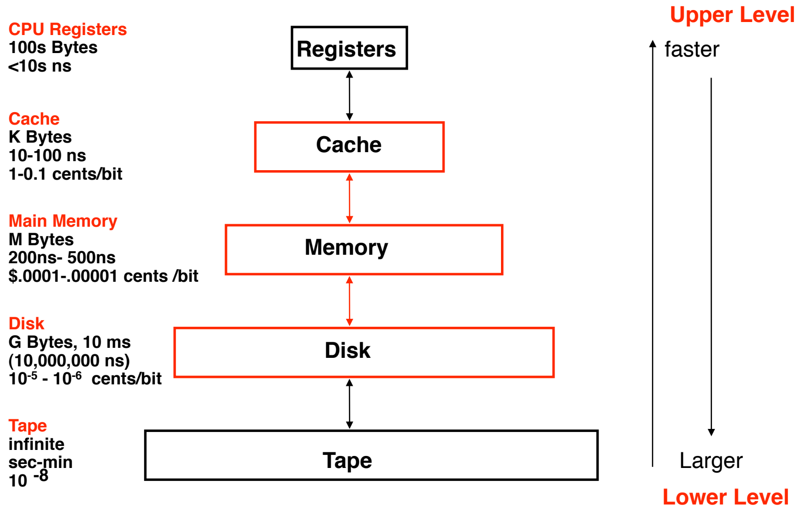
A Branch Predictor



Is there a problem with DRAM?



Common memory hierarchy:

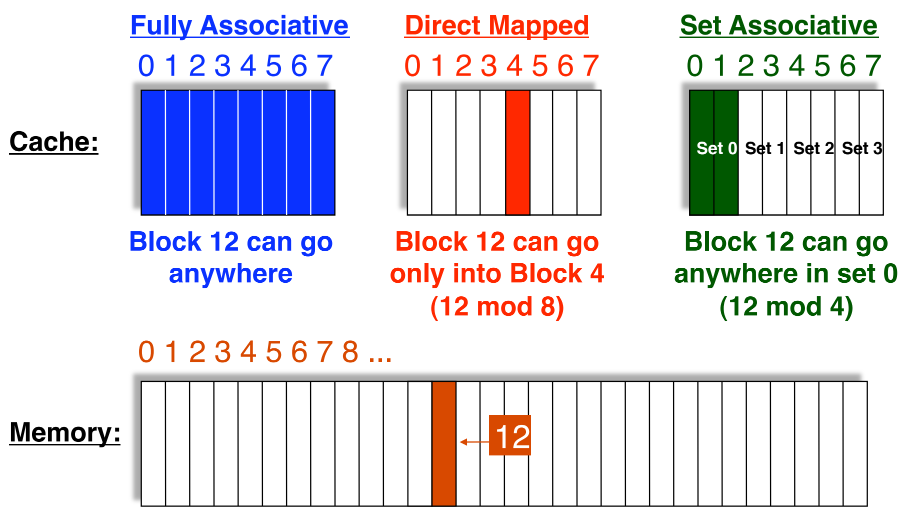


Average Memory Access Time

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

- 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 cache?



How is a block found in the cache?

Block Address		Block Offset
Tag	Index	

- 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

Which block should be replaced on a cache miss?

- If we look something up in cache and entry not there, generally want to get data from memory and put it in cache
  - B/c principle of locality says we'll probably use it again
- Direct mapped caches have 1 choice of what block to replace
- Fully associative or set associative offer more choices
- Usually 2 strategies:
  - Random – pick any possible block and replace it
  - LRU – stands for "Least Recently Used"
    - Why not throw out the block not used for the longest time
    - Usually approximated, not much better than random – i.e. 5.18% vs. 5.69% for 16KB 2-way set associative

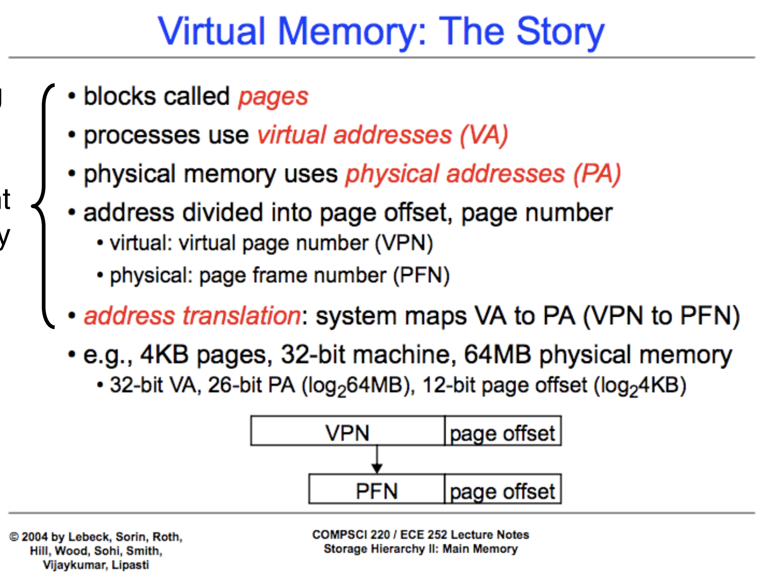
2nd-level caches

- Processor/memory performance gap makes us consider:
  - If we should make caches faster to keep pace with CPUs
  - If we should make caches larger to overcome widening gap between CPU and main memory
- One solution is to do both:
  - Add another level of cache (L2) between the 1st level cache (L1) and main memory
    - Ideally L1 will be fast enough to match the speed of the CPU while L2 will be large enough to reduce the penalty of going to main memory
- This will of course introduce a new definition for average memory access time:
  - Hit time$_{L1}$ + Miss Rate$_{L1}$ * Miss Penalty$_{L1}$
  - Where, Miss Penalty$_{L1}$ =
    - Hit Time$_{L2}$ + Miss Rate$_{L2}$ * Miss Penalty$_{L2}$
    - So 2nd level miss rate measure from 1st level cache misses…

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

Translating VA to PA sort of like finding right cache entry with division of PA

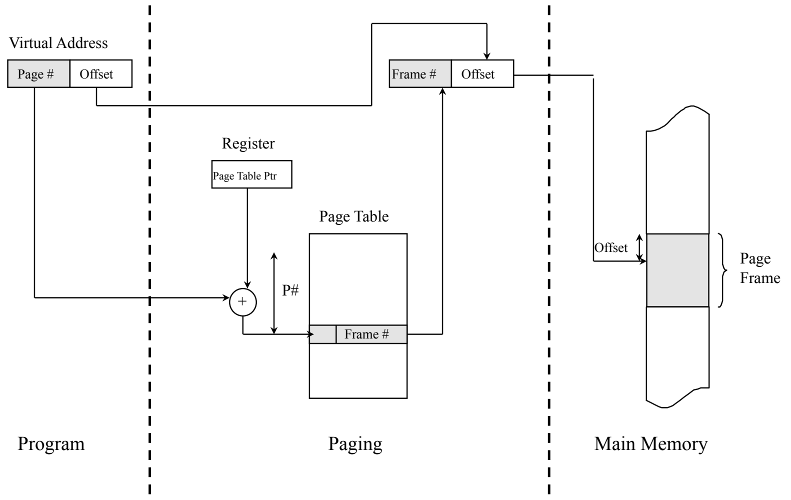


Virtual Memory: The Story

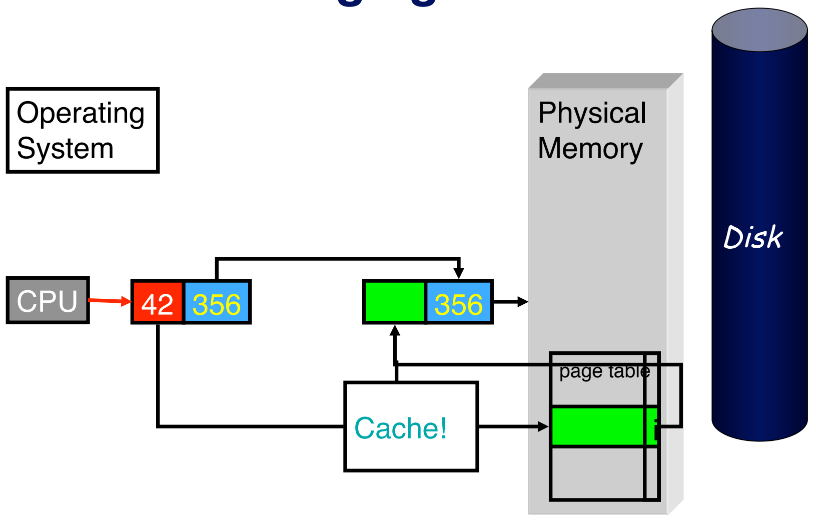
- blocks called *pages*
- processes use *virtual addresses (VA)*
- physical memory uses *physical addresses (PA)*
- address divided into page offset, page number
  - virtual: virtual page number (VPN)
  - physical: page frame number (PFN)
- *address translation*: system maps VA to PA (VPN to PFN)
- e.g., 4KB pages, 32-bit machine, 64MB physical memory
  - 32-bit VA, 26-bit PA ($\log_2$64MB), 12-bit page offset ($\log_2$4KB)

COMPSCI 220 / ECE 252 Lecture Notes
Storage Hierarchy II: Main Memory

Review: Address Translation

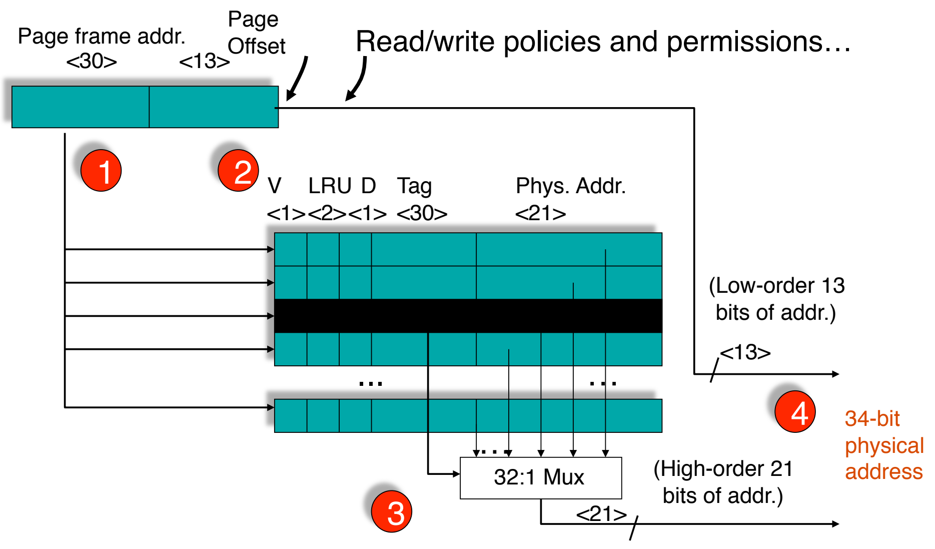


Paging/VM

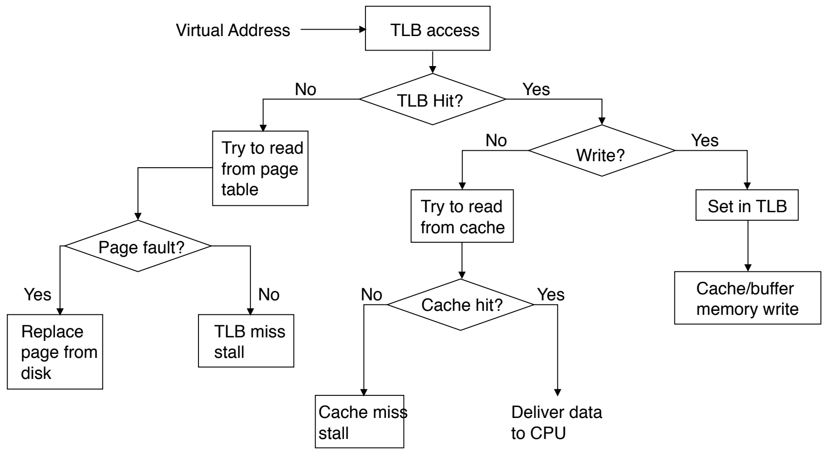


Special-purpose cache for translations
Historically called the TLB: Translation Lookaside Buffer

An example of a TLB

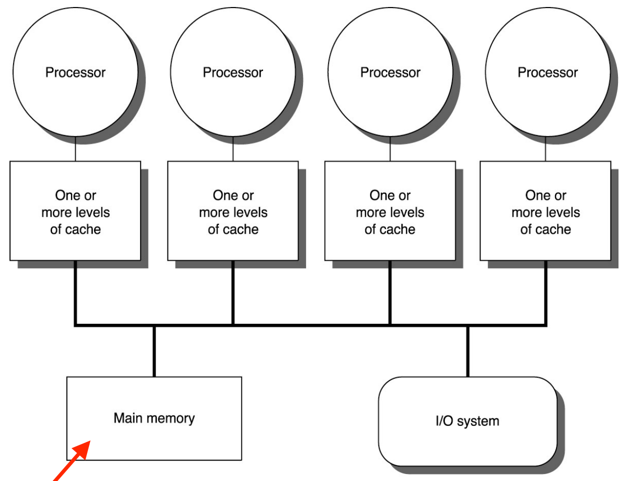


The "big picture" and TLBs



MIMD Multiprocessors

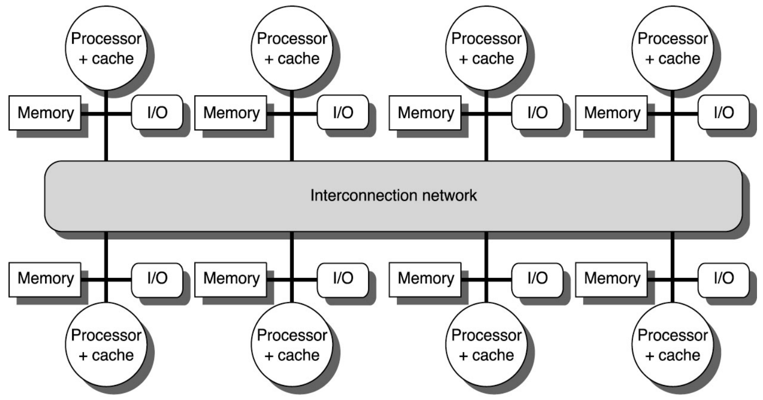
Centralized Shared Memory


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

Note: just 1 memory

MIMD Multiprocessors

Distributed Memory

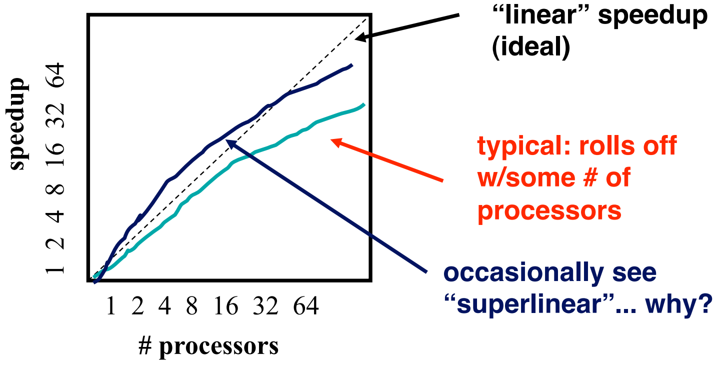


Multiple, distributed memories here.

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.



Impediments to Parallel Performance

- Reliability:
  - Want to achieve high "up time" – especially in non-CMPs
- Contention for access to shared resources
  - i.e. multiple accesses to limited # of memory banks may dominate system scalability
- Programming languages, environments, & methods:
  - Need simple semantics that can expose computational properties to be exploited by large-scale architectures
- Algorithms

Not all problems are parallelizable

$$\text{Speedup} = \frac{1}{\left[1 - \text{Fraction}_{\text{parallelizable}}\right] + \frac{\text{Fraction}_{\text{parallelizable}}}{N}}$$

What if you write good code for 4-core chip and then get an 8-core chip?

- Cache coherency
  - P1 writes, P2 can read
    - Protocols can enable $ coherency but add overhead

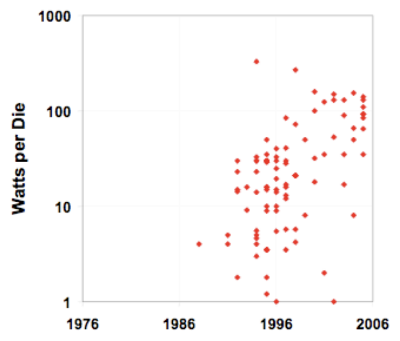
Challenges: Latency

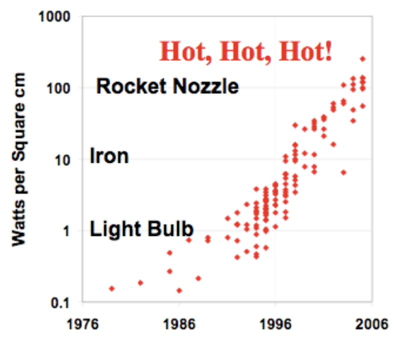
- …is already a major source of performance degradation
  - Architecture charged with hiding local latency
    - (that's why we talked about registers & caches)
  - Hiding global latency is also task of programmer
    - (I.e. manual resource allocation)
- Today:
  - access to DRAM in 100s of CCs
  - round trip remote access in 1000s of CCs
  - multiple clock cycles to cross chip or to communicate from core-to-core
    - Not "free" as we assumed in send-receive example from L27

Recap: L24 – Parallel Processing on MC

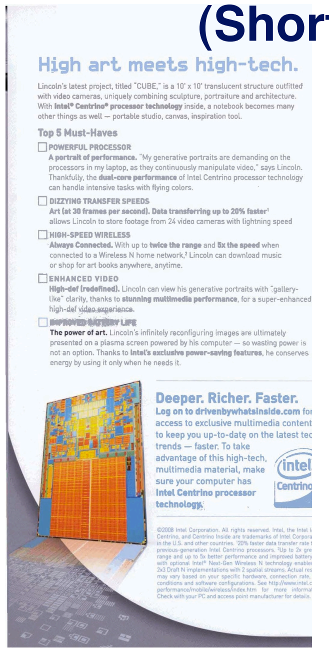
- Simple quantitative examples on how (i) reliability, (ii) communication overhead, and (iii) load balancing impact performance of parallel systems
- Technology drive to multi-core computing

Why the clock flattening? POWER!!!!





(Short term?) Solution



High art meets high-tech.

Lincoln's latest project, titled "CUBE," is a 10' x 10' translucent structure outfitted with video cameras, uniquely combining sculpture, portraiture and architecture. With **Intel® Centrino® processor technology** inside, a notebook becomes many other things as well — portable studio, canvas, inspiration tool.

Top 5 Must-Haves

☐ POWERFUL PROCESSOR
A portrait of performance. "My generative portraits are demanding on the processors in my laptop, as they continuously manipulate video," says Lincoln. Thankfully, the **dual-core performance** of Intel Centrino processor technology can handle intensive tasks with flying colors.

☐ DIZZYING TRANSFER SPEEDS
Art (at 30 frames per second). Data transferring up to 20% faster[1] allows Lincoln to store footage from 24 video cameras with lightning speed

☐ HIGH-SPEED WIRELESS
Always Connected. With up to **twice the range** and **5x the speed** when connected to a Wireless N home network,[2] Lincoln can download music or shop for art books anywhere, anytime.

☐ ENHANCED VIDEO
High-def (redefined). Lincoln can view his generative portraits with "gallery-like" clarity, thanks to **stunning multimedia performance**, for a super-enhanced high-def video experience.

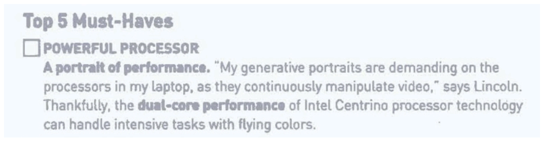
☐ IMPROVED BATTERY LIFE
The power of art. Lincoln's infinitely reconfiguring images are ultimately presented on a plasma screen powered by his computer — so wasting power is not an option. Thanks to **Intel's exclusive power-saving features**, he conserves energy by using it only when he needs it.

Deeper. Richer. Faster.
Log on to drivenbywhatsinside.com for access to exclusive multimedia content to keep you up-to-date on the latest tec trends — faster. To take advantage of this high-tech, multimedia material, make sure your computer has **Intel Centrino processor technology.**

©2008 Intel Corporation. All rights reserved. Intel, the Intel l Centrino, and Centrino Inside are trademarks of Intel Corpora in the U.S. and other countries. [1]20% faster data transfer rate t previous-generation Intel Centrino processors. [2]Up to 2x gre range and up to 5x better performance and improved battery with optional Intel® Next-Gen Wireless N technology enable 2x3 Draft N implementations with 2 spatial streams. Actual res may vary based on your specific hardware, connection rate, conditions and software configurations. See http://www.intel.c performance/mobile/wireless/index.htm for more informat Check with your PC and access point manufacturer for details.

- Processor complexity is good enough
- Transistor sizes can still scale
- Slow processors down to manage power
- Get performance from...

Parallelism



Top 5 Must-Haves

☐ POWERFUL PROCESSOR
A portrait of performance. "My generative portraits are demanding on the processors in my laptop, as they continuously manipulate video," says Lincoln. Thankfully, the **dual-core performance** of Intel Centrino processor technology can handle intensive tasks with flying colors.

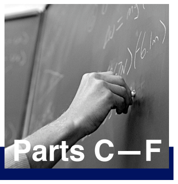
(i.e. 1 processor, 1 ns clock cycle
vs.
2 processors, 2 ns clock cycle)

Processes vs. Threads

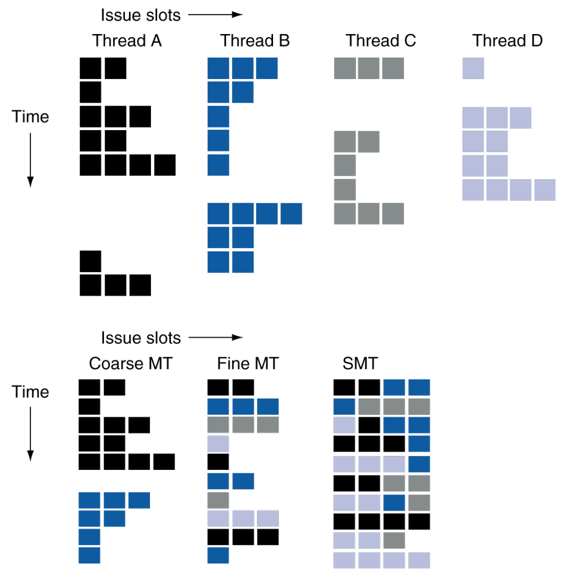
- Process
  - Created by OS
  - Much "overhead"
    - Process ID
    - Process group ID
    - User ID
    - Working directory
    - Program instructions
    - Registers
    - Stack space
    - Heap
    - File descriptors
    - Shared libraries
    - Shared memory
    - Semaphores, pipes, etc.
- Thread
  - Can exist within process
  - Shares process resources
  - Duplicate bare essentials to execute code on chip
    - Program counter
    - Stack pointer
    - Registers
    - Scheduling priority
    - Set of pending, blocked signals
    - Thread specific data

Multi-threading

- Idea:
  - Performing multiple threads of execution in parallel
    - Replicate registers, PC, etc.
  - Fast switching between threads
- Flavors:
  - Fine-grain multithreading
    - Switch threads after each cycle
    - Interleave instruction execution
    - If one thread stalls, others are executed
  - Coarse-grain multithreading
    - Only switch on long stall (e.g., L2-cache miss)
    - Simplifies hardware, but doesn't hide short stalls
      - (e.g., data hazards)
  - SMT (Simultaneous Multi-Threading)
    - Especially relevant for superscalar



Coarse MT vs. Fine MT vs. SMT



Mixed Models:

- Threaded systems and multi-threaded programs are not specific to multi-core chips.
  - In other words, could imagine a multi-threaded uni-processor too…
- However, could have an N-core chip where:
  - … N threads of a single process are run on N cores
  - … N processes run on N cores – and each core splits time between M threads

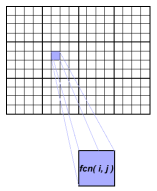
GPUs

- GPU = Graphics Processing Unit
  - Efficient at manipulating computer graphics
  - Graphics accelerator uses custom HW that makes mathematical operations for graphics operations fast/efficient
    - Why SIMD? Do same thing to each pixel
- Often part of a *heterogeneous* system
  - GPUs don't do all things CPU does
  - Good at some specific things
    - i.e. matrix-vector operations
- GPU HW:
  - No multi-level caches
  - Hide memory latency with threads
    - To process all pixel data
  - GPU main memory oriented toward bandwidth

Parallel Programming

- To develop parallel software, must first understand if serial code can be parallelized…
  - Consider 2 examples…
- (A) Calculations on 2D array elements
  - Computation on each array element independent from others
    - Serial code:

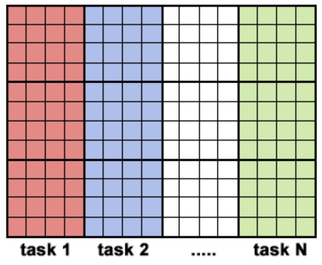
```
for j =1:N
   for i=1:N
      a(i,j) = f(i,j)
```



    - If calculation of elements is independent from one another, problem is "embarrassingly parallel"
      - (usually computationally intensive)

Understand the Problem and Program

- Can distribute elements so each processor owns its own array (or subarray)
  - This type of problem can lead to "superlinear" speedup
    - (Entire dataset may now fit in cache)
- Parallel code might look like…



```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
  end do

  send MASTER results

endif
```

Understand the Problem and Program

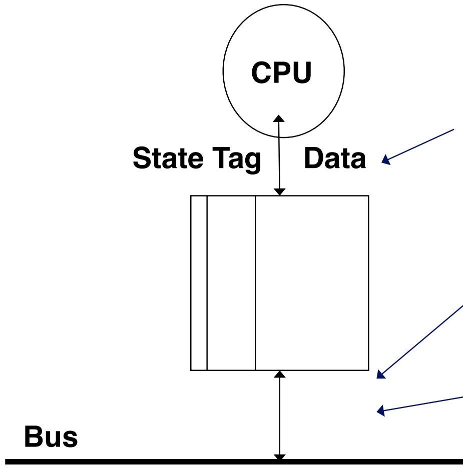
- (B) Calculate the numbers in a Fibonacci sequecne
  - (Hint: Part of a project benchmark…)
  - Fibonacci number defined by:
    - F(n) = F(n-1) + F(n-2)
      - Fibonacci series (1,1,2,3,5,8,13,21,…)
  - This problem is non-parallelizable!
    - Calculation of F(n) *dependent* on other calculations
    - F(n-1) and F(n-2) cannot be calculated *independently*

Others issues programmer must consider

- Partitioning
  - Domain Decomposition
  - Functional Decomposition
- Communication Overhead
- Synchronization Overhead
- Load Balancing

- Remember: Could write very nicely parallelized code but expected performance may not be delivered if you don't account for realistic implementation overheads

Coherence overhead: How Snooping Works



CPU references check cache tags (as usual)

Cache misses filled from memory (as usual)

+

Other read/write on bus must check tags, too, and possibly invalidate

Often 2 sets of tags…why?

Will need to update/invalidate $ed data

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				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 a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

- Assumes neither cache had value/location X in it 1st
- When 2nd miss by B occurs, CPU A responds with value canceling response from memory.
- Update B's cache & memory contents of X updated
- Typical and simple…

M(E)SI Snoopy Protocols for $ coherency

- State of block B in cache C can be
  - Invalid: B is not cached in C
    - To read or write, must make a request on the bus
  - Modified: B is dirty in C
    - C has the block, no other cache has the block, and C must update memory when it displaces B
    - Can read or write B without going to the bus
  - Shared: B is clean in C
    - C has the block, other caches have the block, and C need not update memory when it displaces B
    - Can read B without going to bus
    - To write, must send an upgrade request to the bus
  - Exclusive: B is exclusive to cache C
    - Can help to eliminate bus traffic
    - E state not absolutely necessary

Board notes are a good resource

Cache to Cache transfers

- Problem
  - P1 has block B in M state
  - P2 wants to read B, puts a RdReq on bus
  - If P1 does nothing, memory will supply the data to P2
  - What does P1 do?
- Solution 1: abort/retry
  - P1 cancels P2's request, issues a write back
  - P2 later retries RdReq and gets data from memory
  - Too slow (two memory latencies to move data from P1 to P2)
- Solution 2: intervention
  - P1 indicates it will supply the data ("intervention" bus signal)
  - Memory sees that, does not supply the data, and waits for P1's data
  - P1 starts sending the data on the bus, memory is updated
  - P2 snoops the transfer during the write-back and gets the block

Shared Memory Performance

- Another "C" for cache misses
  - Still have Compulsory, Capacity, Conflict
  - Now have Coherence, too
    - We had it in our cache and it was invalidated
- Two sources for coherence misses
  - True sharing
    - Different processors access the same data
  - False sharing
    - Different processors access different data, *but they happen to be in the same block*

Communication NWs add real-life overheads

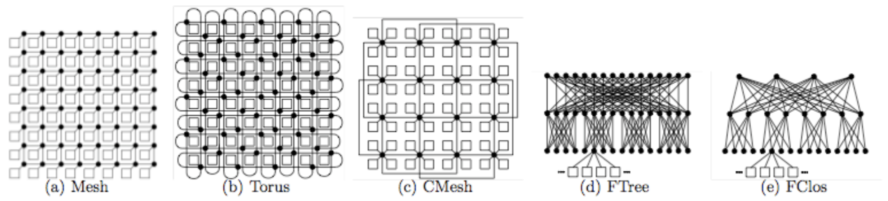


Figure 8: Network Topologies

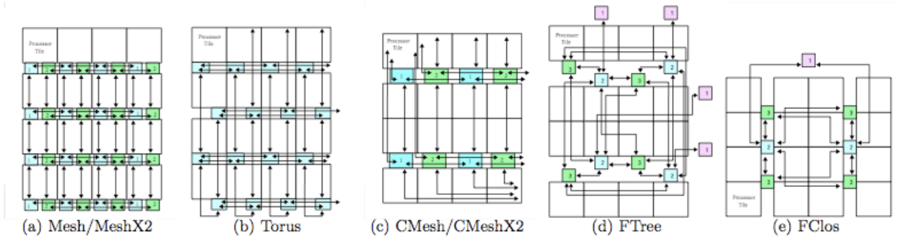


Figure 9: Placement of Routers used to Estimate Area (Lower Left Quadrant)

From Balfour, Dally, *Supercomputing*

Overhead is function of traffic…

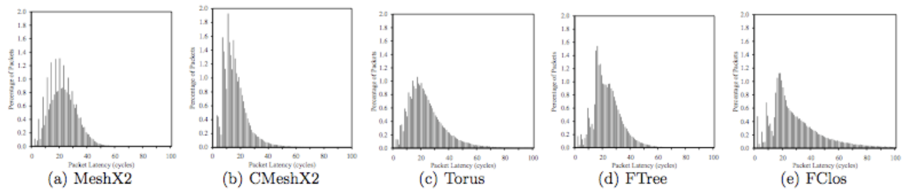


Figure 11: Workload Packet Latency Distribution for Uniform Random Traffic Pattern

From Balfour, Dally, *Supercomputing*