Lecture 29
Review

Suggested reading:

Everything

Be sure you understand CC, clock period

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

…

I[15]: Add R2, R1, R4

I[16]: MOV R3, 8

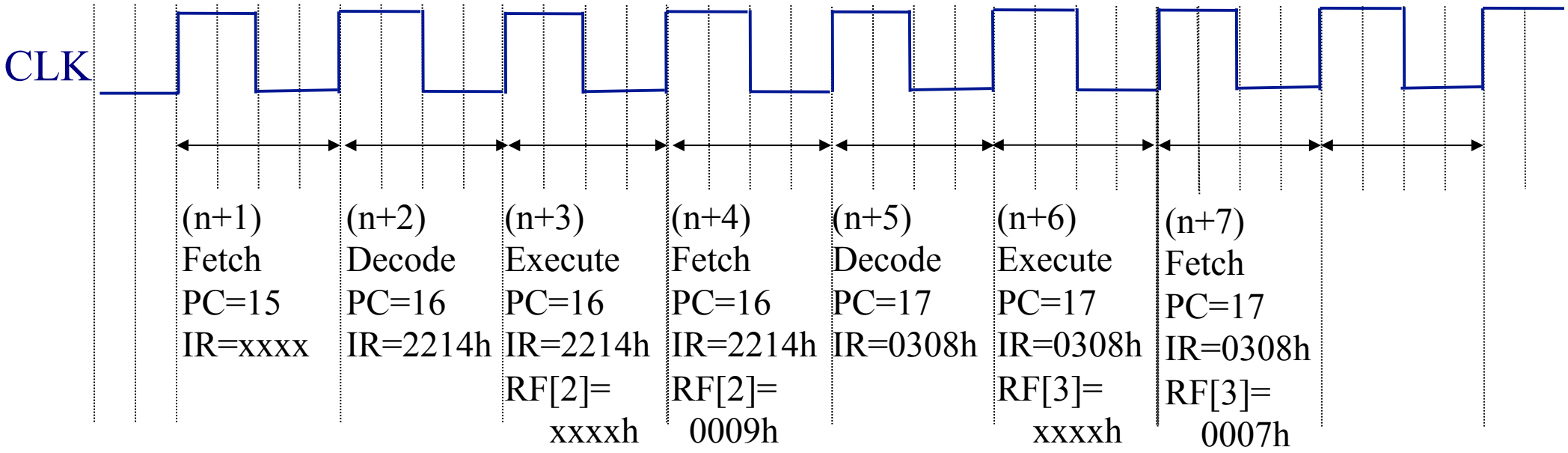
I[17]: Add R2, R2, R3

…

RF[1] = 4

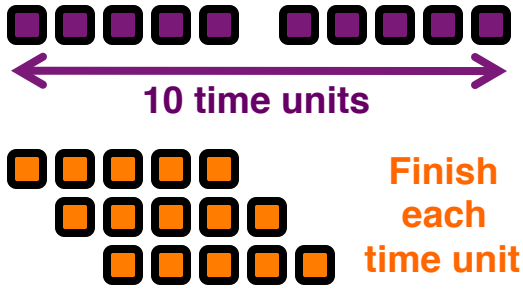
RF[4] = 5

D[8] = 7



Common (and good) performance metrics

- latency: response time, execution time
  - good metric for fixed amount of work (minimize time)
- throughput: work per unit time
  - = (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)

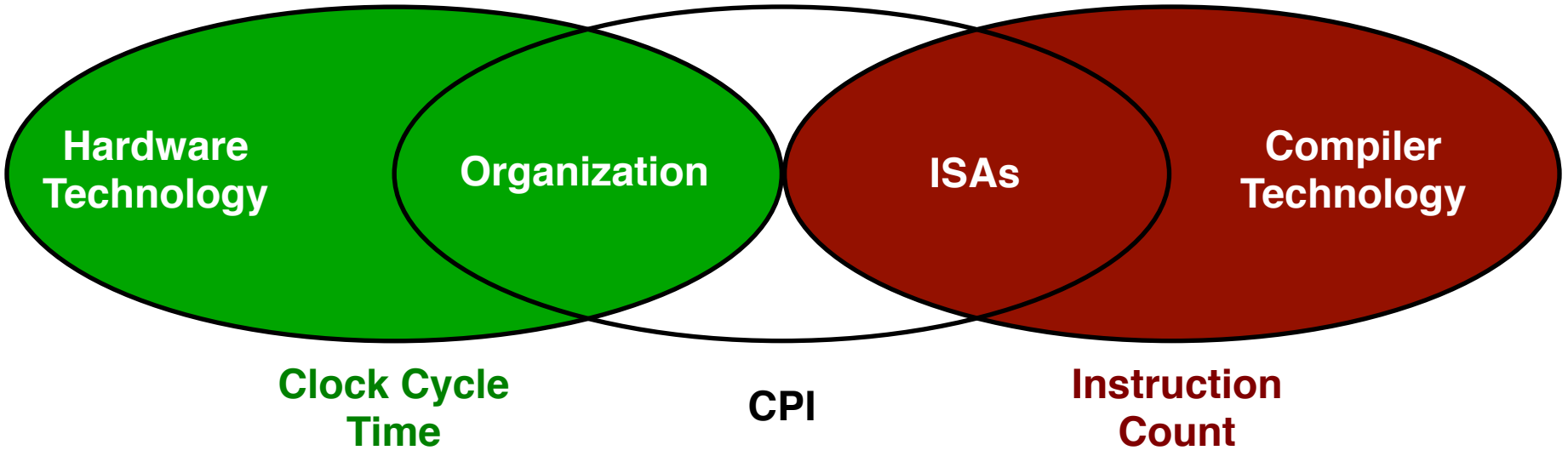


- comparing performance
  - A is N times faster than B if and only if:
    - time(B)/time(A) = N
  - A is X% faster than B if and only if:
    - time(B)/time(A) = 1 + X/100

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 ✕ % improvement in any one variable leads to an ✕ % improvement in CPU performance
- But, everything usually affects everything:

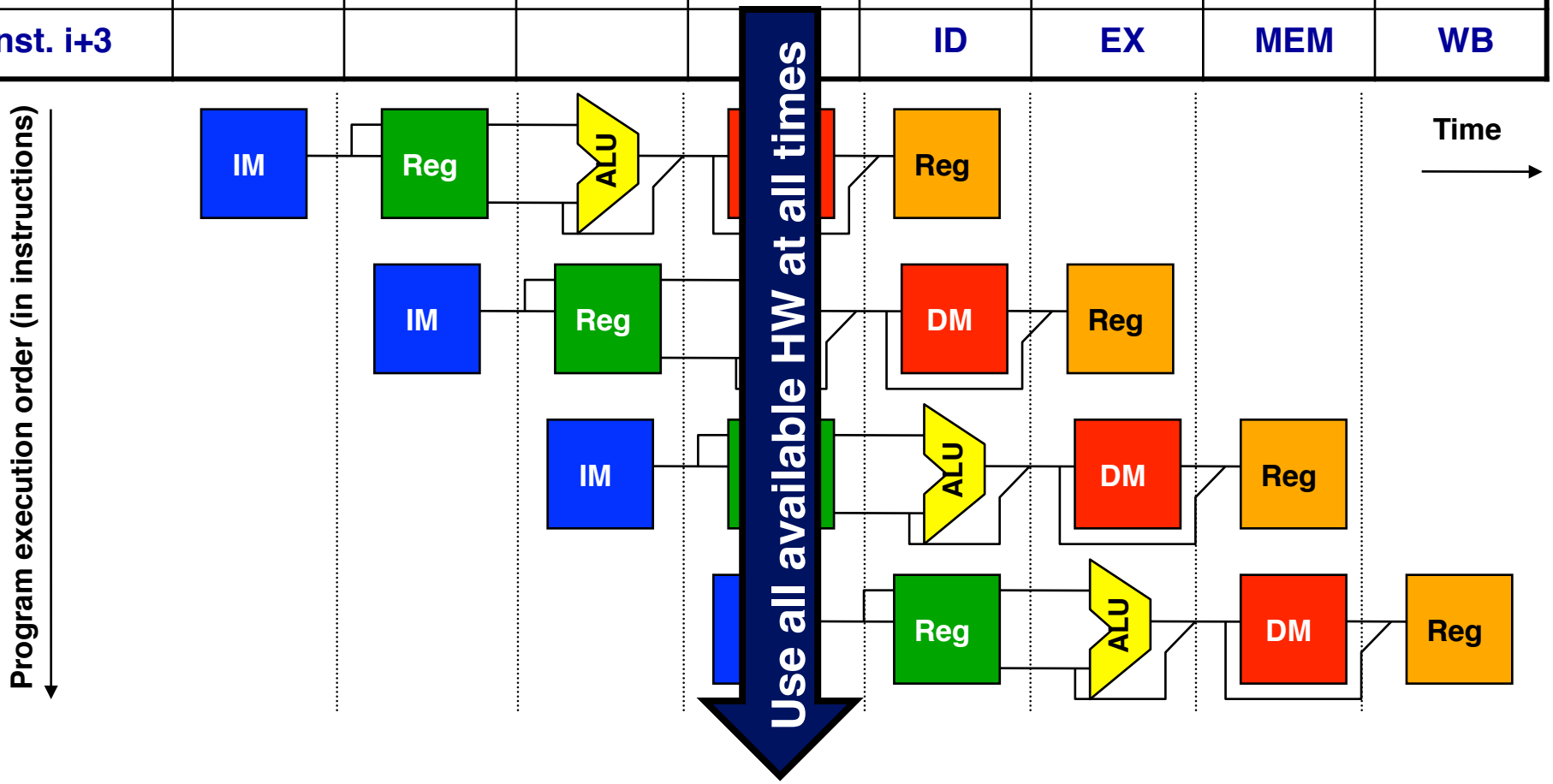


PIPELINES

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					ID	EX	MEM	WB



Time for N instructions in a pipeline:

- 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: NT + (S-1)T

Throughput: Time per initiation = T + (S-1)T/N → T!

  - (assumes no stalls)

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

Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle

- CPI pipelined =
  - Ideal CPI + Pipeline stall cycles per instruction
  - 1 + Pipeline stall cycles per instruction

- Ignoring overhead and assuming stages are balanced:

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

- Ideally, speedup equal to # of pipeline stages

Stalls occur because of hazards!

Pipelining hazards

- **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)**

Structural hazards (why)

- 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

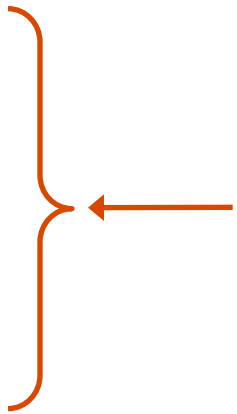
- Most common instances of structural hazards:
  - When some resource not duplicated enough

Data hazards (why)

- 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

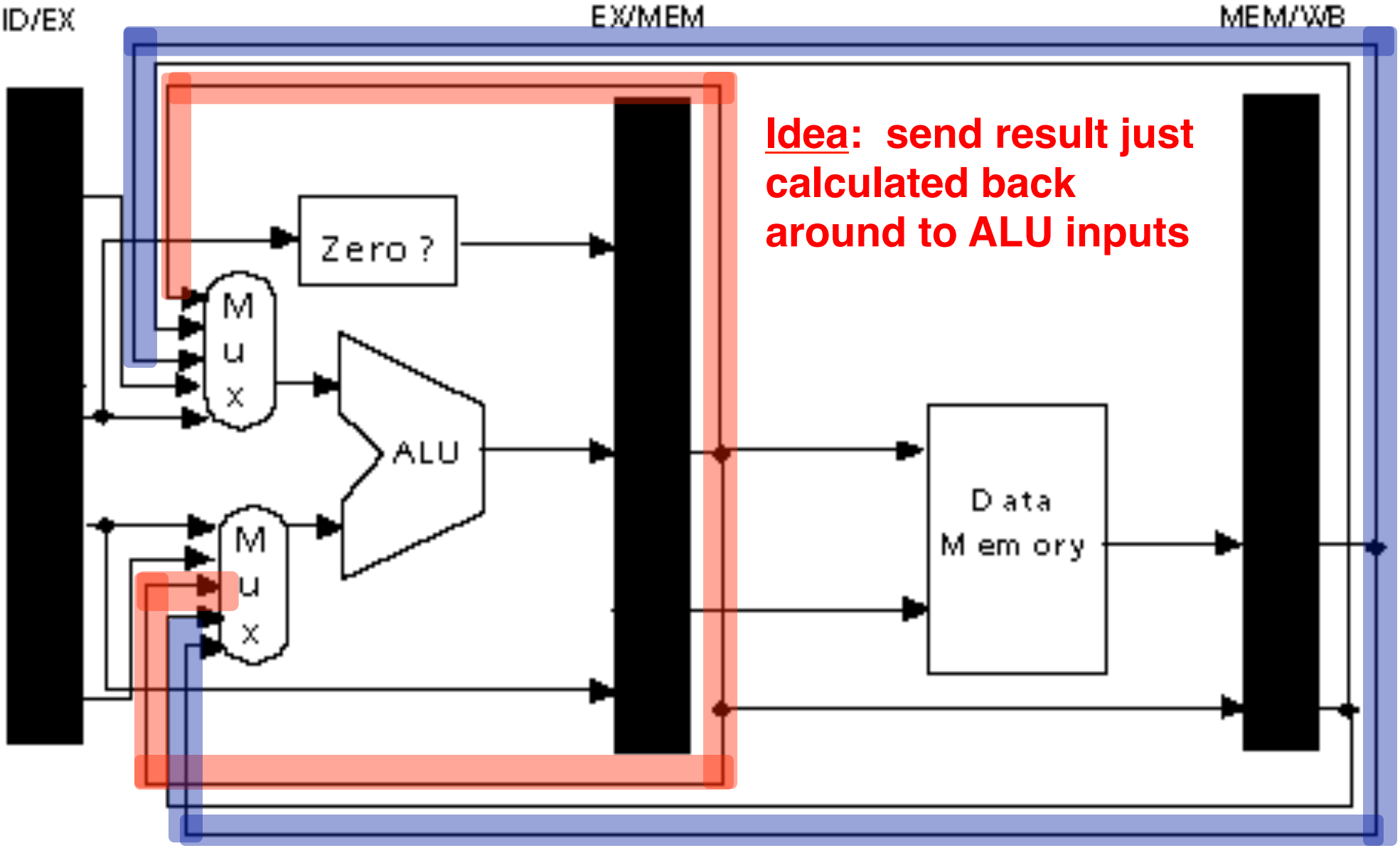
ADD writes the register in WB but SUB needs it in ID.

This is a data hazard

Data hazards (avoiding with forwarding)

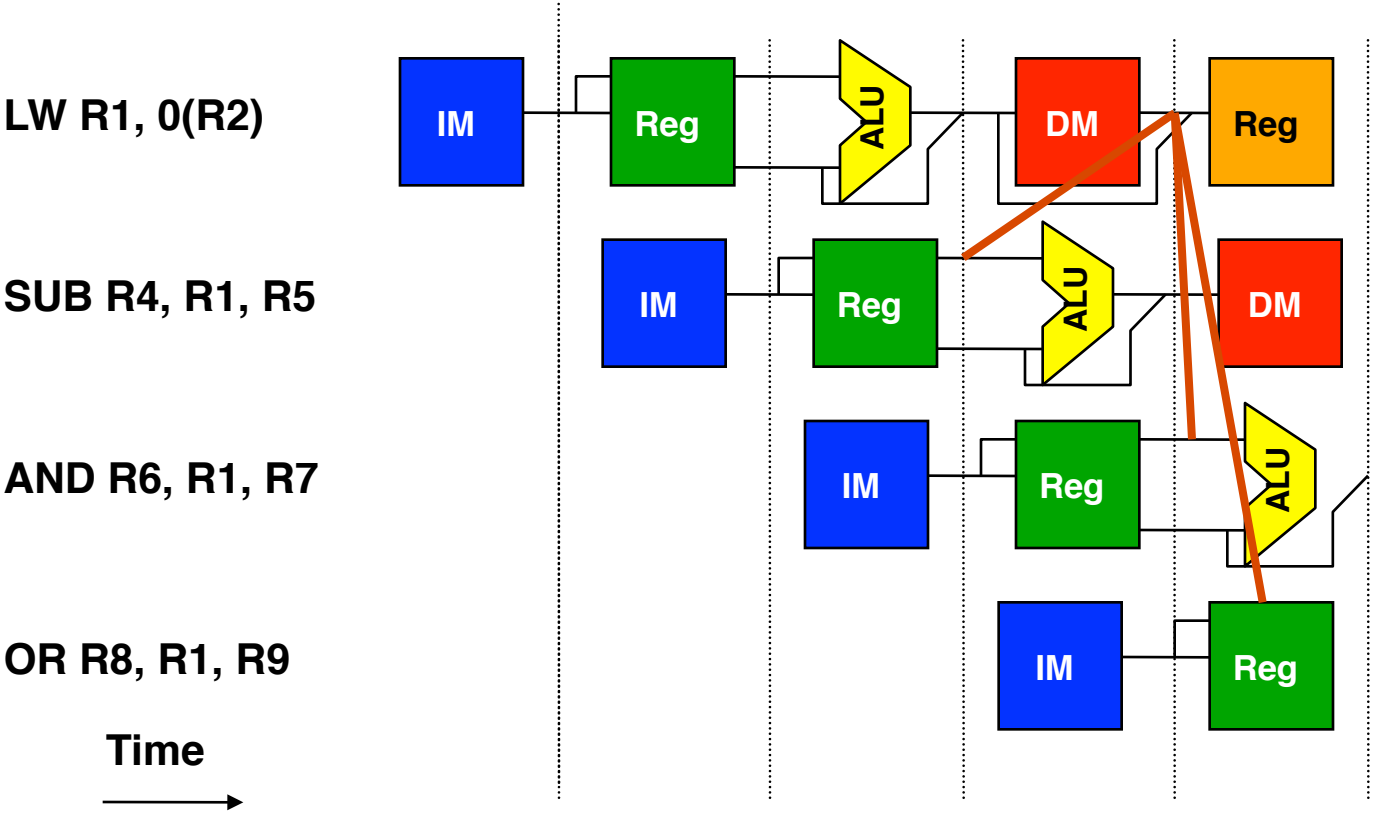
- Problem illustrated on previous slide can actually be solved relatively easily – with forwarding
- In this example, result of the ADD instruction not really needed until after ADD actually produces it
- Can we move the result from EX/MEM register to the beginning of ALU (where SUB needs it)?
  - Yes! 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

Data hazards (avoiding with forwarding)



Send output of memory back to ALU too…

Data hazards (forwarding sometimes fails)



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.

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

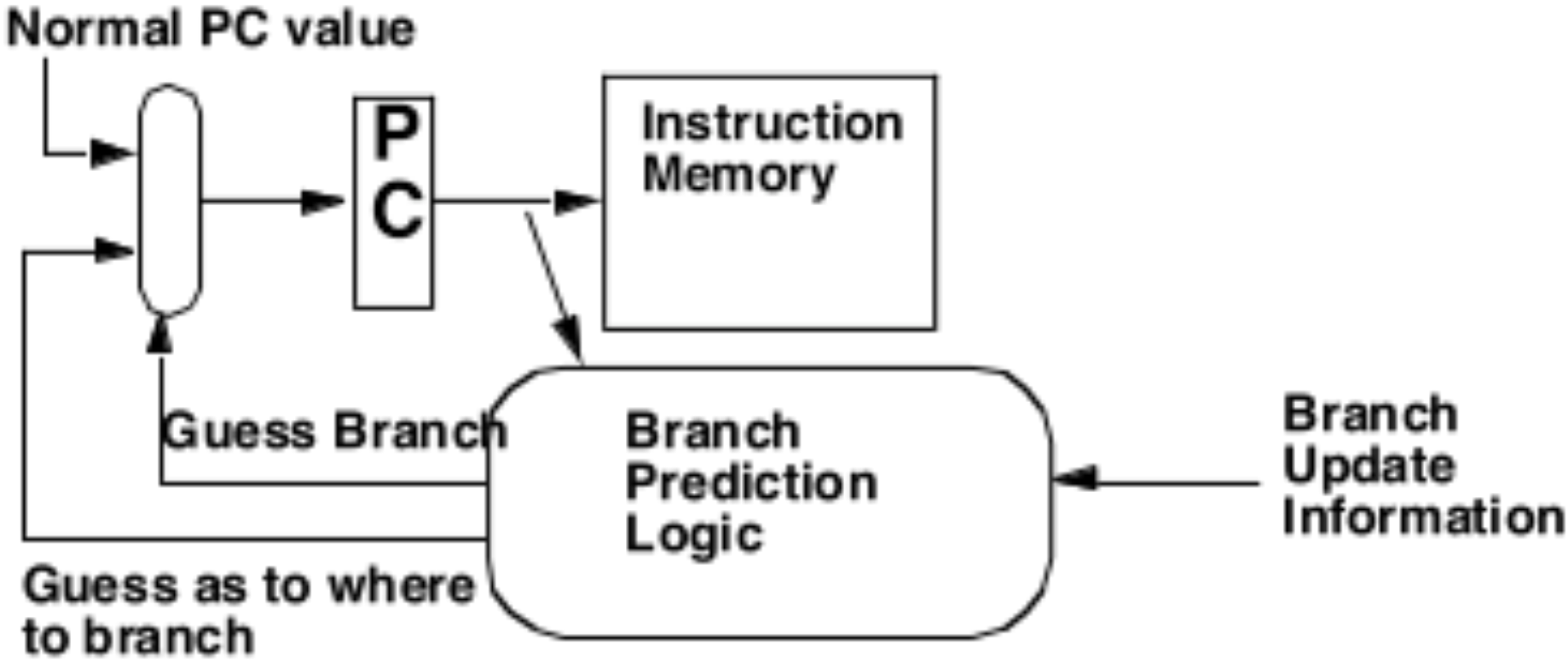
Branch / Control Hazards

- 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)

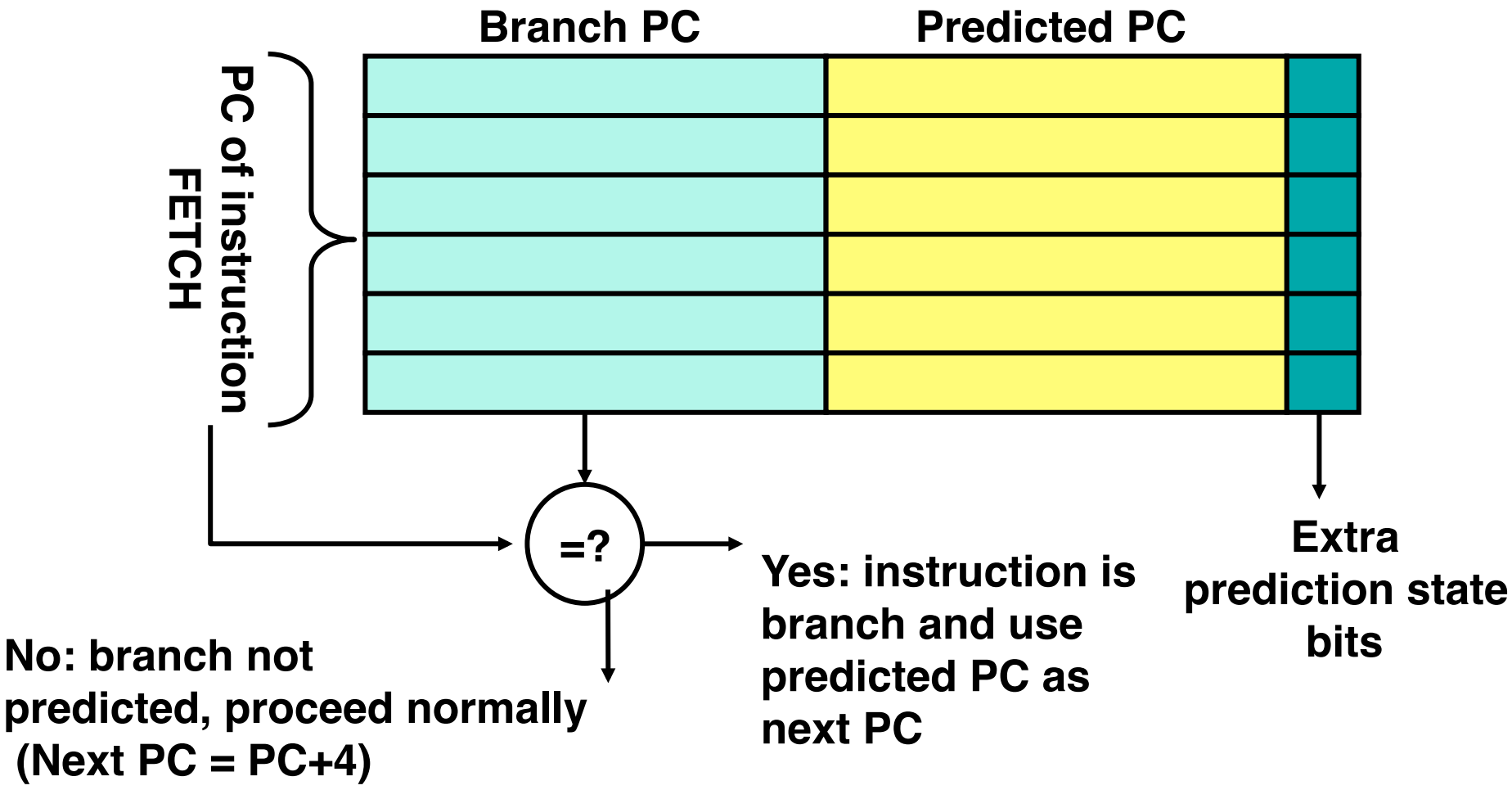
- How long will it take before branch decision?
  - What happens in the meantime?

A Branch Predictor



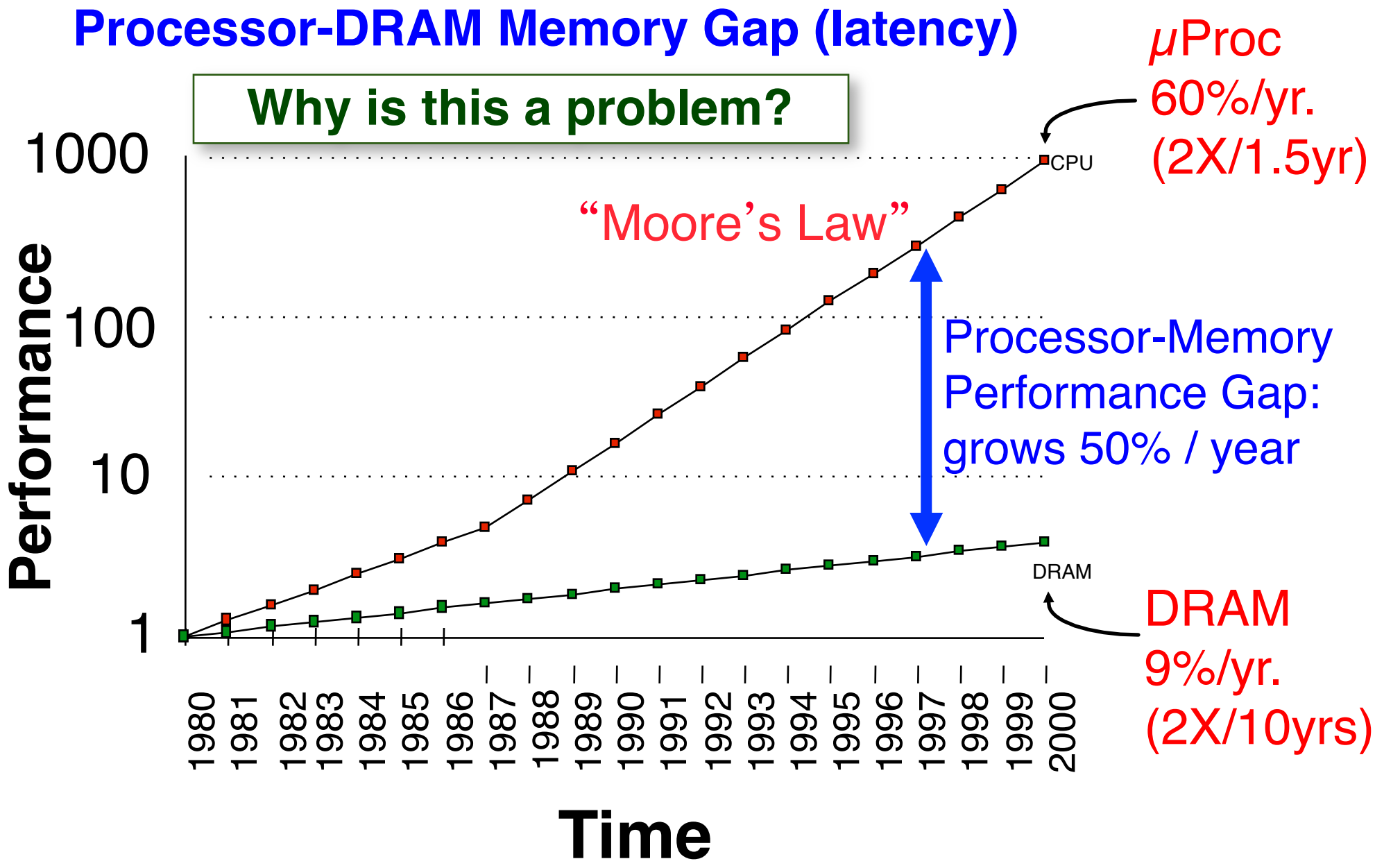
Branch prediction critical path (BTB)

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address
- Example: BTB combined with BHT



MEMORY HIERARCHIES

Is there a problem with DRAM?



A common memory hierarchy

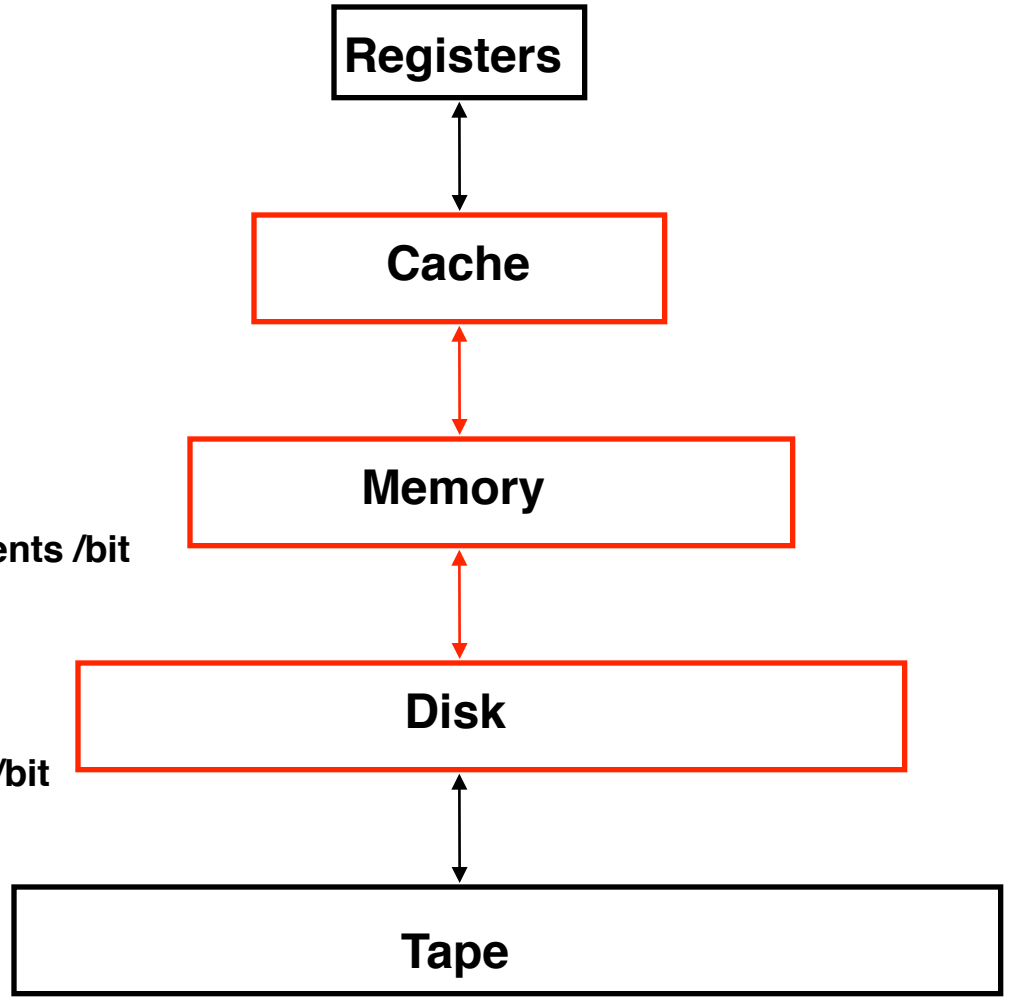
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
10^{-8}



Upper Level
faster
Larger
Lower Level

Average Memory Access Time

AMAT = (Hit Time) + (1 - h) x (Miss Penalty)

- **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

- Introduces *caches* to improve hit time.

Where can a block be placed in a cache?

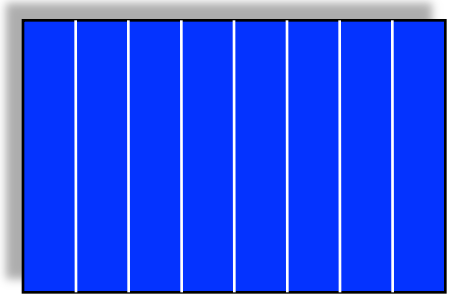
- 3 schemes for block placement in a cache:
  - Direct mapped cache:
    - Block can go to only 1 place in cache
    - Usually:
      - (Block address) MOD (# of blocks in the cache)
  - Fully associative cache:
    - Block can be placed anywhere in cache
  - Set associative cache:
    - A *set* is a 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

Where can a block be placed in a cache?

Cache:

Fully Associative

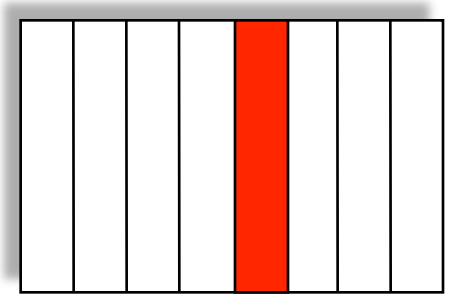
0 1 2 3 4 5 6 7



Block 12 can go anywhere

Direct Mapped

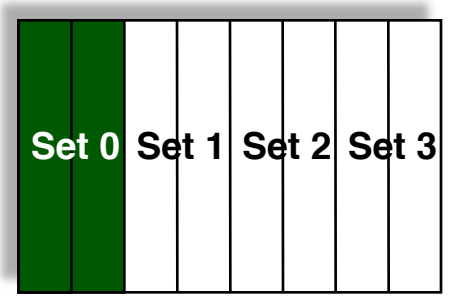
0 1 2 3 4 5 6 7



Block 12 can go only into Block 4 (12 mod 8)

Set Associative

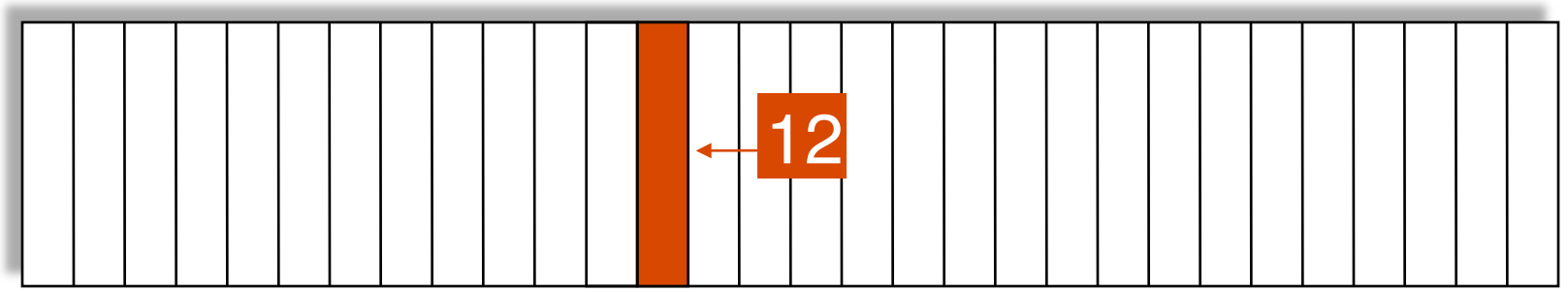
0 1 2 3 4 5 6 7



Block 12 can go anywhere in set 0 (12 mod 4)

Memory:

0 1 2 3 4 5 6 7 8 ...



How is a block found in the cache?

- Cache's have address tag on each block frame that provides block address
  - Tag of every cache block that might have entry is examined against CPU address (in parallel! – why?)

- Each entry usually has a valid bit
  - Tells us if cache data is useful/not garbage
  - If bit is not set, there can't be a match…

- If block data is updated, set a dirty bit
  - Block data must be written back to higher levels of memory hierarchy upon replacement

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

Reducing cache misses

- Want data accesses to result in cache hits, not misses for best performance
- Start by looking at ways to increase % of hits….
- …but first look at 3 kinds of misses
  - Compulsory misses:
    - 1st access to cache block will not be a hit – data not there yet!
  - Capacity misses:
    - Cache is only so big.
    - Can't 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

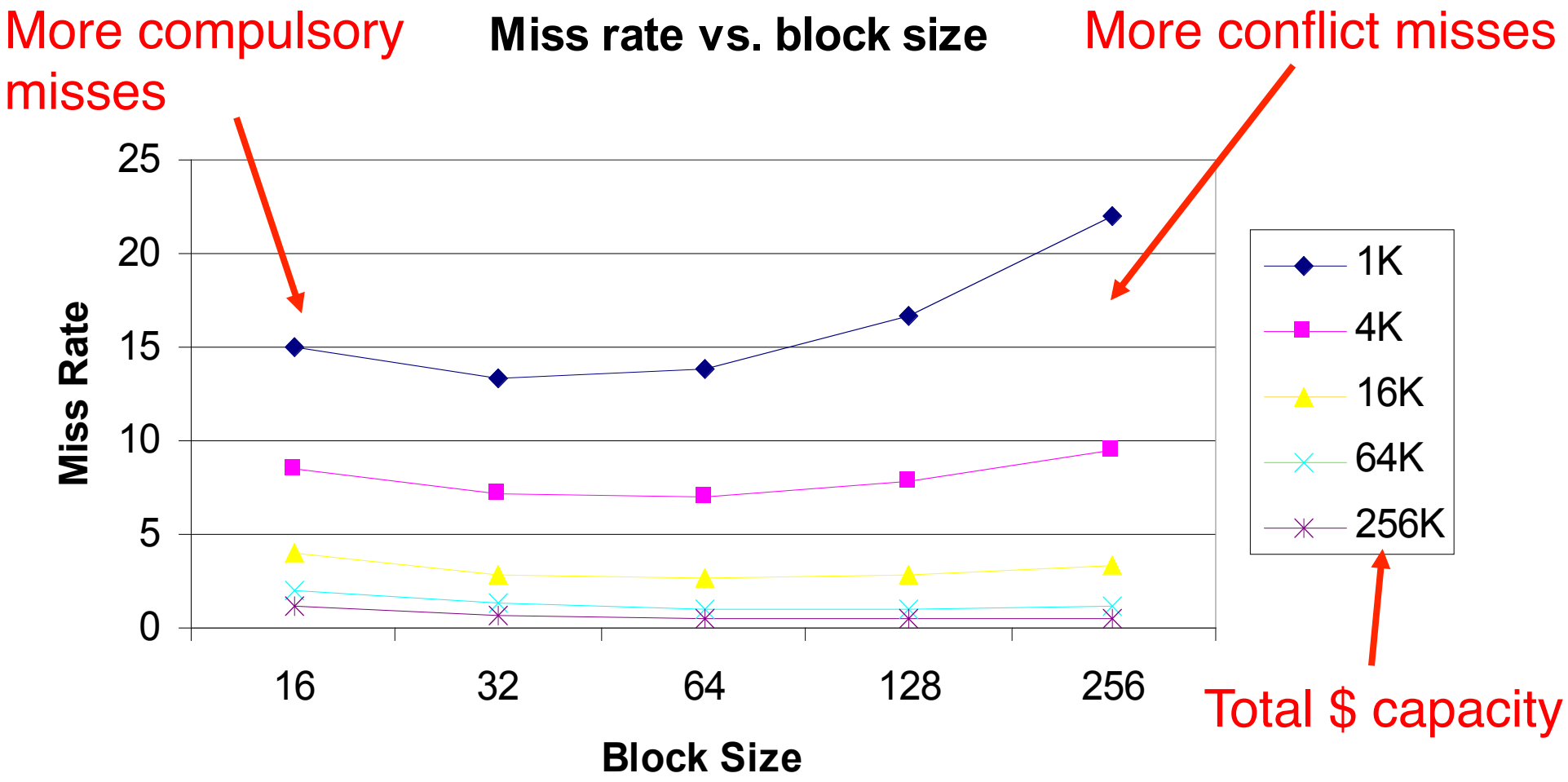
Impact of larger cache blocks:

- Can reduce miss by increasing cache block size
  - This will help eliminate what kind of misses?

- Helps improve miss rate b/c of principle of locality:
  - Temporal locality says that if something is accessed once, it'll probably be accessed again soon
  - Spatial locality says that if something is accessed, something nearby it will probably be accessed
    - Larger block sizes help with spatial locality

- Be careful though!
  - Larger block sizes can increase miss penalty!
    - Generally, larger blocks reduce # of total blocks in cache

Impact of larger cache blocks:



(Assuming total cache size stays constant for each curve)

Second-level caches

- Introduces new definition of AMAT:
  - 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

Virtual Memory

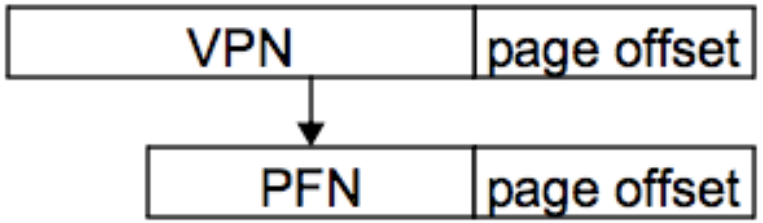
- Computers run lots of processes simultaneously
  - No full address space of memory for each process
    - Physical memory expensive and not dense - thus, too small
  - Share smaller amounts of physical memory among many processes

- Virtual memory is the answer
  - Divide physical memory into blocks, assign 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 Memory: The Story

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

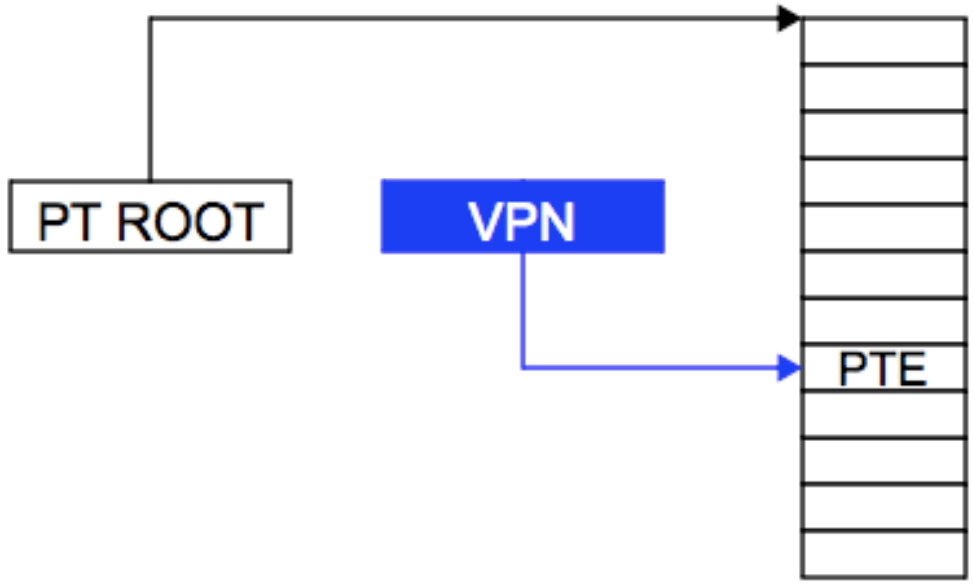
- blocks called *pages*
- processes use *virtual addresses (VA)*
- physical memory uses *physical addresses (PA)*
- address divided into page offset, page number
  - virtual: virtual page number (VPN)
  - physical: page frame number (PFN)
- *address translation*: system maps VA to PA (VPN to PFN)
- e.g.,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



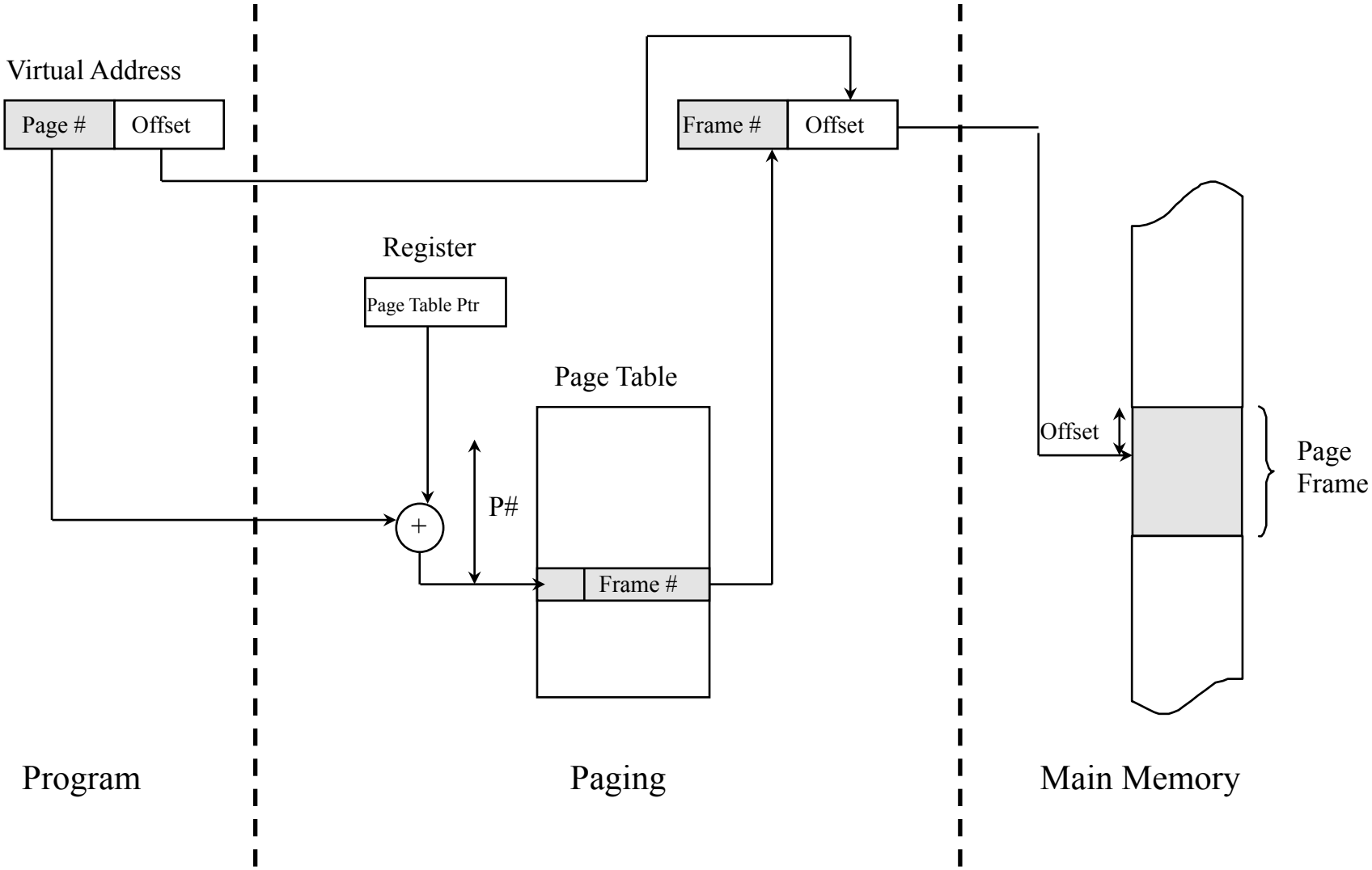
Address Translation: Page Tables

OS performs address translation using a *page table*

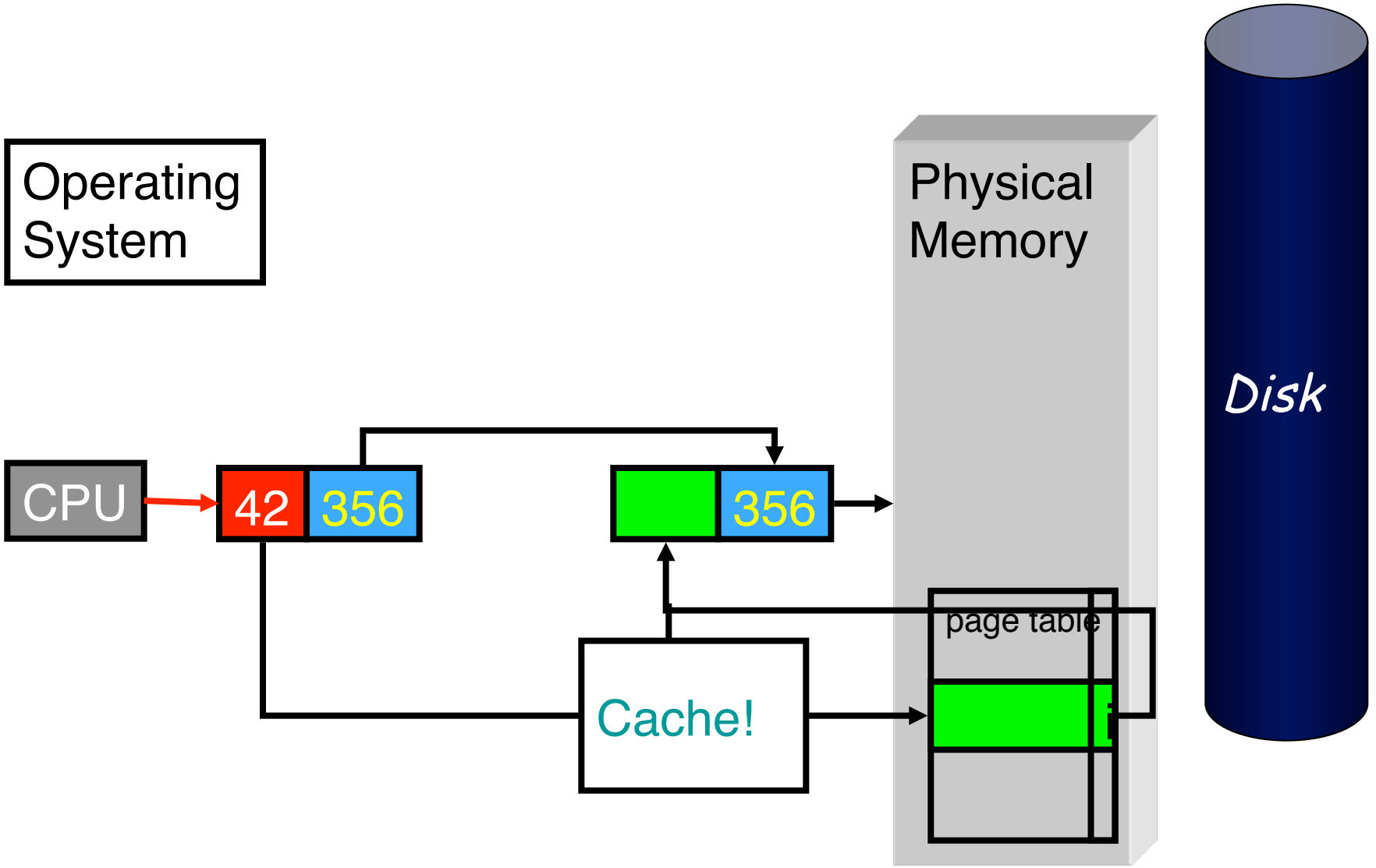
- each process has its own page table
  - OS knows address of each process' page table
- a page table is an array of *page table entries (PTEs)*
  - one for each VPN of each process, indexed by VPN
- each PTE contains
  - PFN
  - permission
  - dirty bit
  - LRU state
  - e.g., 4-bytes total



Review: Address Translation



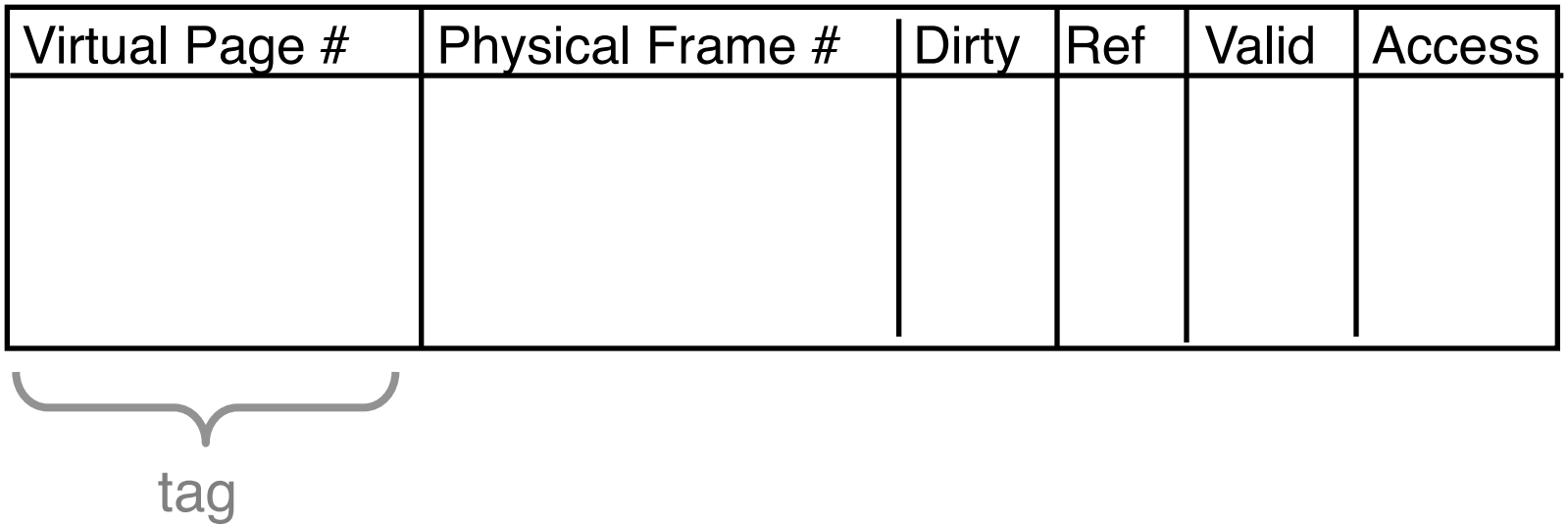
Page table lookups = memory references



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

TLBs speed up VA translations

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is *Translation Lookaside Buffer* or *TLB*



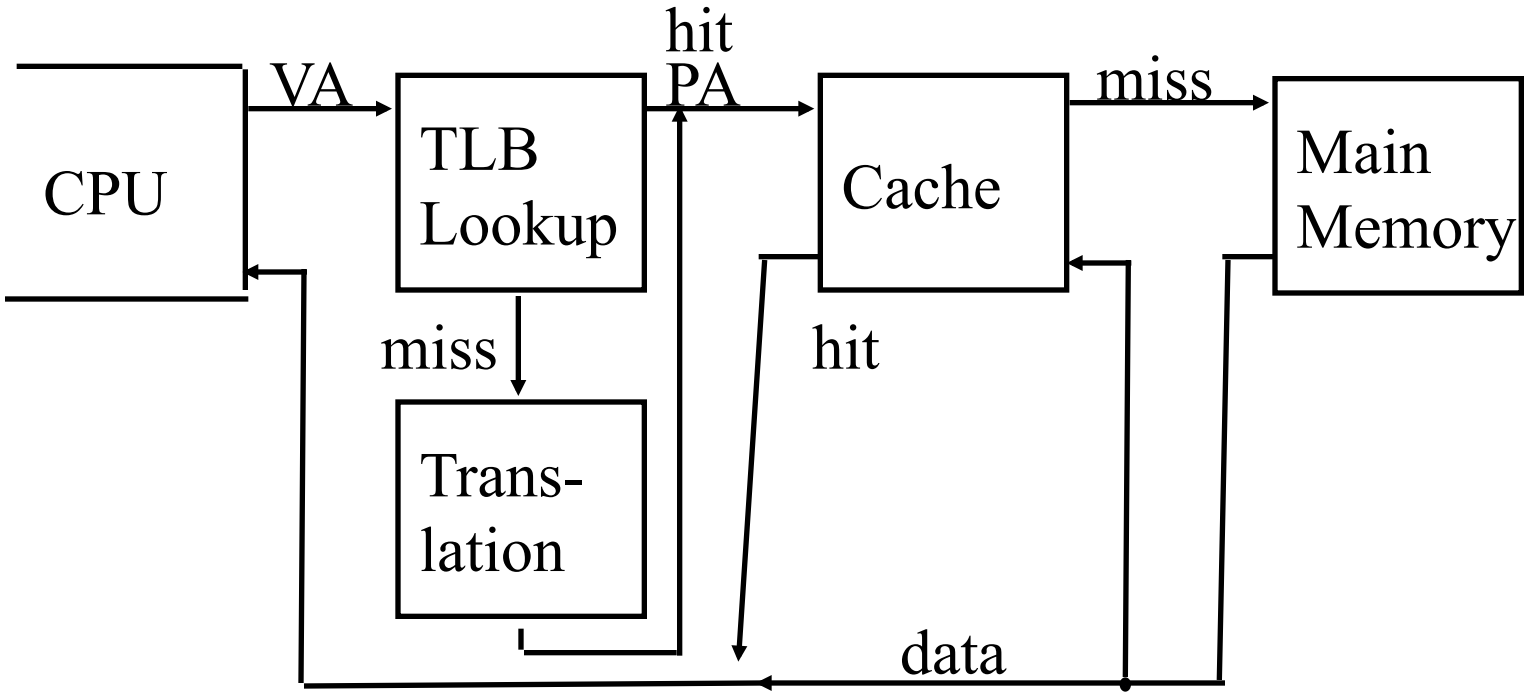
Really just a cache (a special-purpose cache) on the page table mappings

TLB access time comparable to cache access time
(much less than main memory access time)

Critical path of an address translation (1)

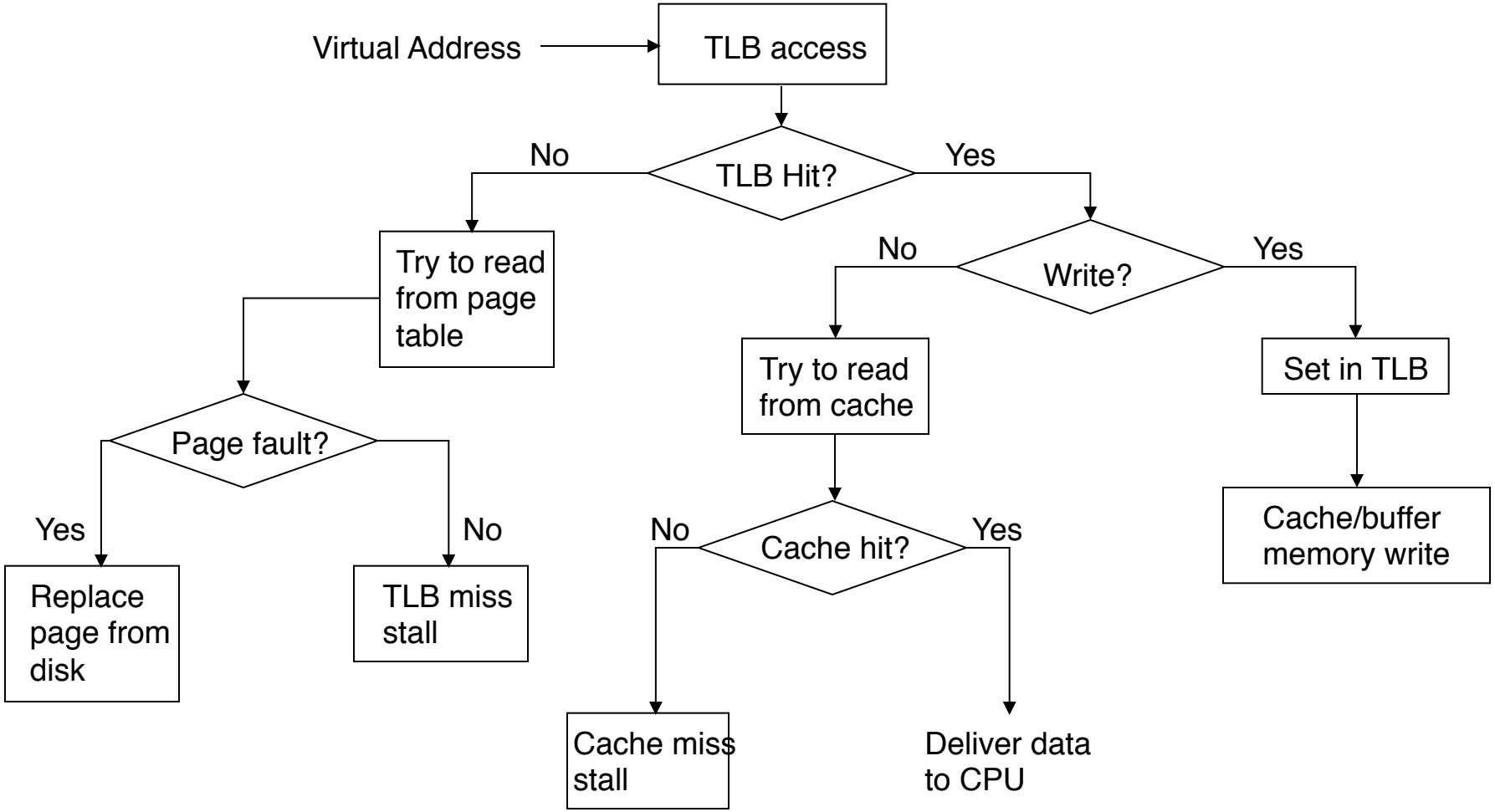
Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

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.



Translation with a TLB

Critical path of an address translation (2)



Disk Performance Example

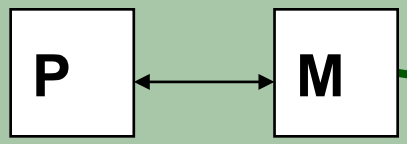
- parameters
  - 3600 RPM ⇒ 60 RPS (may help to think in units of tracks/sec)
  - avg seek time: 9ms
  - 100 sectors per track, 512 bytes per sector
  - controller + queuing delays: 1ms
- Q: average time to read 1 sector (512 bytes)?
  - $rate_{transfer}$ = 100 sectors/track * 512 B/sector * 60 RPS = 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)

PARALLEL PROCESSING

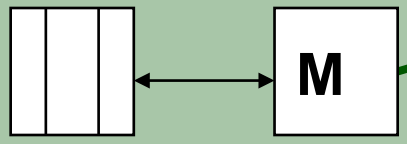
The many ways of parallel computing

Processors can be any of these configurations

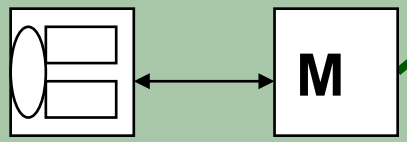
- Uni:



- Pipelined

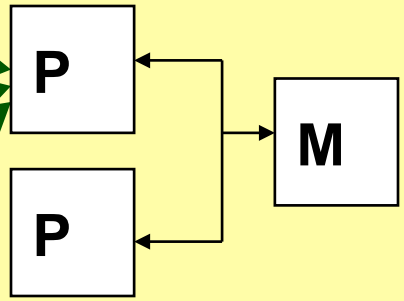


- Superscalar

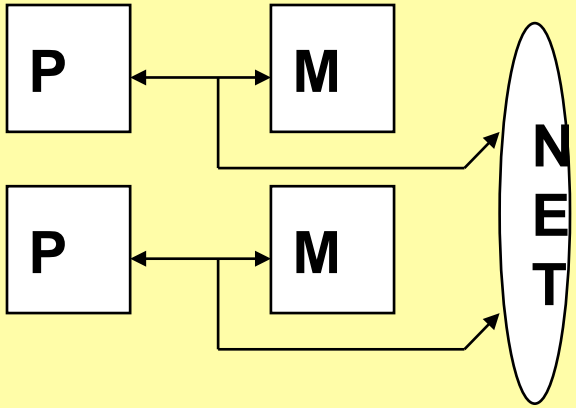


SISD

- Centralized Shared Memory



- Distributed Shared Memory

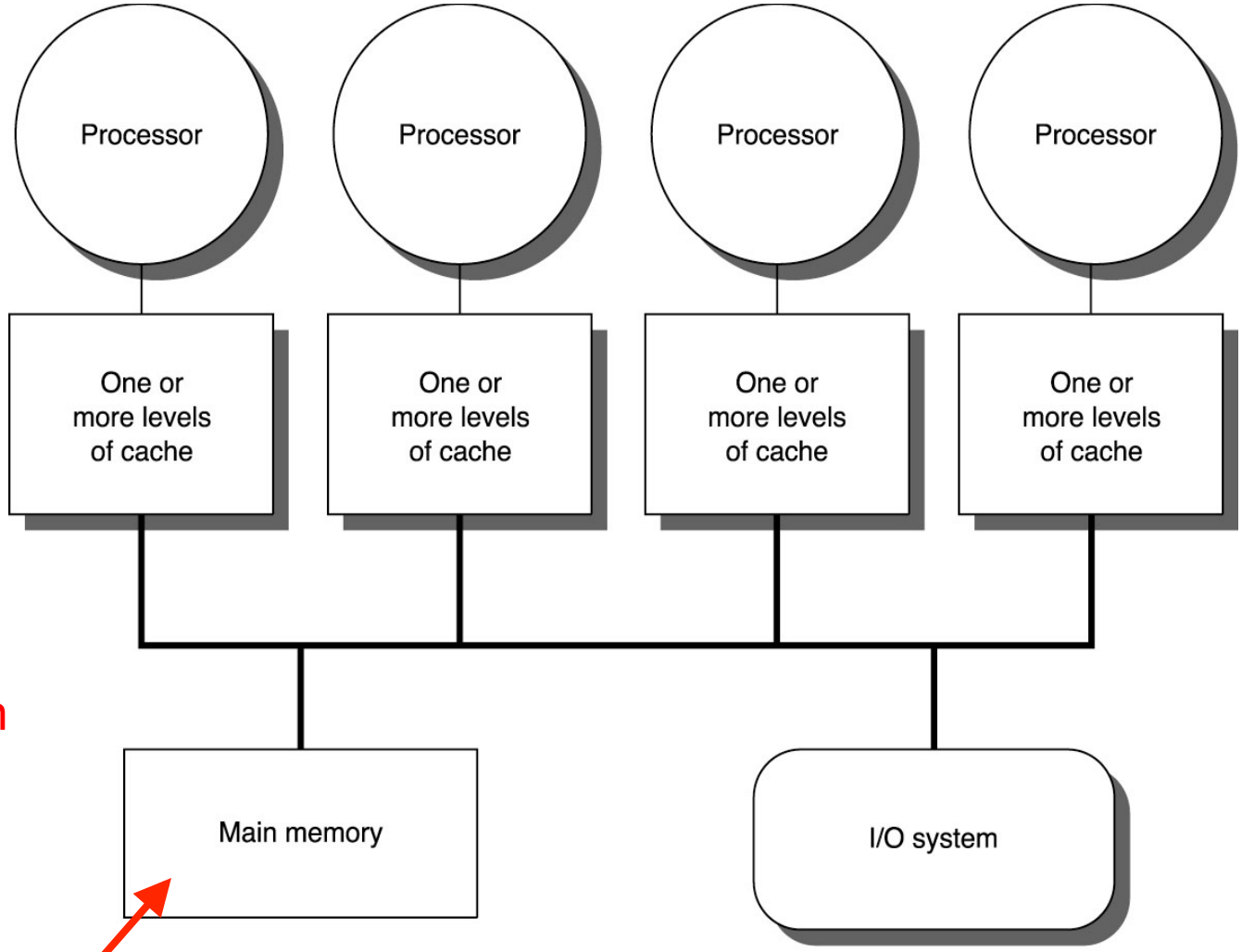


MIMD

This brings us to MIMD configurations…

MIMD Multiprocessors

Centralized Shared Memory

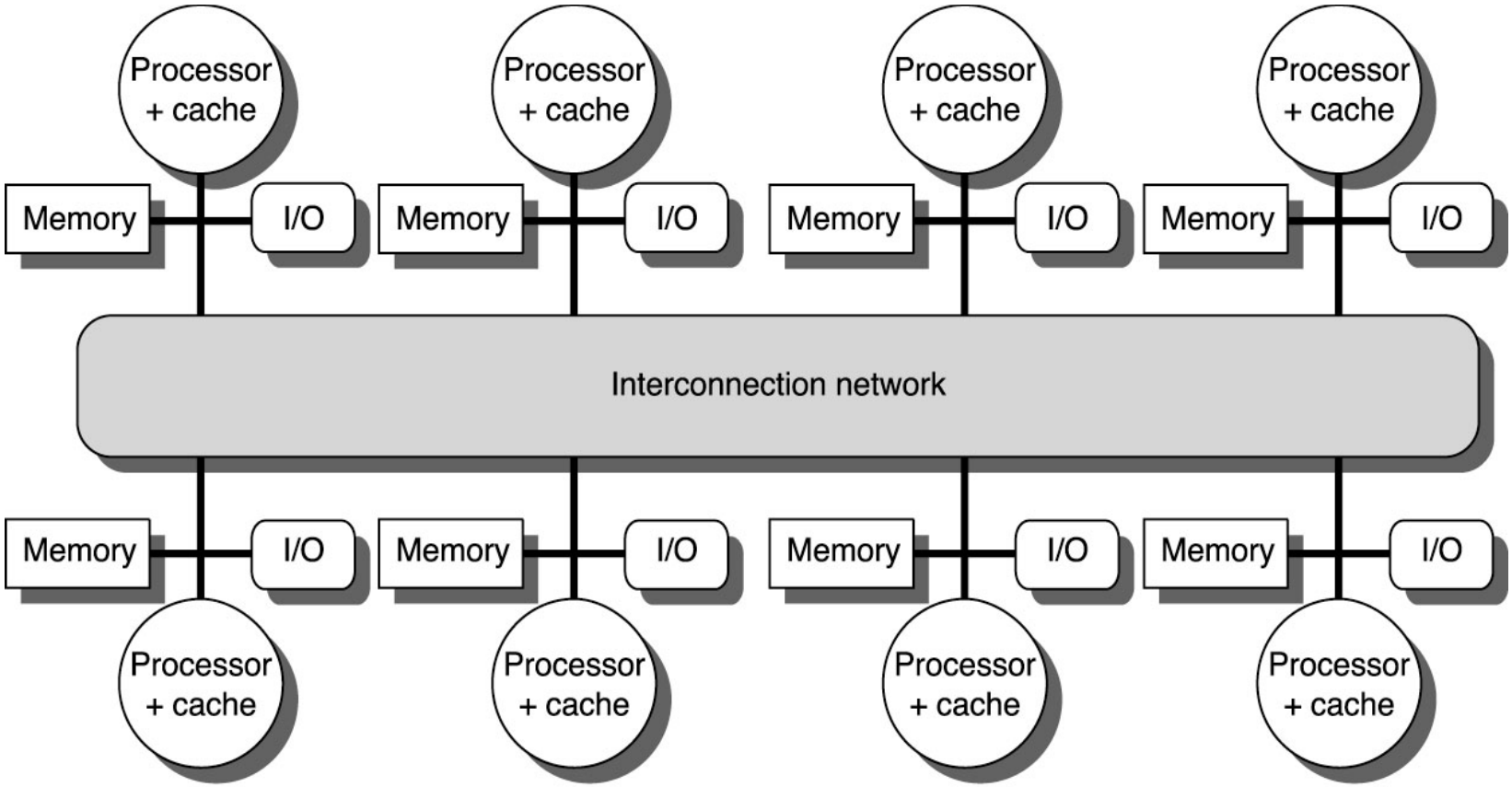


Actually, quite representative of quad core chip, with "main memory" being a shared, L2 cache.

Note: just 1 memory

MIMD Multiprocessors

Distributed Memory

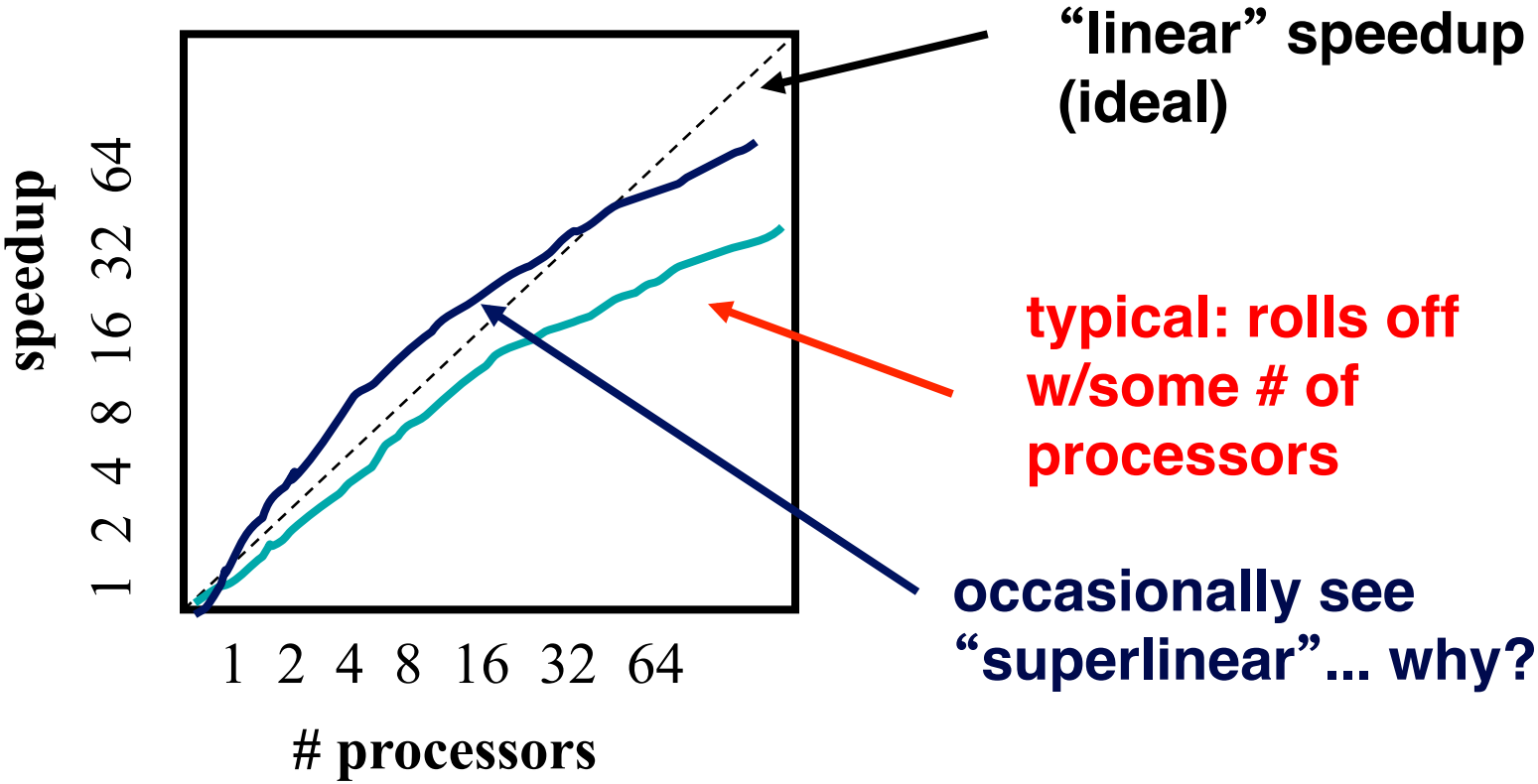


Multiple, distributed memories here.

Speedup from parallel processing

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
- **Overhead where no actual processing is done.**

Impediments to parallel performance

★ **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)**

- **All ★'ed items also affect speedup that could be obtained, add overheads**

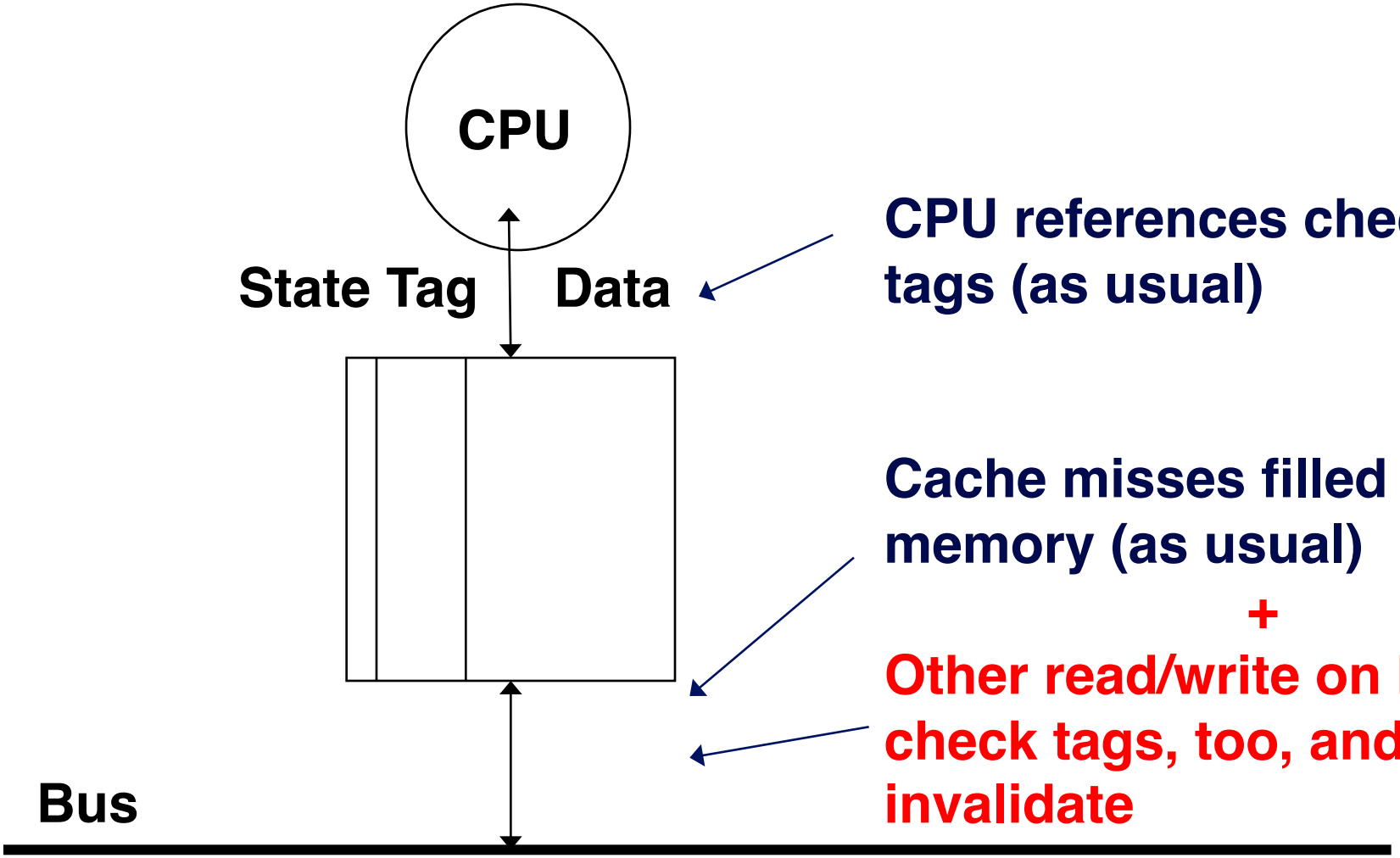
$$\text{Speedup} = \frac{1}{\left[1 - \text{Fraction}_{\text{parallelizable}}\right] + \frac{\text{Fraction}_{\text{parallelizable}}}{N}}$$

★ **Overhead where no actual processing is done.**

Maintaining Cache Coherence

- Hardware schemes
  - Shared Caches
    - Trivially enforces coherence
    - Not scalable (L1 cache quickly becomes a bottleneck)
  - Snooping
    - Needs a broadcast network (like a bus) to enforce coherence
    - Each cache that has a block tracks its sharing state on its own
  - Directory
    - Can enforce coherence even with a point-to-point network
    - A block has just one place where its full sharing state is kept

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?

What happens on write conflicts? (invalidate protocol)

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

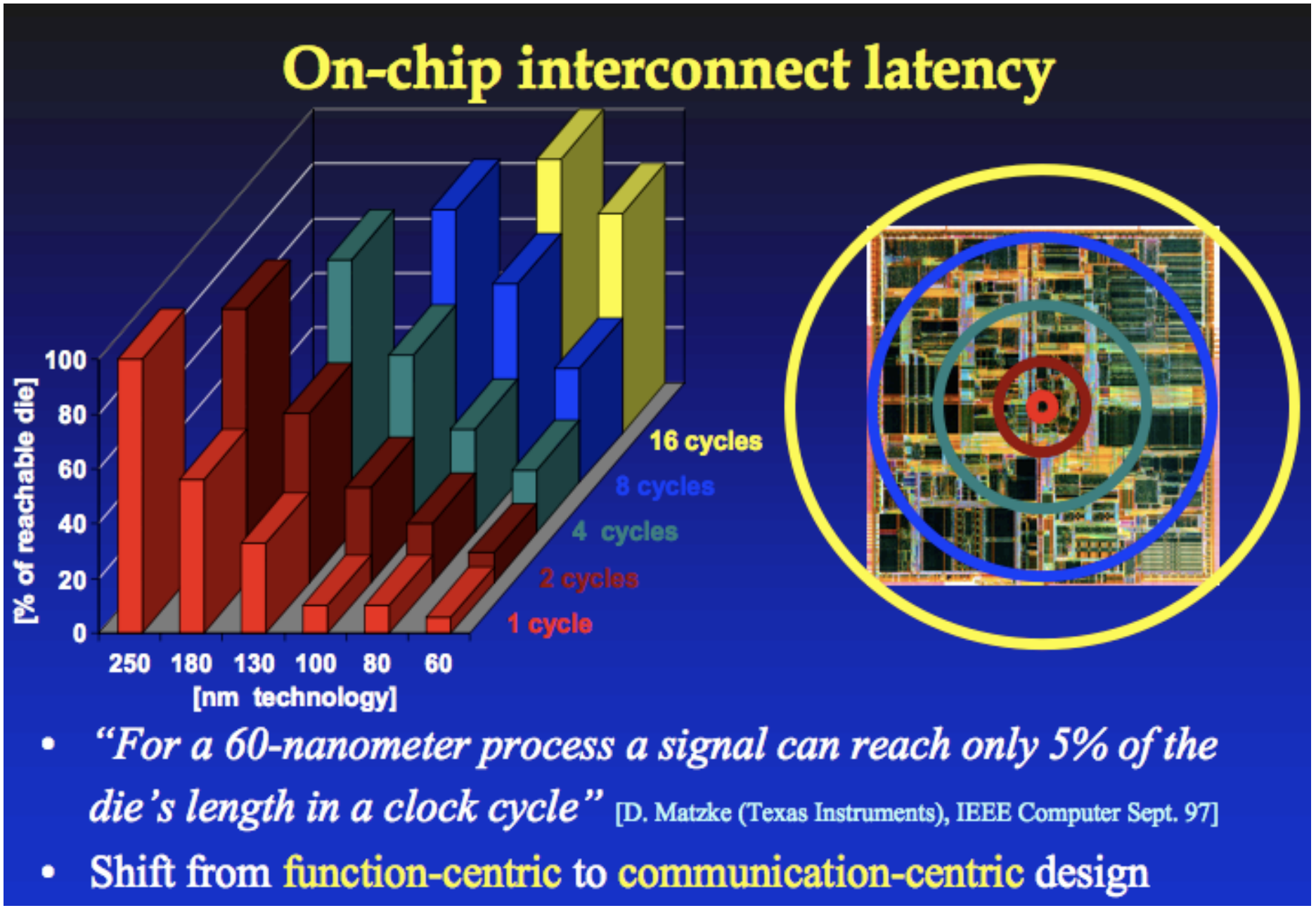
Cache to Cache transfers

- Problem
  - P1 has block B in M state
  - P2 wants to read B, puts a read request 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

Latency



(and this is just the physics – no router overhead, etc. included)

NW topologies facilitate communication among different processing nodes…

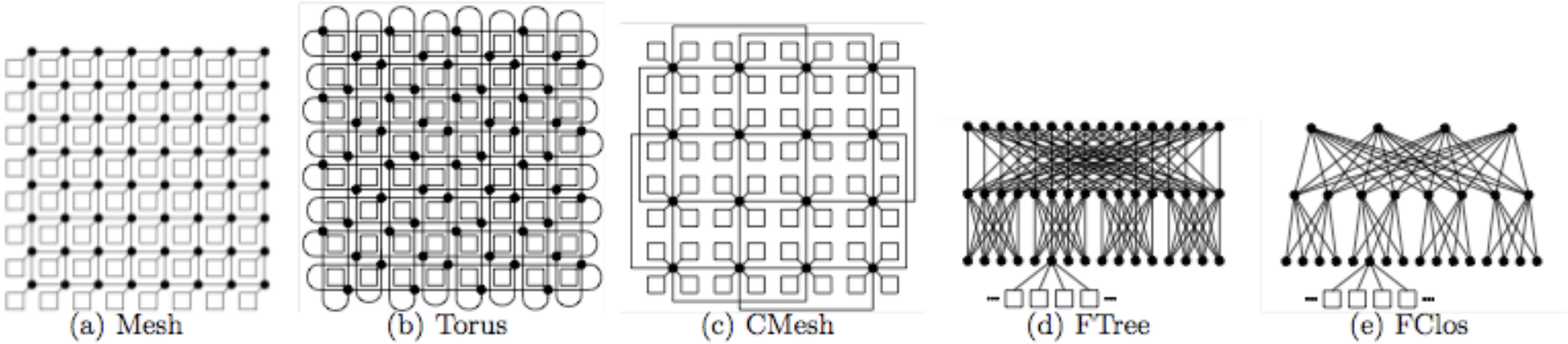


Figure 8: Network Topologies

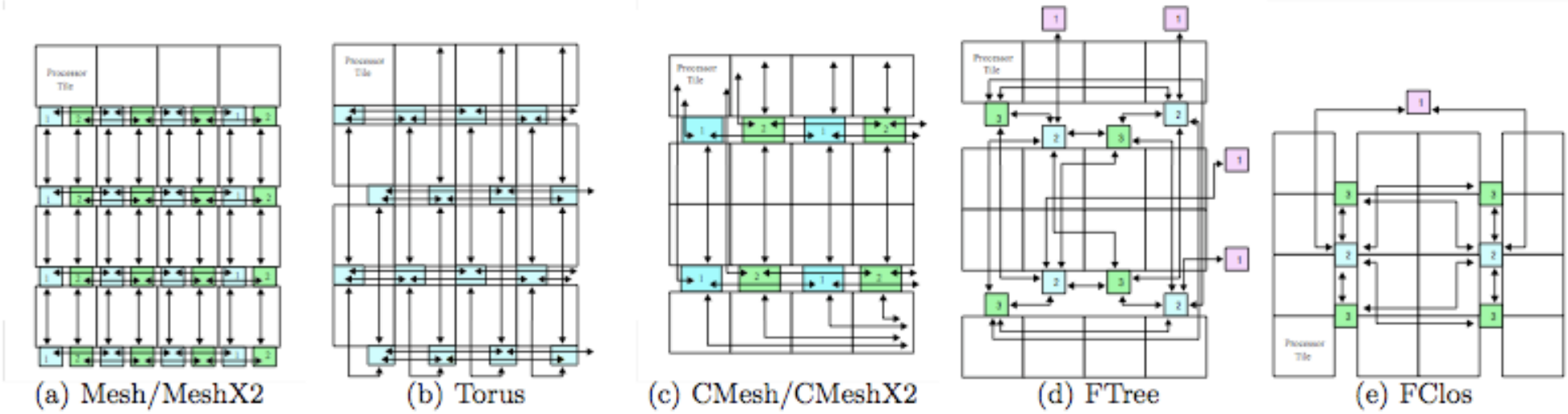


Figure 9: Placement of Routers used to Estimate Area (Lower Left Quadrant)

…but add additional overhead

- **Time to message =**

  # of hops (routers) x time per router +

  # of links x time per link +

  serialization latency (ceiling(message size / link BW))

Load balancing

- Load balancing: keep all cores busy at all times
  - (i.e. minimize idle time)
- Example:
  - If all tasks subject to barrier synchronization, slowest task determines overall performance: