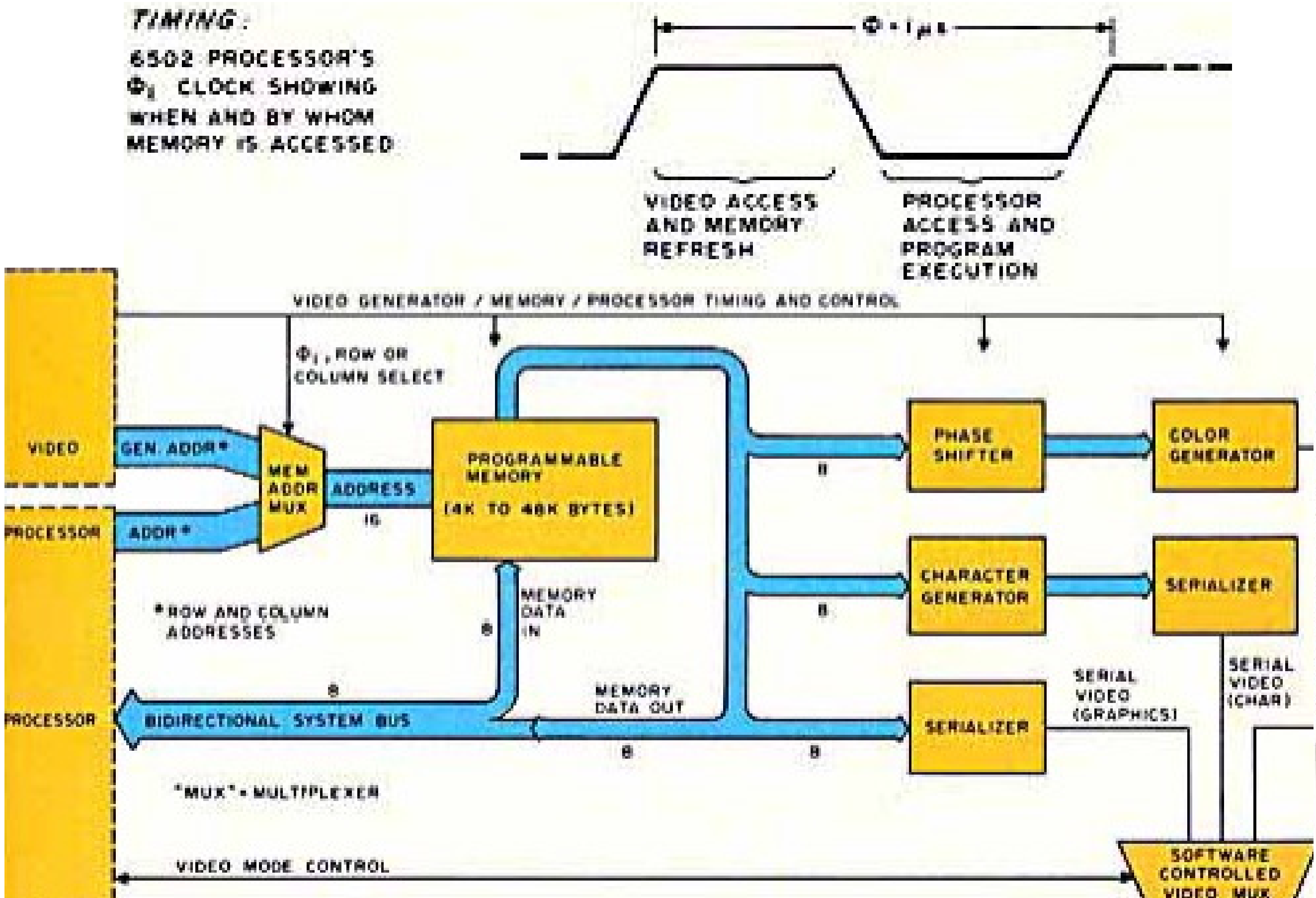
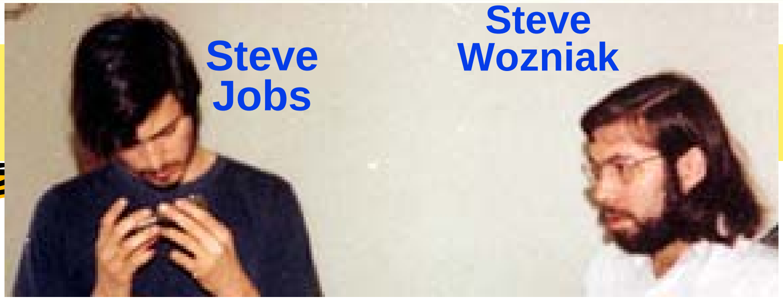
5008: Computer Architecture

Appendix C – Review of Memmory Hierarchy

1977: DRAM faster than microprocessors

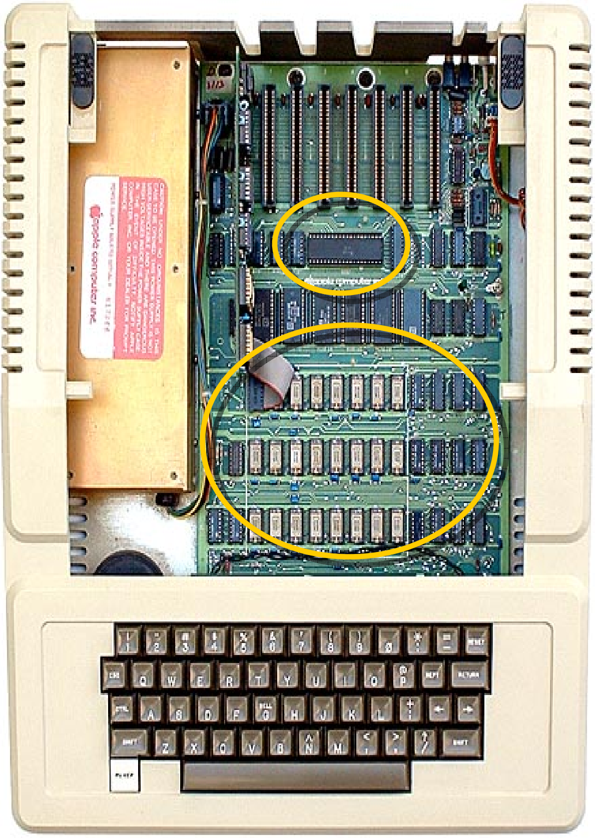




Apple][(1977)

CPU: 1000 ns

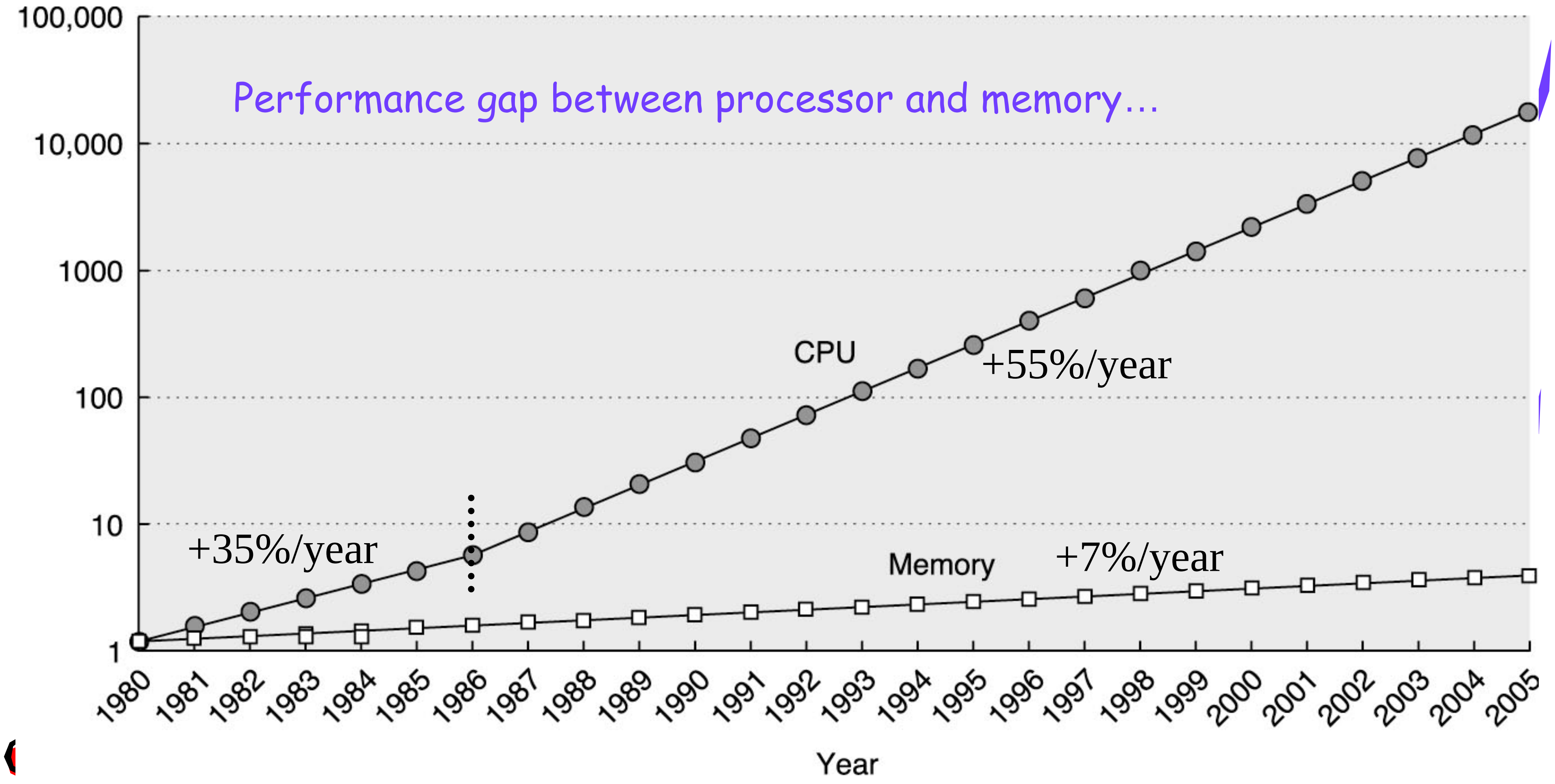
DRAM: 400 ns



RAM Complement	Apple II System
4K	$ 1,298.00
48K	2,638.00

CPU vs. Memory Performance Trends

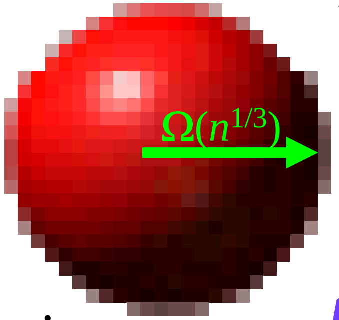
Relative performance (vs. 1980 perf.) as a function of year



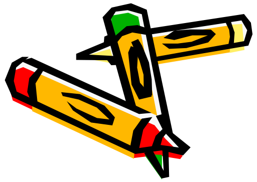
Why?

- Because of a fundamental constraint:
  - The larger the memory, the higher the access latency.

(Beyond some point.)



  - A characteristic of all present memory technologies.
- This will remain true in *all* future technologies!
  - Quantum mechanics gives a minimum size for bits
    - (Assuming energy density is limited.)
  - Thus *n* bits require $\Omega(n)$ volume of space.
  - At light speed, random access takes $\Omega(n^{1/3})$ time!
    - (Assuming a roughly flat region of space time.)
    - Of course, specific memory technologies (or a suite of available technologies) may scale even worse than this!



What Programmers Want

- Programmers like to be insulated from physics
  - It's easier to think about programming models if you don't have to worry about physical constraints.
  - However, ignoring physics in algorithm design always sacrifices some runtime efficiency.
  - But, programmer productivity is more important economically than performance (for now).
- Programmers want to pretend they have the following memory model:
  - An unlimited number of memory locations, all accessible instantly!

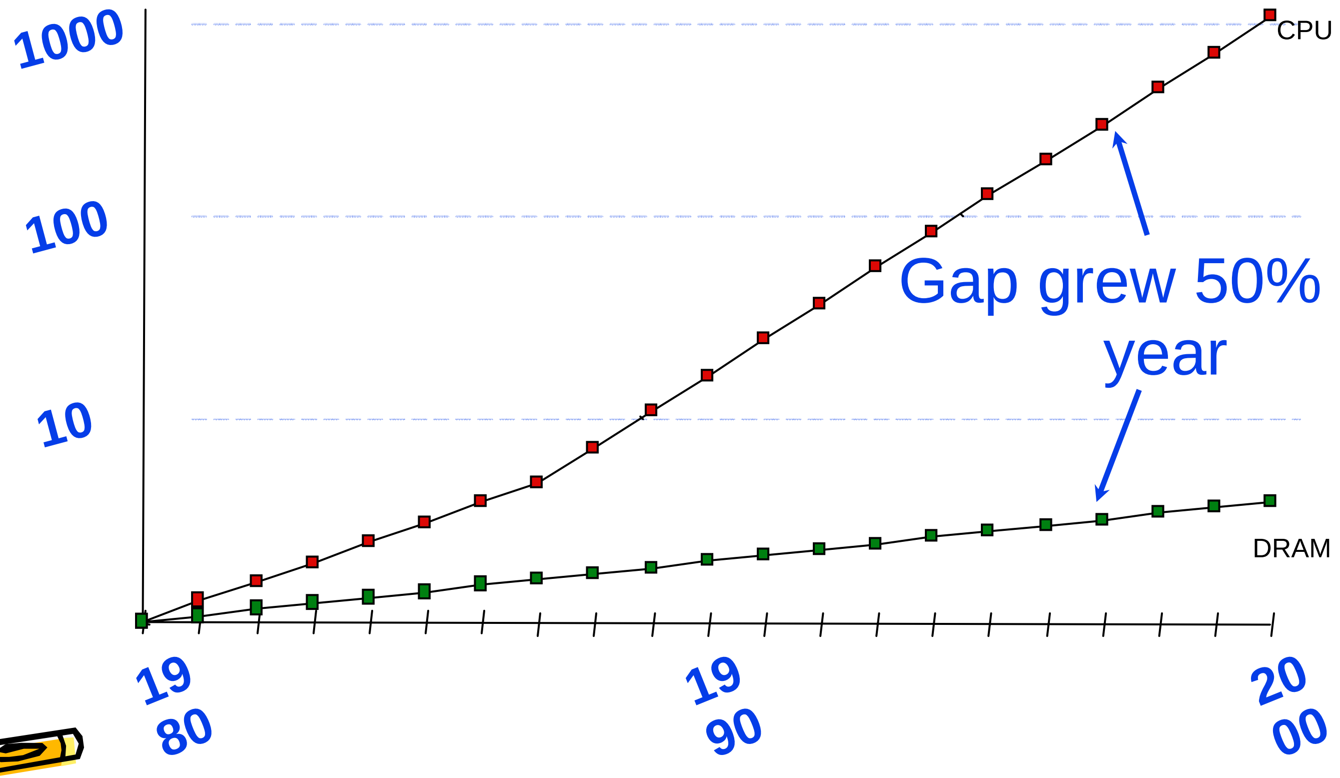
What We Can Provide?

- A small number of memory locations, all accessible quickly; and/or
- A large number of memory locations, all accessible more slowly; and/or
- A memory hierarchy,
  - Has both kinds of memory (& maybe others)
  - Can automatically transfer data between them
    - often (hopefully) before it's needed
  - Approximates (gives the illusion of having):
    - As many locations as the large memory has,
    - All accessible almost as quickly as the small memory!

Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?

A. Put smaller, faster "cache" memories between CPU and DRAM. Create a "memory hierarchy".



Gap grew 50% per year

CPU
60% per yr
2X in 1.5 yrs

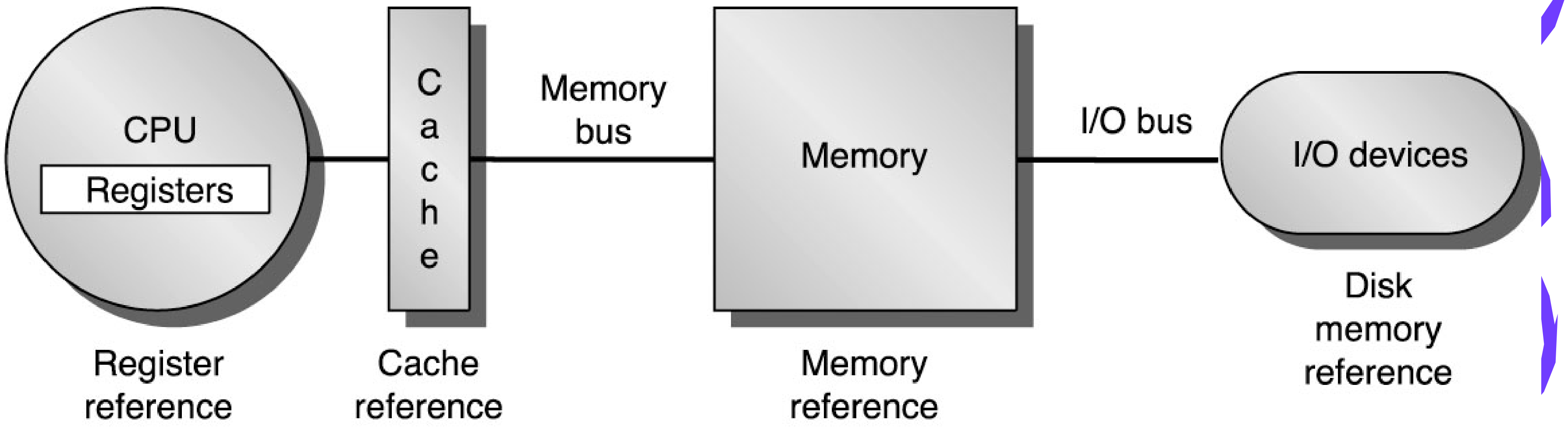
DRAM
9% per yr
2X in 10 yrs

Big & Fast: How is that possible?

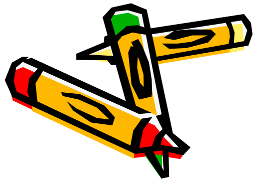
- Illusion of big & fast is enabled by pre-fetching of data from remote locations (before needed).
- How can simple, dumb hardware figure out what remote data to automatically fetch, before we need it?
- Basis for this: Principle of Locality
  - When a location is accessed, it *and* "nearby" locations are likely to be accessed again soon.
    - "Temporal" locality - Same location likely again soon.
    - "Spatial" locality - Nearby locations likely soon.
- This only works if the programmer cooperates!

Simple Hierarchy Example

- Note many orders of magnitude change in characteristics between levels:



Size:	500 bytes	×128→	64 KB	×8192→	512 MB	×200→	100 GB
Speed:	0.25 ns	×4 →	1 ns	×100 →	100 ns	×50,000 →	5 ms
(for random access)							



Cache Basics

- A cache is a (usually automatically managed) store that is intermediate in size, speed, and cost-per-bit between the programmer-visible registers (usually SRAM) and main physical memory (usually DRAM).
- The cache itself may be SRAM or fast DRAM.
- There may be >1 levels of caches; for now we'll focus on a single level.

Levels of the Memory Hierarchy

Capacity
Access Time
Cost

CPU Registers
100s Bytes
<10s ns

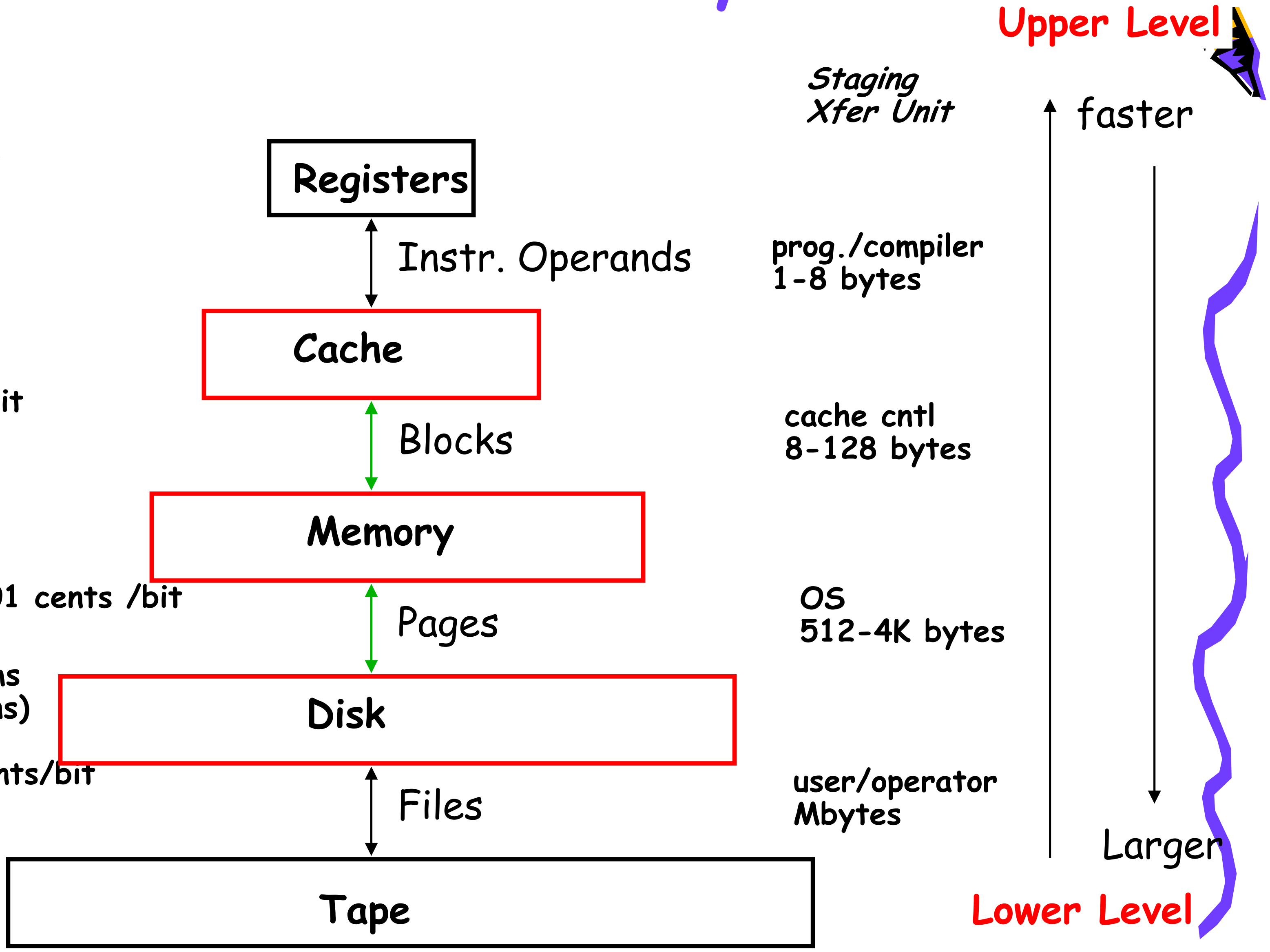
Cache
K Bytes
10-100 ns
1-0.1 cents/bit

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}



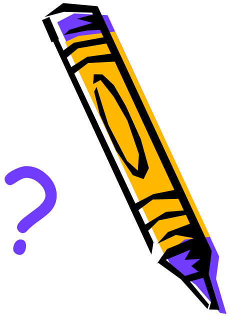
Staging
Xfer Unit

prog./compiler
1-8 bytes

cache cntl
8-128 bytes

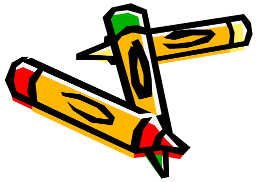
OS
512-4K bytes

user/operator
Mbytes



Programmer Control, or Not?

- Even if the programmer isn’t afraid of the physics, hierarchies are still useful. Why?
  - The lower (larger, slower) levels are cheaper per bit.
  - Generalized Amdahl’s Law demands we include memory at different cost levels in our design.
- But: Automatic movement between hierarchy levels can’t always give you the best algorithm!
- The ideal memory hierarchy would…
  - Allow (but not require) programmer control of data movement at all levels.
    - the ambitious & skilled programmer could then better optimize how the hardware is used.



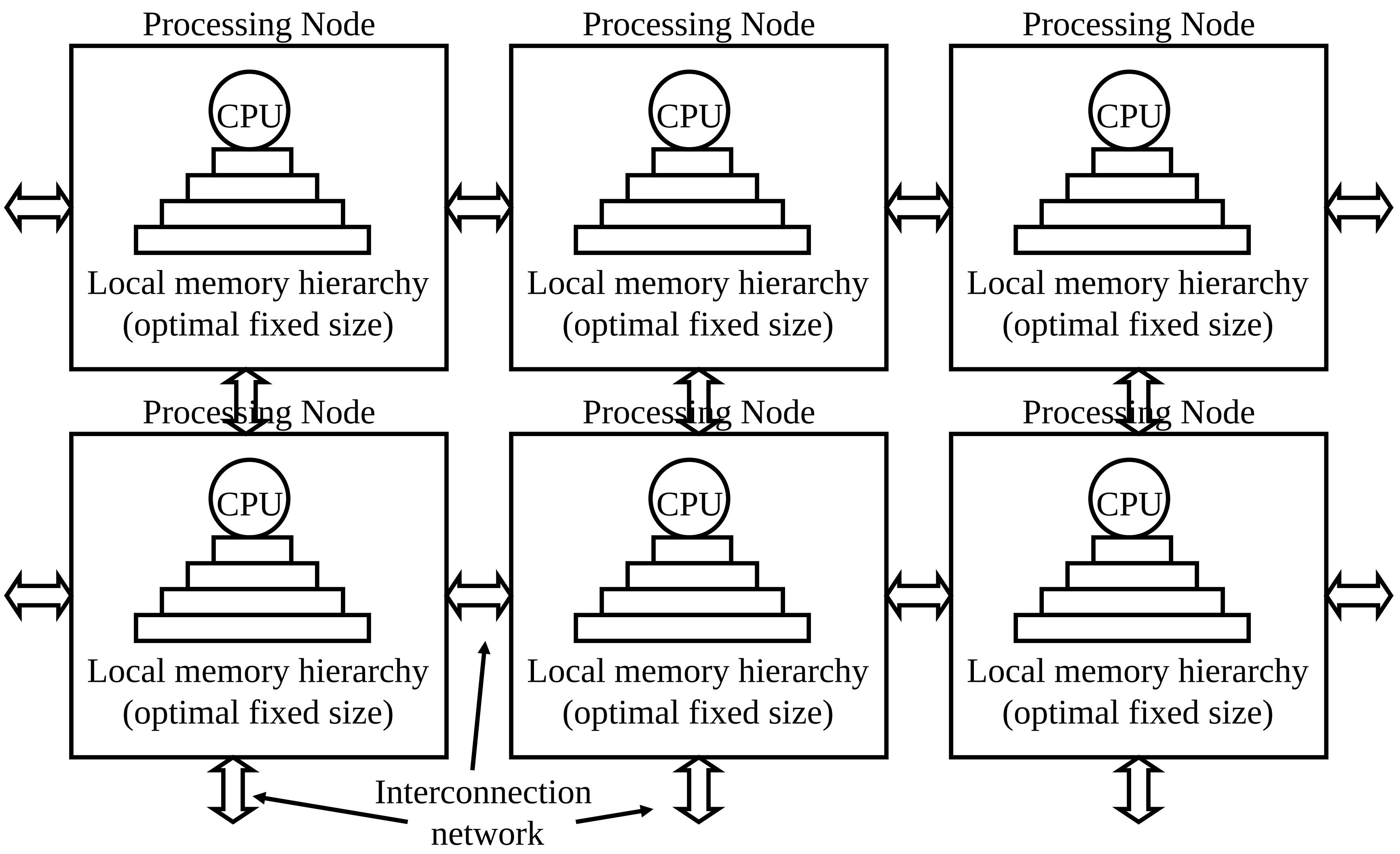
Memory Scaling Isn't Everything

- The goal of memory hierarchy:
  - Scale available memory to arbitrarily large sizes
  - without a big impact on access time.
- But should scaling memory PER CPU really be the goal in systems design? No!
- Consider:
  - The cost/size/power of a sufficiently large memory, network file server, or tape archive system can easily far exceed that of a CPU.
- A balanced system design should *also* scale the number of CPUs with the amount of memory!

Memory Architectures

- As your problem size increases, here’s what you should do, as a systems engineer:
  - Add memory to your CPU (at the appropriate hierarchy level for best overall performance/cost)
    - Up until the size/cost/power/latency due to memory access starts to dominate.
  - Then, rather than keep throwing more hierarchy onto that one poor CPU,
    - Add another CPU to the system, and build another memory hierarchy around it.
    - CPUs can be networked

One Scalable Architecture



Advantages

- Mirrors the architecture of physics
- Every item in memory has a CPU relatively nearby that can process it.
- To manipulate some remote data in a certain way, the code to do the manipulation can be moved to the data, rather than vice-versa.
  - This can be much cheaper, as data sizes grow!
- Every item in memory can still be accessed, if needed, by any CPU, via sharing
  - Through interconnection-network, message-passing.
  - Memory accessible by a CPU still scales arbitrarily.

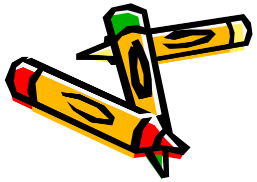
Outline

- Review
- Cache
- Cache Performance
- 6 Basic Cache Optimization
- Virtual Memory

The Principle of Locality

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon
    (e.g., straightline code, array access)
- Last 15 years, HW relied on locality for speed

It is a property of programs which is exploited in machine design.

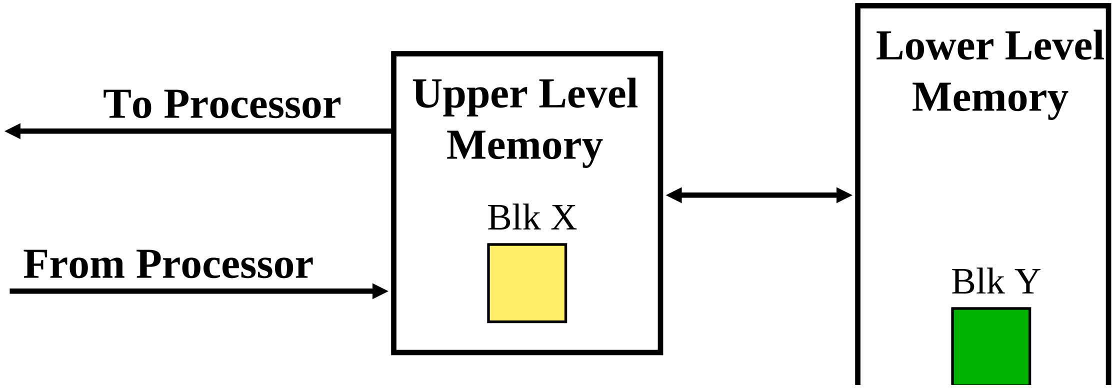


Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of

    RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +

    Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



Cache Measures

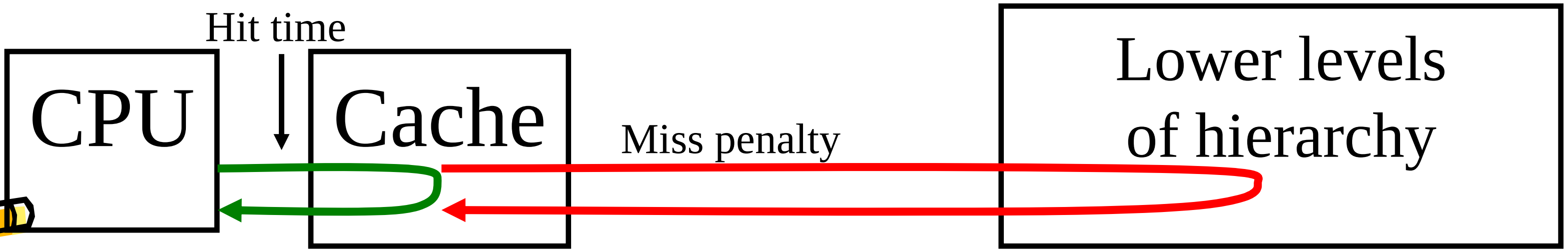
- *Hit rate*: fraction found in that level
  - So high that usually talk about *Miss rate*
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- Average memory-access time
  = Hit time + Miss rate x Miss penalty (ns or clocks)
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    = f(latency to lower level)
  - *transfer time*: time to transfer block
    =f(BW between upper & lower levels)

Cache Performance Formulas

(Average memory access time) =
(Hit time) + (Miss rate)×(Miss penalty)

$$\overline{T_{acc}} = T_{hit} + f_{miss} T_{+miss}$$

- The times T_{acc}, T_{hit}, and T_{+miss} can be all either:
  - Real time (*e.g.*, nanoseconds)
  - Or, number of clock cycles
    - In contexts where cycle time is known to be a constant
- Important:
  - T_{+miss} means the extra (not total) time for a miss
    - in *addition* to T_{hit}, which is incurred by all accesses



Four Questions for Memory Hierarchy

- Consider any level in a memory hierarchy.
  - Remember a block is the unit of data transfer.
    - Between the given level, and the levels below it
- The level design is described by four behaviors:
  - Block Placement:
    - Where could a new block be placed in the level?
  - Block Identification:
    - How is a block found if it is in the level?
  - Block Replacement:
    - Which existing block should be replaced if necessary?
  - Write Strategy:
    - How are writes to the block handled?

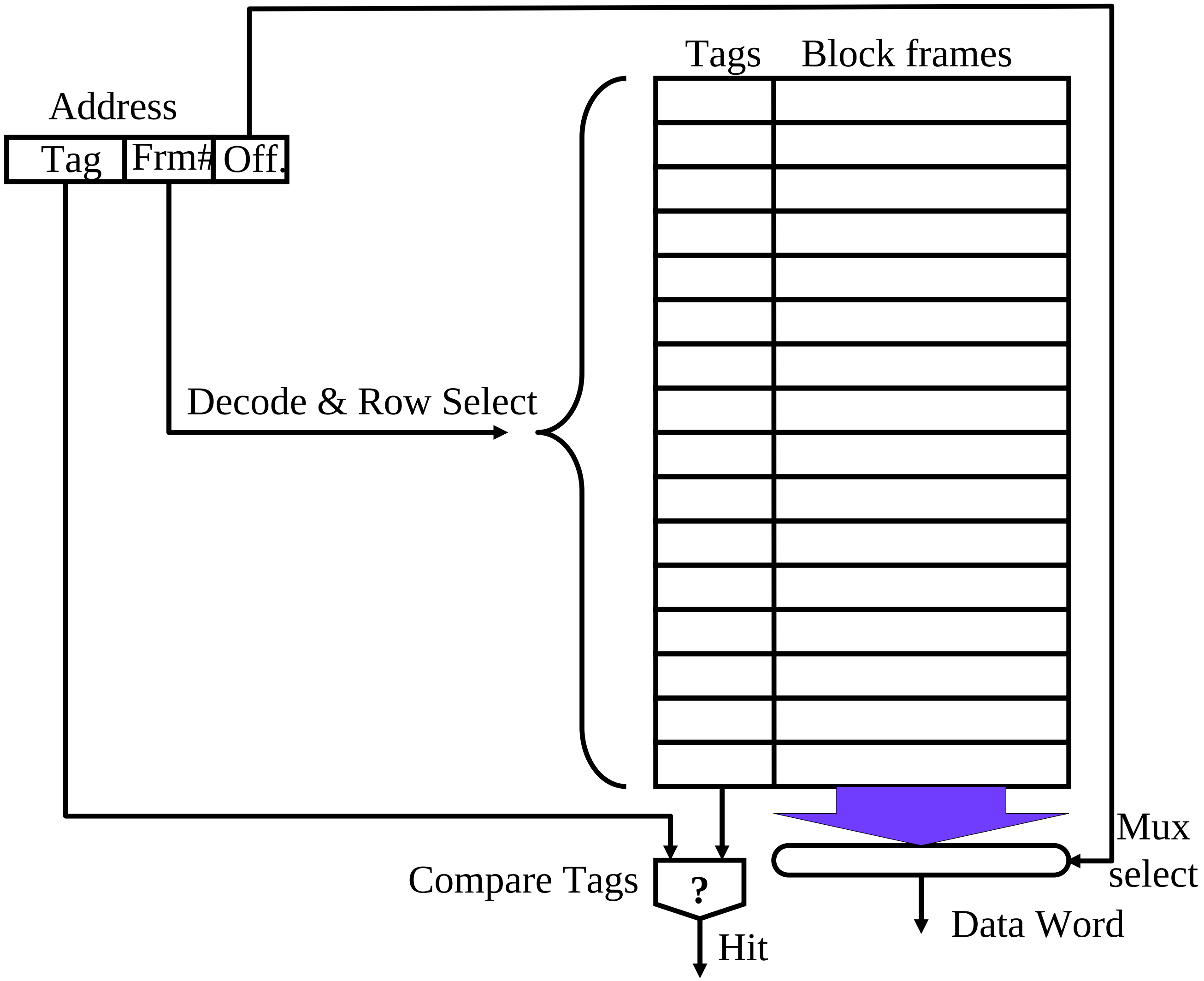
Direct-Mapped Placement

- A block can only go into one frame in the cache
  - Determined by block's address (in memory space)
    - Frame number usually given by some low-order bits of block address.
- This can also be expressed as:

(Frame number) =
(Block address) **mod** (Number of frames in cache)

- Note that in a direct-mapped cache,
  - block placement & replacement are both completely determined by the address of the new block that is to be accessed.

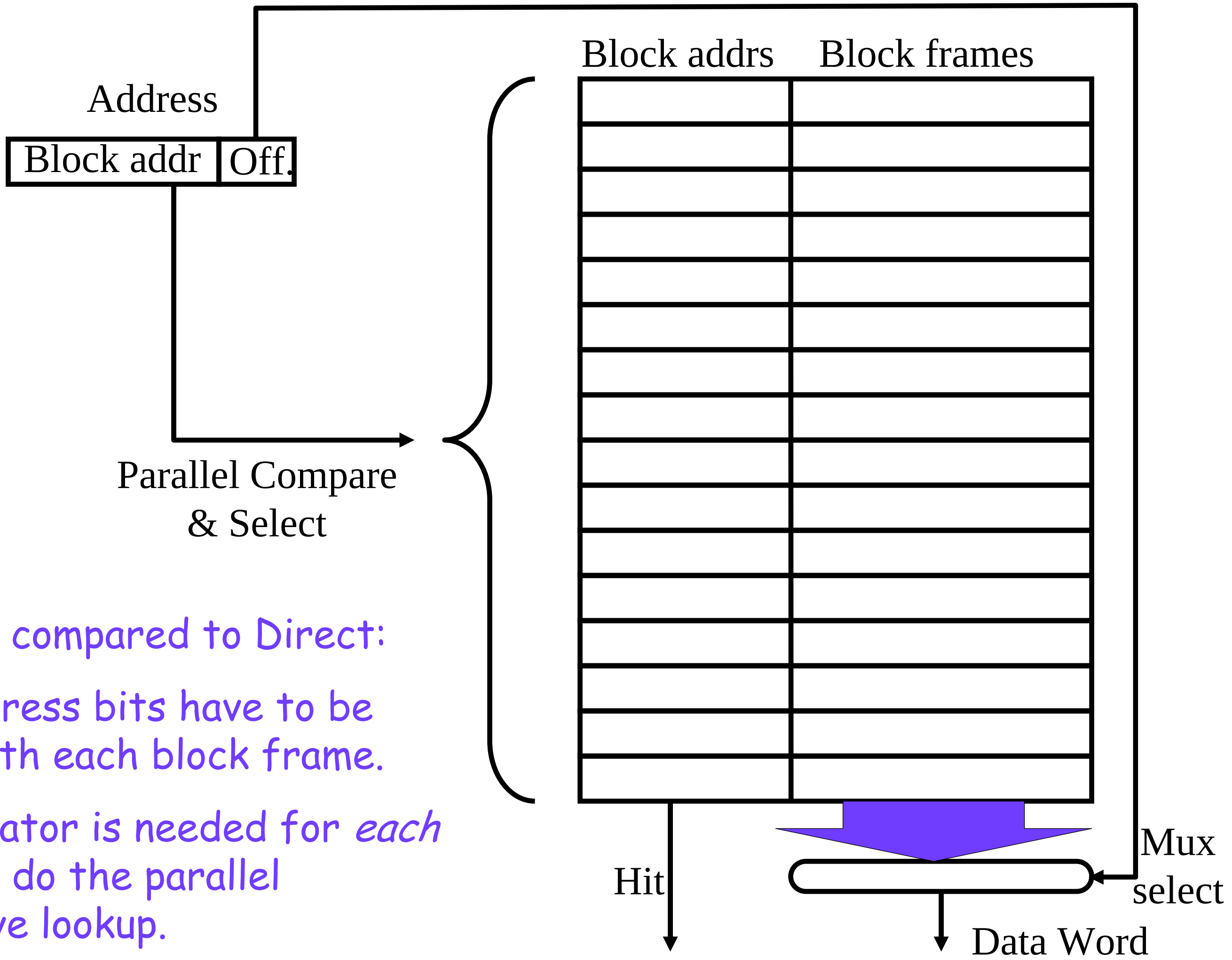
Direct-Mapped Identification



Fully-Associative Placement

- One alternative to direct-mapped is…
  - Allow block to fill any empty frame in the cache.
- How do we then locate the block later?
  - Can associate each stored block with a tag
    - Identifies the block's home address.
  - When the block is needed, we can use the cache as an associative memory, using the tag to match all frames in parallel, to pull out the appropriate block.
- Another alternative to direct-mapped is placement under full program control.
  - A register file can be viewed as a small programmer-controlled cache (w. 1-word blocks).

Fully-Associative Identification



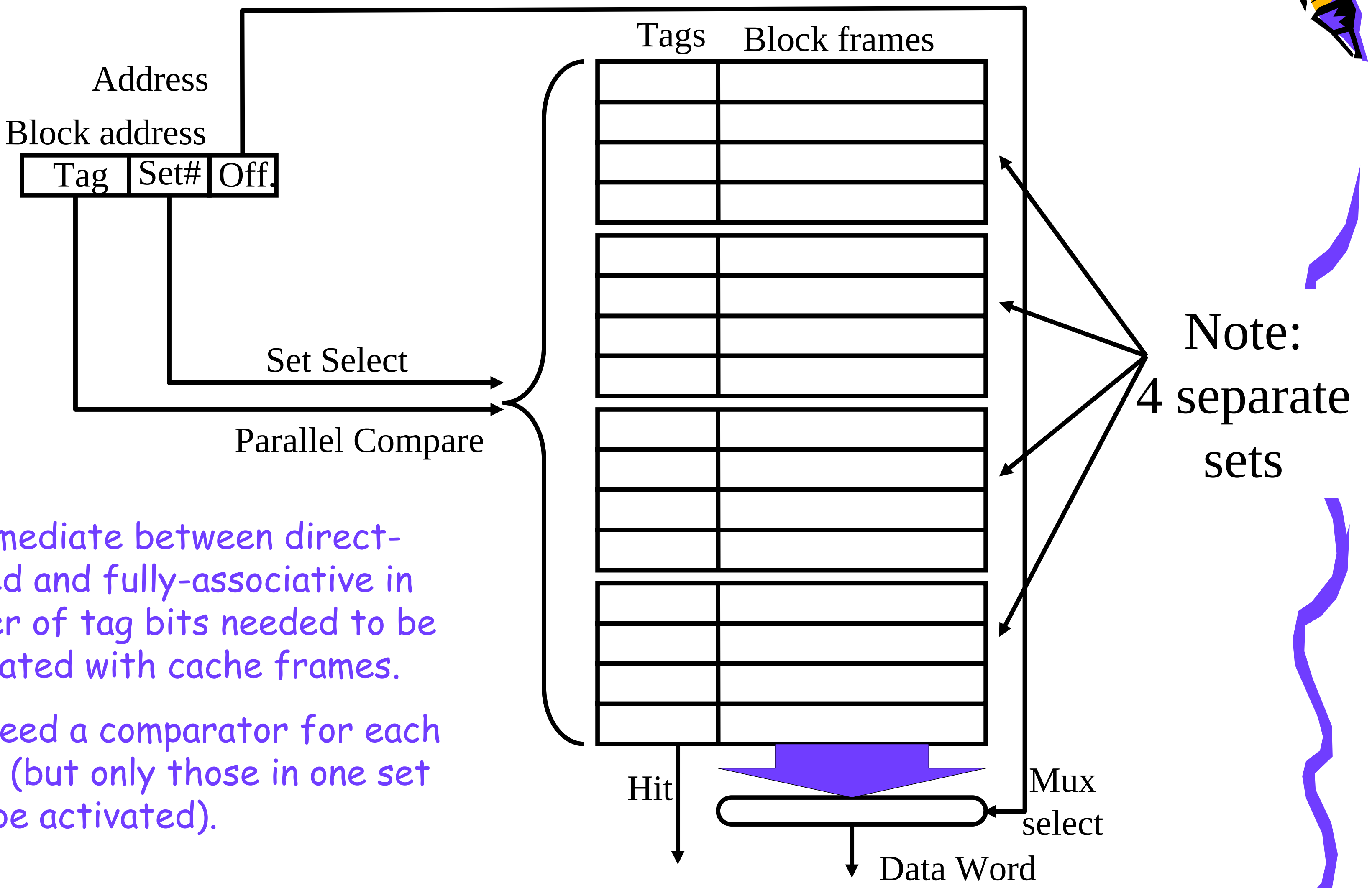
Note that, compared to Direct:

- More address bits have to be stored with each block frame.
- A comparator is needed for *each* frame, to do the parallel associative lookup.

Set-Associative Placement

- The block address determines not a single frame, but a frame set (several frames, grouped together).
  - (Frame set #) = (Block address) **mod** (# of frame sets)
- The block can be placed associatively anywhere within that frame set.
- If there are *n* frames in each frame set, the scheme is called "*n*-way set-associative".
- Direct mapped = 1-way set-associative.
- Fully associative: There is only 1 frame set.

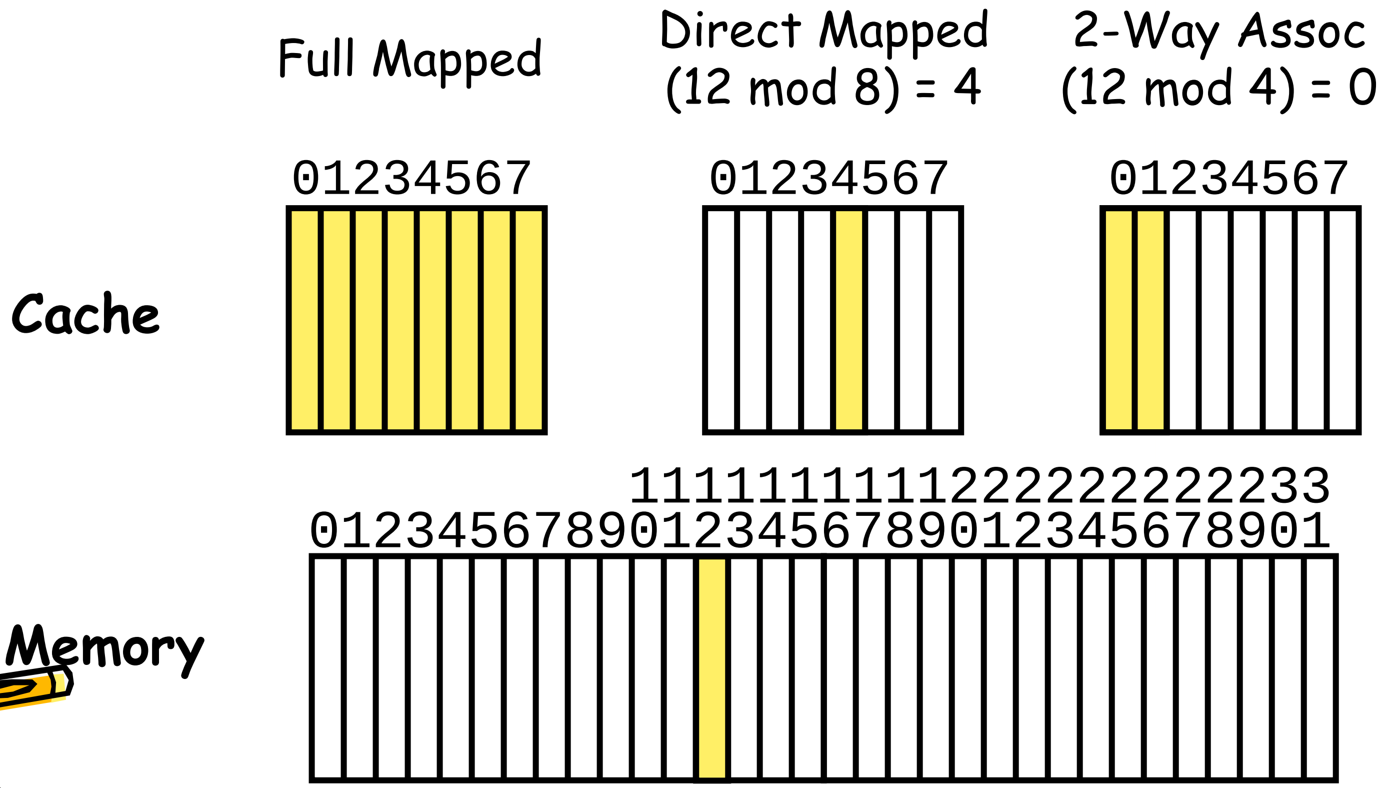
Set-Associative Identification



- Intermediate between direct-mapped and fully-associative in number of tag bits needed to be associated with cache frames.
- Still need a comparator for each frame (but only those in one set need be activated).

Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets



Full Mapped

Direct Mapped
(12 mod 8) = 4

2-Way Assoc
(12 mod 4) = 0

Cache Size Equation

- Simple equation for the size of a cache:

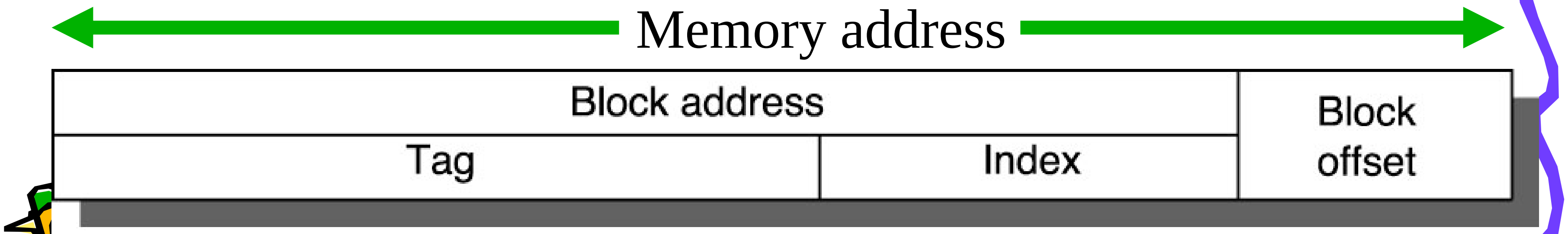
  (Cache size) = (Block size) × (Number of sets) × (Set Associativity)

- Can relate to the size of various address fields:

  (Block size) = $2^{(\#\text{ of offset bits})}$

  (Number of sets) = $2^{(\#\text{ of index bits})}$

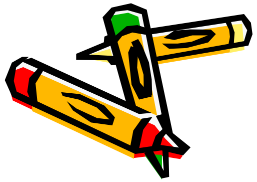
  (# of tag bits) = (# of memory address bits) – (# of index bits) – (# of offset bits)



Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

<table>
<tr><th colspan="2">Block Address</th><th rowspan="2">Block Offset</th></tr>
<tr><th>Tag</th><th>Index</th></tr>
</table>



Q3: Replacement Strategies

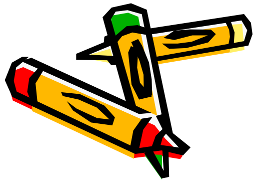
- Which existing block do we replace, when a new block comes in?
- Direct-mapped:
  - There's only one choice!
- Associative (fully- or set-):
  - If any frame in the set is empty, pick one of those.
  - Otherwise, there are many possible strategies:
    - (Pseudo-) random
      - Simple, fast, and fairly effective
    - Least-recently used (LRU), and approximations thereof
      - Makes little difference in larger caches
    - First in, first out (FIFO)
      - Use time since block was read
      - May be easier to track than time since last access

Write Strategies

- Most accesses are reads, not writes
  - Especially if instruction reads are included
- Writes are more difficult
  - Can't write to cache till we *know* the right block
  - Object written may have various sizes (1-8 bytes)
- When to synchronize cache with memory?
  - Write through - Write to cache & to memory
    - Prone to stalls due to high bandwidth requirements
  - Write back - Write to memory upon replacement
    - Memory may be out of date

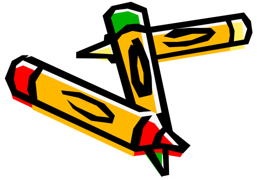
Q4: What happens on a write?

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No



Write Miss Strategies

- What do we do on a write to a block that’s not in the cache?
- Two main strategies:
  - Write-allocate (fetch on write) - Cache the block.
  - No write-allocate (write around) - Just write to memory.
- Write-back caches tend to use write-allocate.
- White-through tends to use no write-allocate.



Dirty Bits

- Useful in write-back strategies to minimize memory bandwidth
- When a cache block is modified, it is marked as “dirty” (no longer a pristine copy of memory)
- When a block is replaced, it only needs to be written back to memory if it is dirty

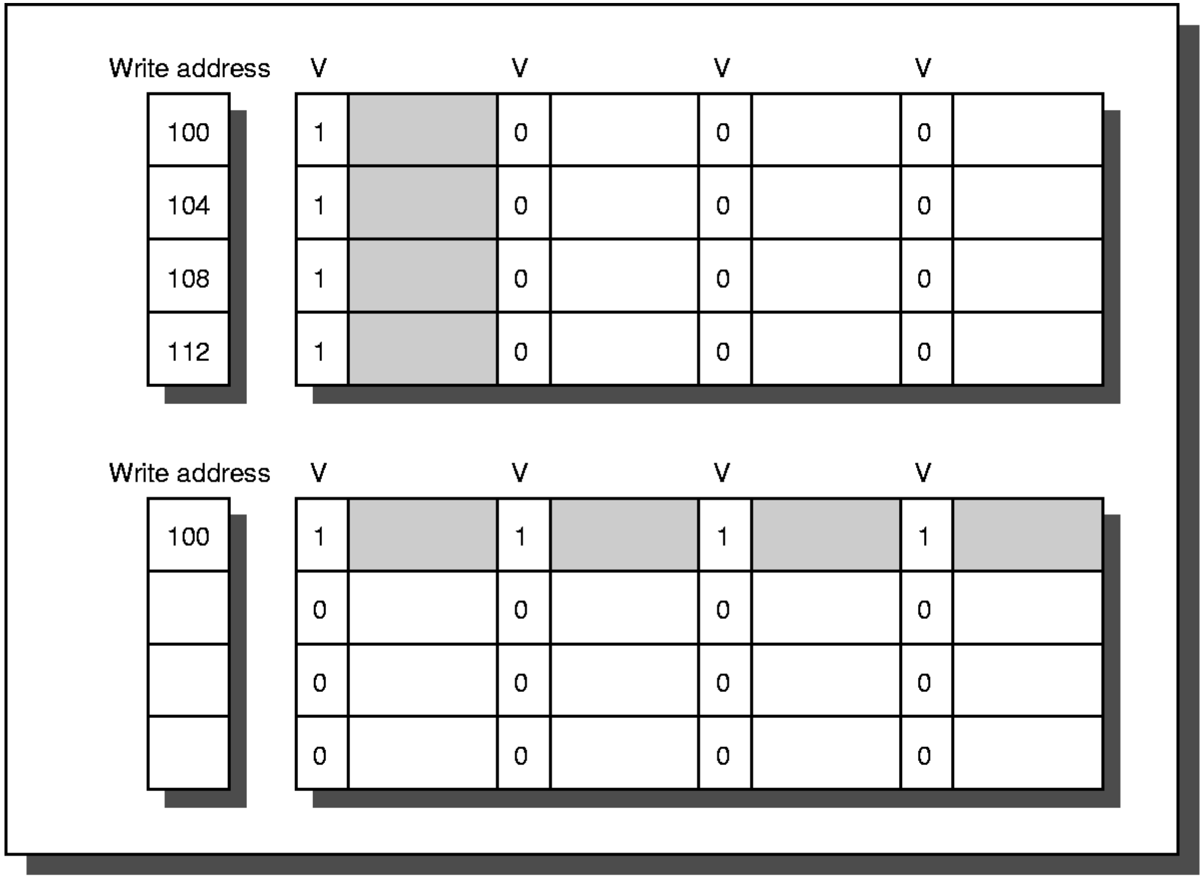
Another Write Strategy

- Maintain a FIFO queue of all dirty cache frames (*e.g.* can use a doubly-linked list)
  - When a block is written to, put it at the end of the queue, if it's not already dirty
  - When a dirty block is evicted, remove it from the queue, and write it immediately
- Meanwhile, take items from top of queue and write them to memory as fast as bus can handle
  - Reads might take priority, or have a separate bus
- Advantages: Write stalls are minimized, while keeping memory as up-to-date as possible

Write Buffers

- A mechanism to help reduce write stalls
- On a write to memory, block address and data to be written are placed in a write buffer.
- CPU can continue immediately
  - Unless the write buffer is full.
- Write merging:
  - If the same block is written again before it has been flushed to memory, old contents are replaced with new contents.

Write Merging Example



Write address	V		V		V		V	
100	1	▨	0		0		0	
104	1	▨	0		0		0	
108	1	▨	0		0		0	
112	1	▨	0		0		0	

Write address	V		V		V		V	
100	1	▨	1	▨	1	▨	1	▨
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

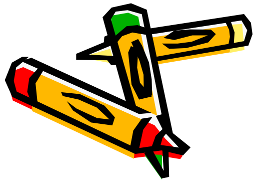
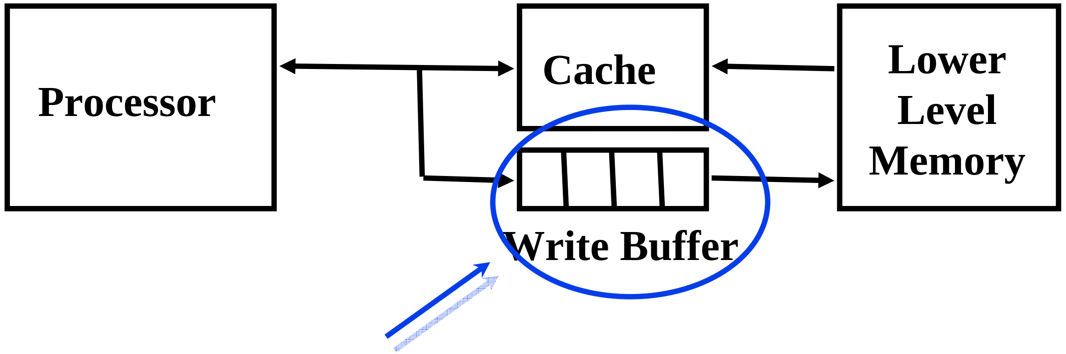


FIGURE To illustrate write merging, the write buffer on top does not use it while the write buffer on the bottom does.

Write Buffers for Write-Through Caches



Holds data awaiting write-through to lower level memory

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Outline

- Review
- Cache
- Cache Performance
- 6 Basic Cache Optimization
- Virtual Memory

Cache Performance Review

- From chapter 1:
  - (Memory stall cycles) =

    (Instruction count) ×
    (Accesses per instruction) × (Miss rate) ×
    (Miss penalty)

- A better metric:
  - (Average memory access time) =

    (Hit time) + (Miss rate) × (Miss penalty)

More Cache Performance Metrics

- Can split access time into instructions & data:

  Avg. mem. acc. time = Hit time + miss rate × miss penalty =
  (% instruction accesses) × (inst. mem. access time) +
  (% data accesses) × (data mem. access time)

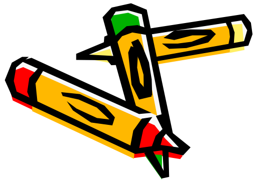
- Another formula from chapter 1:

  CPU time = (CPU execution clock cycles + Memory stall clock cycles) × cycle time

  – Useful for exploring ISA changes

- Can break stalls into reads and writes:

  Memory stall cycles =
  (Reads × read miss rate × read miss penalty) +
  (Writes × write miss rate × write miss penalty)



Factoring out Instruction Count

- Gives (lumping together reads & writes):

$$CPU\ time = IC \times Clock\ cycle\ time \times \left(CPI_{exec} + \frac{Accesses}{Inst} \times Miss\ rate \times Miss\ penalty \right)$$

- May replace:

$$\frac{Accesses}{instruction} \times Miss\ rate \rightarrow \frac{Misses}{instruction}$$

  - So that miss rates aren’t affected by redundant accesses to same location within an instruction.

Improving Cache Performance

- Reducing cache miss Penalty
- Reducing miss rate
- Reducing hit time

- Note that by Amdahl's Law, there will be diminishing returns from reducing only hit time or amortized miss penalty by itself, instead of both together

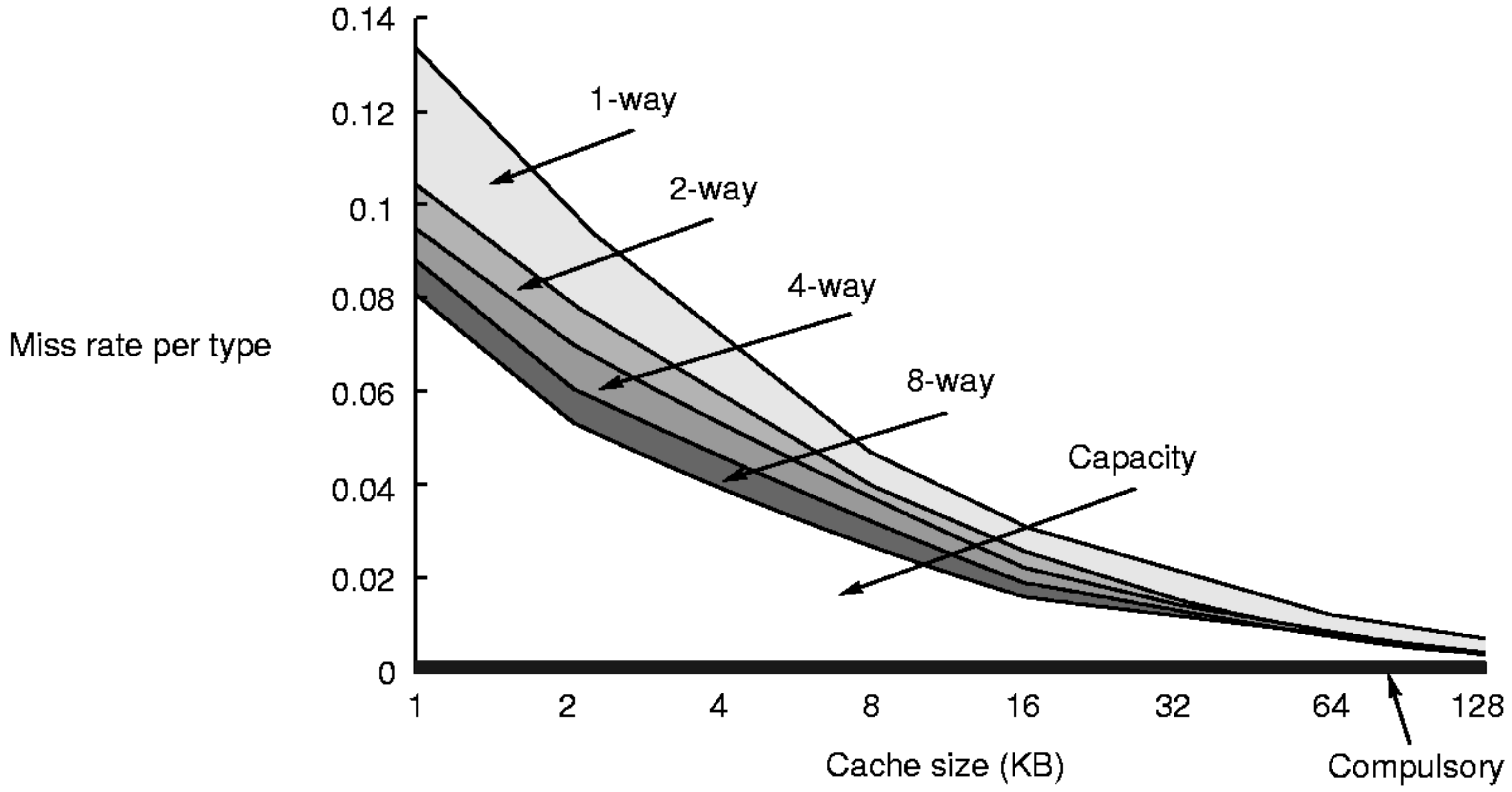
Three Types of Misses

- Compulsory
  - During a program, the very first access to a block will not be in the cache (unless pre-fetched)
- Capacity
  - The working set of blocks accessed by the program is too large to fit in the cache
- Conflict
  - Unless cache is fully associative, sometimes blocks may be evicted too early (compared to fully-associative) because too many frequently-accessed blocks map to the same limited set of frames.

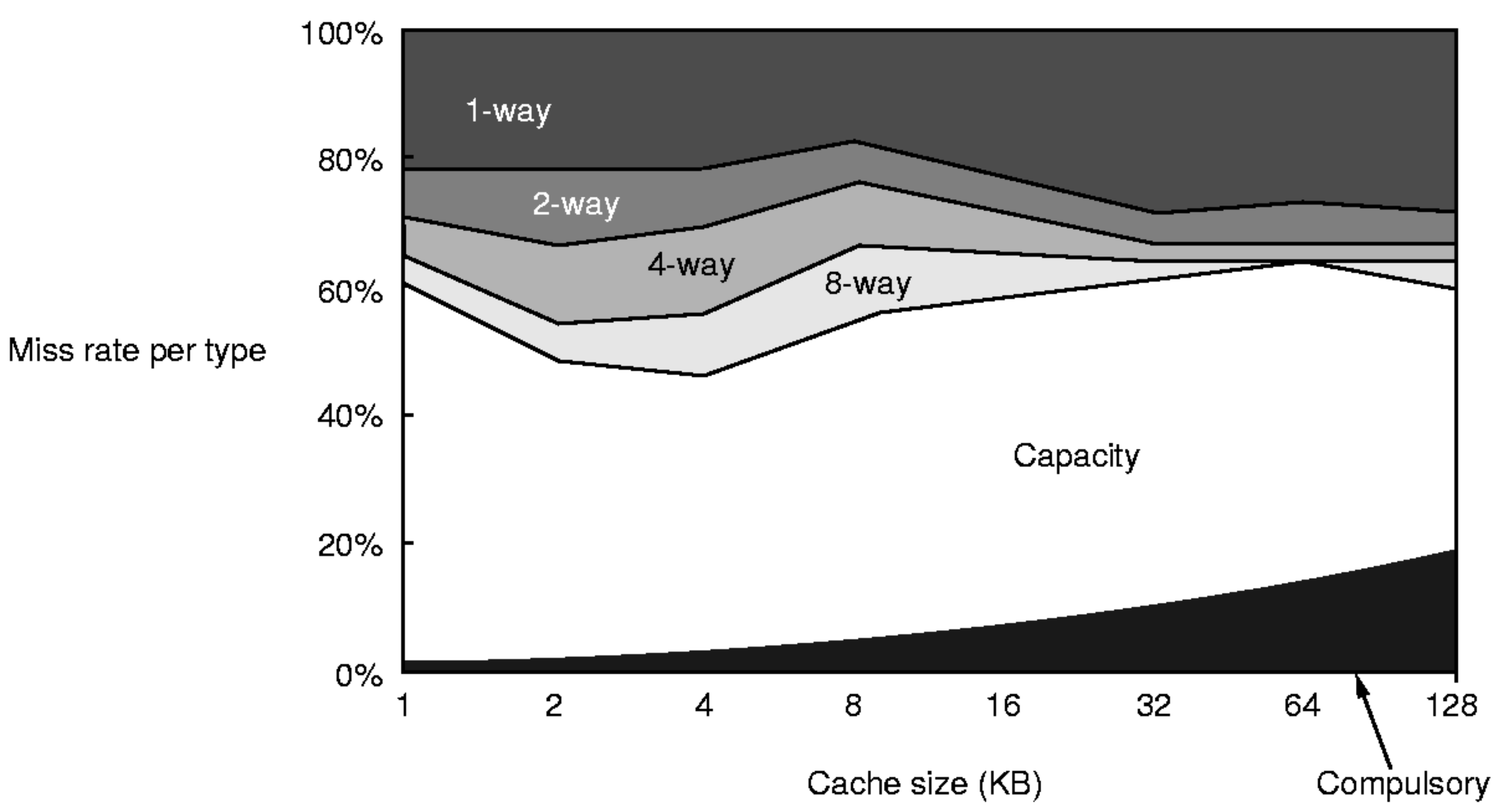
Miss Statistics

- See Fig. C.8 (too large to fit legibly on slide)
- Note:
  - Conflict misses are a significant fraction of total misses in a direct-mapped cache.
  - Going from direct-mapped to 2-way helps almost as much as doubling cache size.
  - Going from direct-mapped to 4-way is *better* than doubling cache size.

Misses by Type



As fraction of total misses



Outline

- Review
- Cache
- Cache Performance
- 6 Basic Cache Optimization
- Virtual Memory

6 Basic Cache Optimizations

- Reducing Miss Rate

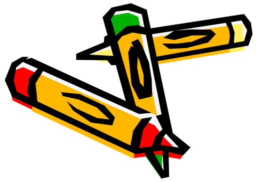
1. Larger Block size (compulsory misses)
2. Larger Cache size (capacity misses)
3. Higher Associativity (conflict misses)

- Reducing Miss Penalty

4. Multilevel Caches
5. Giving Reads Priority over Writes

- Reducing hit time

6. Avoiding Address Translation during Indexing of the Cache

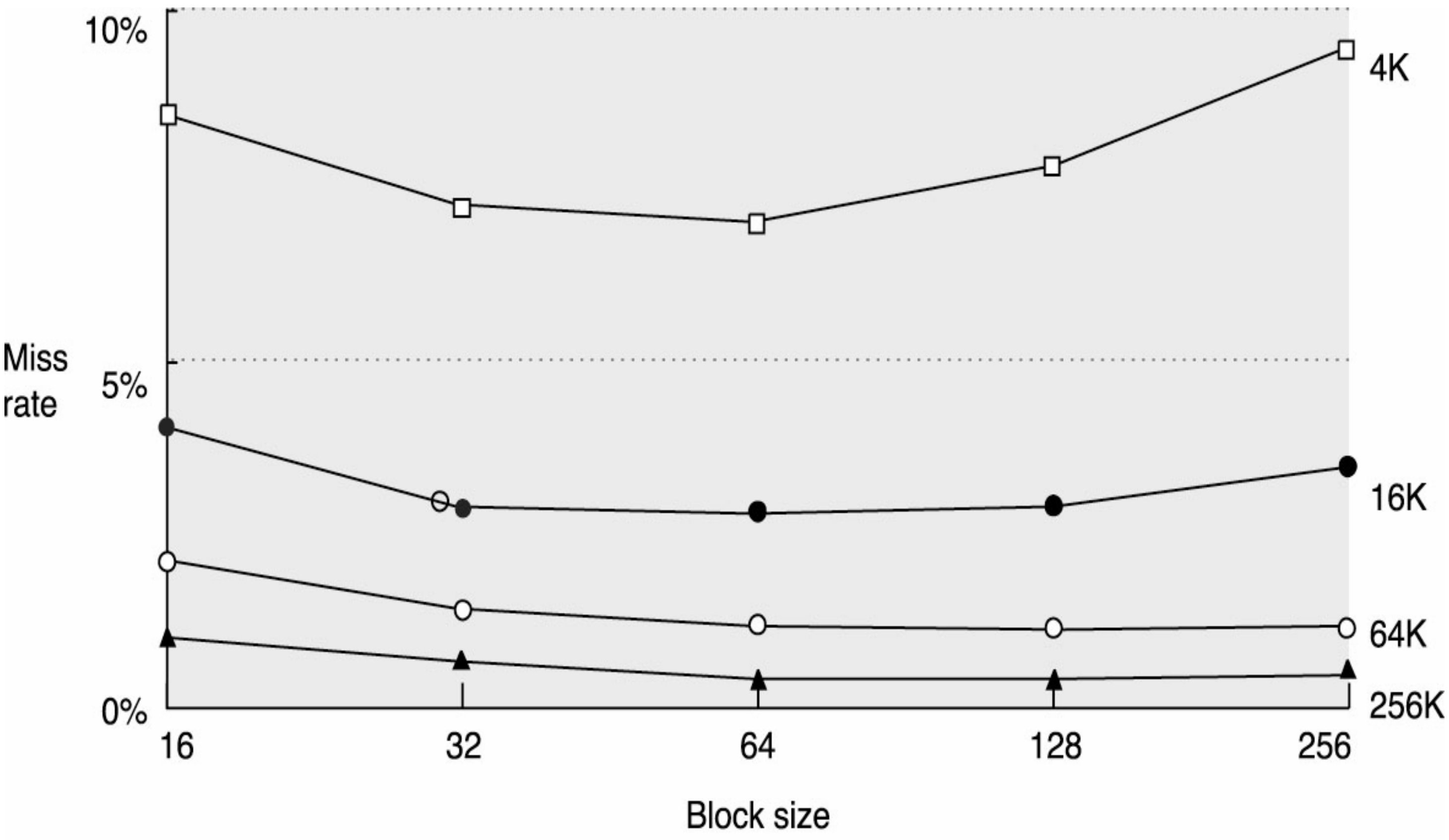


1. Larger Block Sizes

- Larger block size ➔ no. of blocks ↓
- Obvious advantages: reduce compulsory misses
  - Reason is due to spatial locality
- Obvious disadvantage
  - Higher miss penalty: larger block takes longer to move
  - May increase conflict misses and capacity miss if cache is small

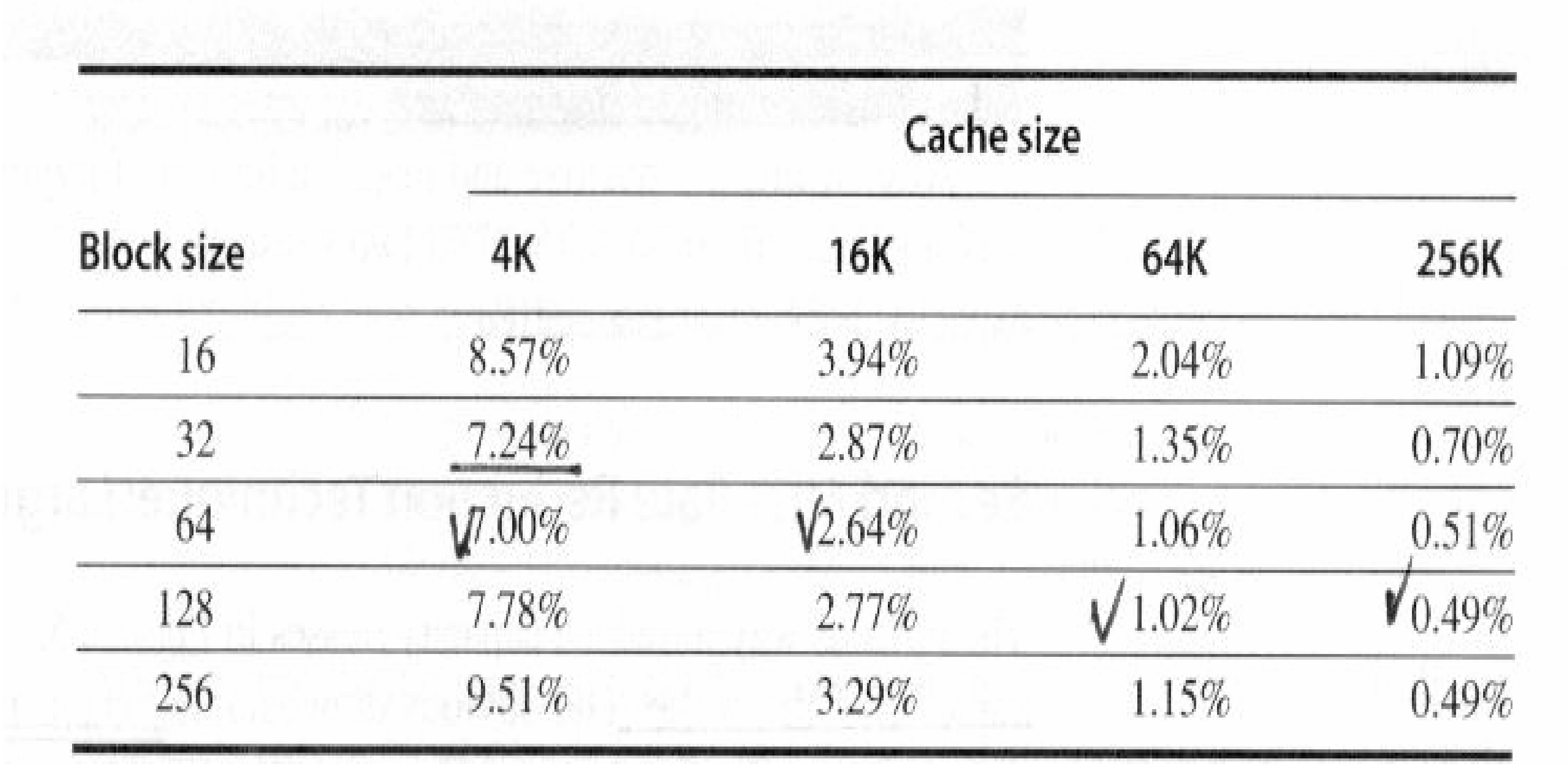
Don't let increase in miss penalty outweigh the decrease in miss rate

Miss Rate vs. Block Size



Larger block may increase conflict and capacity miss

Actual Miss Rate vs. Block Size

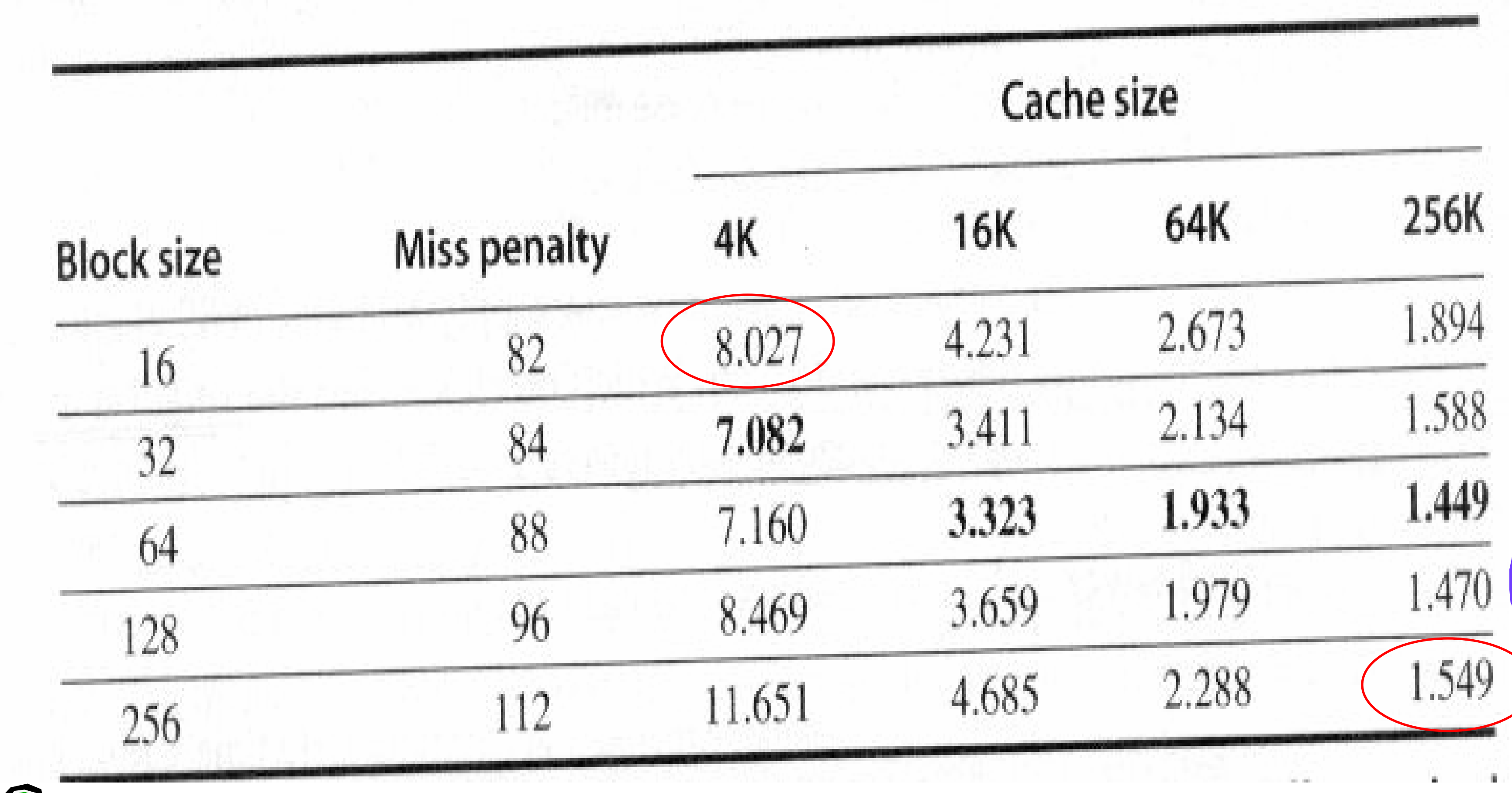


	Cache size			
Block size	4K	16K	64K	256K
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	✓7.00%	✓2.64%	1.06%	0.51%
128	7.78%	2.77%	✓1.02%	✓0.49%
256	9.51%	3.29%	1.15%	0.49%

Miss Rate VS. Miss Penalty

- Assume memory system takes 80 CC of overhead and then deliver 16 bytes every 2 CC. And the Hit time = 1 CC
- Miss penalty
  - Block size 16 = 80 + 2 = 82
  - Block size 32 = 80 + 2 * 2 = 84
  - Block size 256 = 80 + 16 * 2 = 112
- AMAT = hit_time + miss_rate*miss_penalty
  - 16-byte in a 4 KB cache = 1 + 8.57% * 82 = 8.027 CC
  - 256-byte in a 256 KB cache = 1 + 0.49% * 112 = 1.549 CC

AMAT VS. Block Size for Different-Size Caches



Block size	Miss penalty	Cache size			
		4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

2. Large Caches

- Cache size↑ ➔ miss rate↓; hit time↑
- Help with both conflict and capacity misses
- May need longer hit time AND/OR higher HW cost
- Popular in off-chip caches

3. Higher Associativity

- (Fig. C.8, C.9)
  - 8-way set associative is for practical purposes as effective in reducing misses as fully associative
- 2: 1 Cache rule of thumb
  - Miss rate: 2 way set associative of size N/ 2 is about the same as a direct mapped cache of size N (held for cache size < 128 KB)
- Greater associativity comes at the cost of increased hit time
  - Lengthen the clock cycle

4. Multi-Level Caches

- Probably the best miss-penalty reduction
- Performance measurement for 2-level caches
  - Average memory access time (AMAT) = Hit-time-L1 + Miss-rate-L1× Miss-penalty-L1
  - Miss-penalty-L1 = Hit-time-L2 + Miss-rate-L2 × Miss-penalty-L2
  - AMAT = Hit-time-L1 + Miss-rate-L1 × (Hit-time-L2 + Miss-rate-L2 × Miss-penalty-L2)

Multi-Level Caches (Cont.)

- Definitions:
  - Local miss rate: misses in this cache divided by the total number of memory accesses to this cache (Miss-rate-L2)
  - Global miss rate: misses in this cache divided by the total number of memory accesses generated by CPU (Miss-rate-L1 x Miss-rate-L2)
  - Global Miss Rate is what matters
- Advantages:
  - Capacity misses in L1 end up with a significant penalty reduction since they likely will get supplied from L2
    - No need to go to main memory
  - Conflict misses in L1 similarly will get supplied by L2

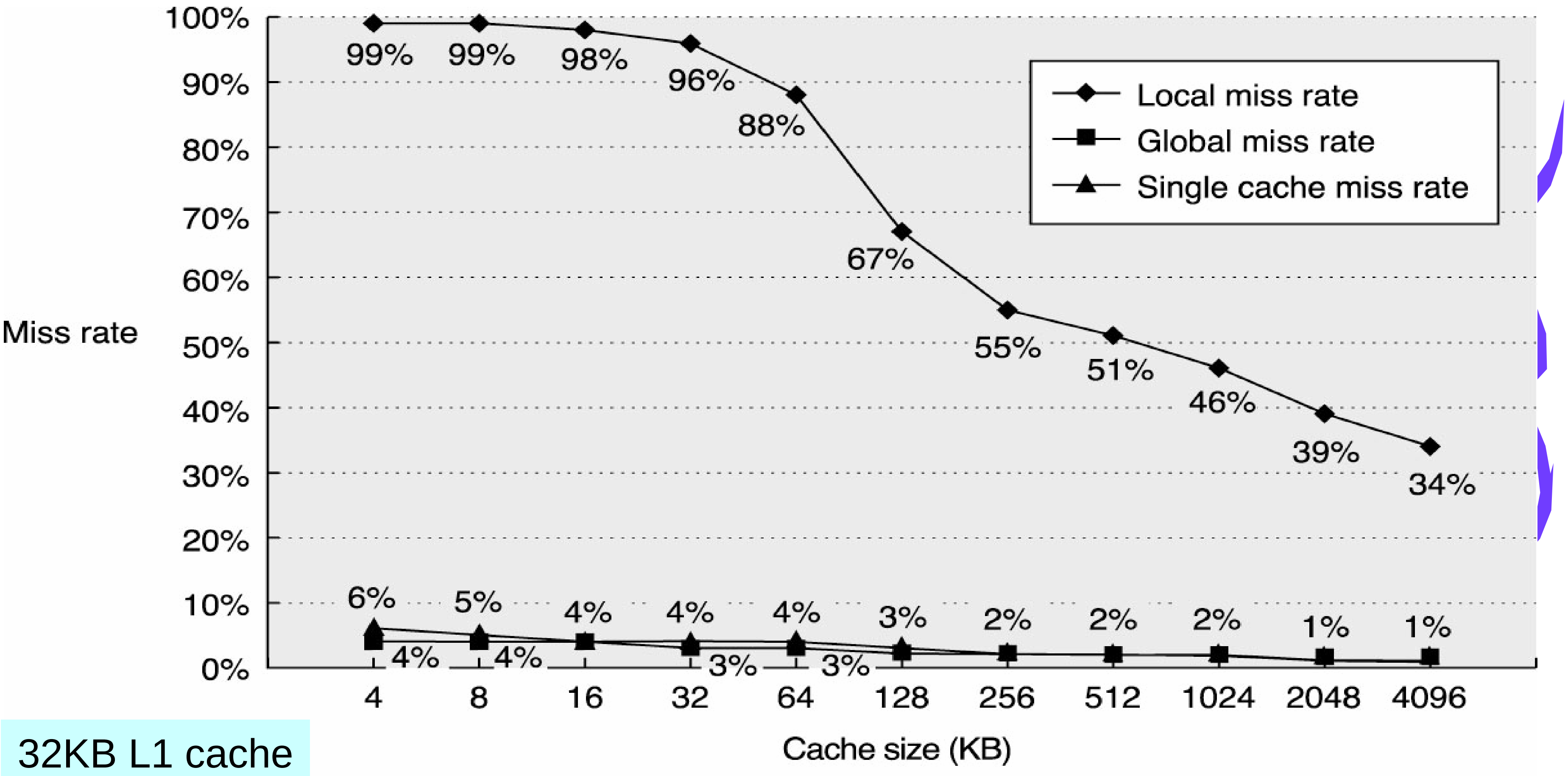
Effect of 2-level Caching

- Holding size of 1st level cache constant:
  - Decreases miss penalty of 1st-level cache.
  - Or, increases average global hit time a bit:
    - hit time-L1 + miss rate-L1 x hit time-L2
  - but decreases global miss rate
- Holding total cache size constant:
  - Global miss rate, miss penalty about the same
  - Decreases average global hit time significantly!
    - New L1 much smaller than old L1

Miss Rate Example

- Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache
  - Miss rate for the first-level cache = 40/1000 (4%)
  - Local miss rate for the second-level cache = 20/40 (50%)
  - Global miss rate for the second-level cache = 20/1000 (2%)
- Assume miss-penalty-L2 is 200 CC, hit-time-L2 is 10 CC, hit-time-L1 is 1 CC, and 1.5 memory reference per instruction. What is average memory access time and average stall cycles per instructions? Ignore writes impact.
  - AMAT = Hit-time-L1 + Miss-rate-L1 × (Hit-time-L2 + Miss-rate-L2× Miss-penalty-L2) = 1 + 4% × (10 + 50% × 200) = 5.4 CC
  - Average memory stalls per instruction = Misses-per-instruction-L1 × Hit-time-L2 + Misses-per-instructions-L2×Miss-penalty-L2 = (40×1.5/1000) × 10 + (20×1.5/1000) ×200 = 6.6 CC
  - Or (5.4 – 1.0) × 1.5 = 6.6 CC

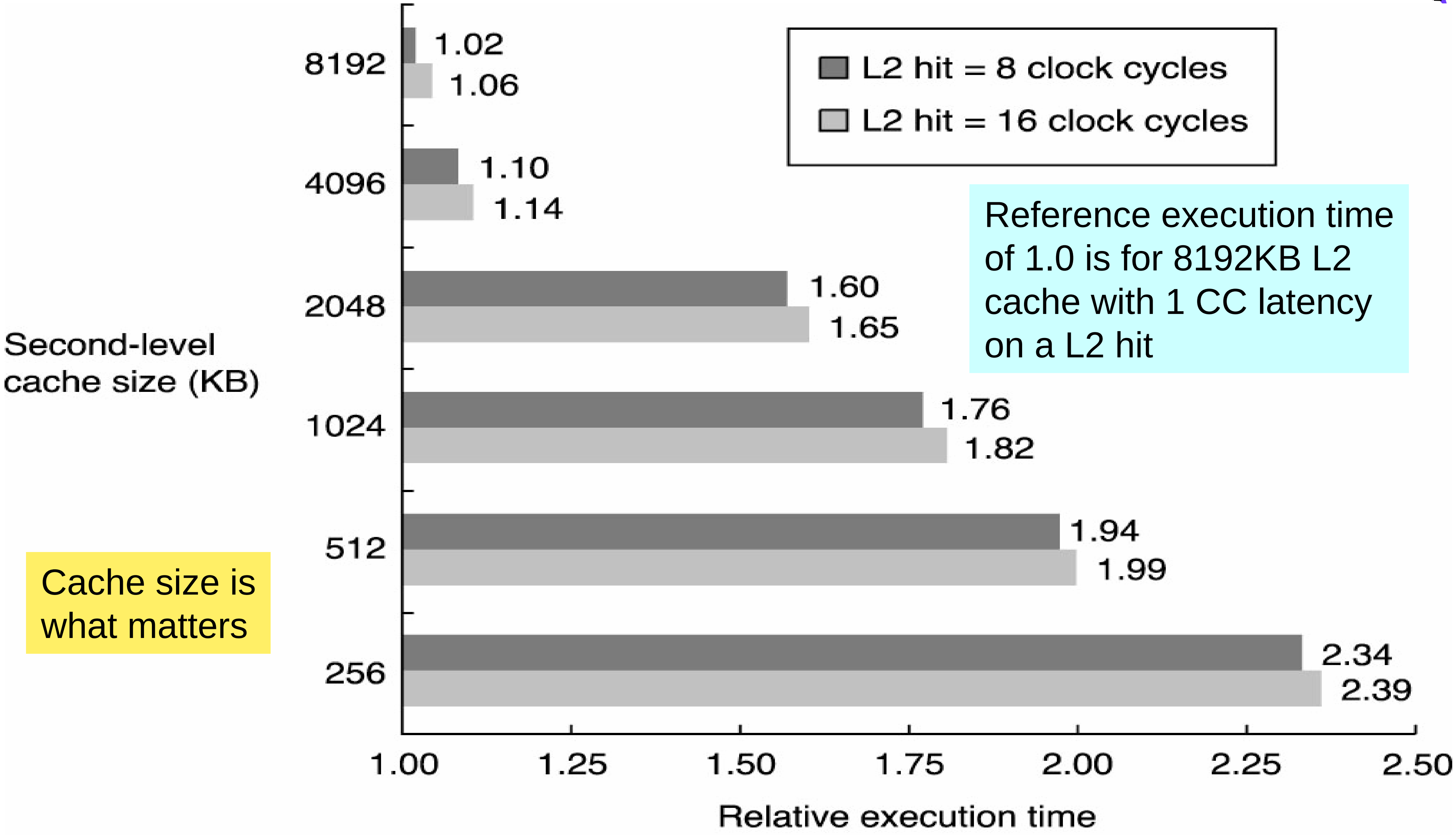
Comparing Local and Global Miss Rates



32KB L1 cache

More assumptions are shown in the legend of Figure C.14

Relative Execution Time by L2-Cache Size



Reference execution time of 1.0 is for 8192KB L2 cache with 1 CC latency on a L2 hit

Cache size is what matters

Comparing Local and Global Miss Rates

- Huge 2nd level caches
- Global miss rate close to single level cache rate provided L2 >> L1
- Global cache miss rate should be used when evaluating second-level caches (or 3rd, 4th,… levels of hierarchy)
- Many fewer hits than L1, target reduce misses

Example: Impact of L2 Cache Associativity

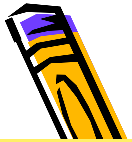
- Hit-time-L2
  - Direct mapped = 10 CC;
  - 2-way set associativity = 10.1 CC (usually round up to integral number of CC, 10 or 11 CC)
- Local-miss-rate-L2
  - Direct mapped = 25%;
  - 2-way set associativity = 20%
- Miss-penalty-L2 = 200CC
- Impact of Miss-penalty-L2
  - Direct mapped = 10 + 25% * 200 = 60 CC
  - 2-way (10 CC) = 10 + 20% * 200 = 50 CC
  - 2-way (11 CC) = 11 + 20% * 200 = 51 CC

5: Critical Word First and Early Restart

CPU normally needs one word of the block at a time

- Do not wait for full block to be loaded before restarting CPU
  - Critical Word First – request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
  - Early restart -- as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
- Benefits of critical word first and early restart depend on
  - Block size: generally useful only in large blocks
  - Likelihood of another access to the portion of the block that has not yet been fetched
    - Spatial locality problem: tend to want next sequential word, so not clear if benefit

5. Giving Priority to Read Misses Over Writes



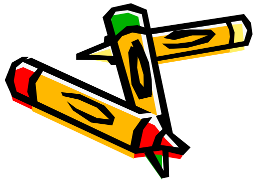
```
SW R3, 512(R0)     ;cache index 0
LW R1, 1024(R0)    ;cache index 0
LW R2, 512(R0)     ;cache index 0
```

R2=R3 ?

- In write through, write buffers complicate memory access in that they might hold the updated value of location needed on a read miss
  - **RAW** conflicts with main memory reads on cache misses
- Read miss waits until the write buffer empty → increase read miss penalty
- Check write buffer contents before read, and if no conflicts, let the memory access continue

read priority over write

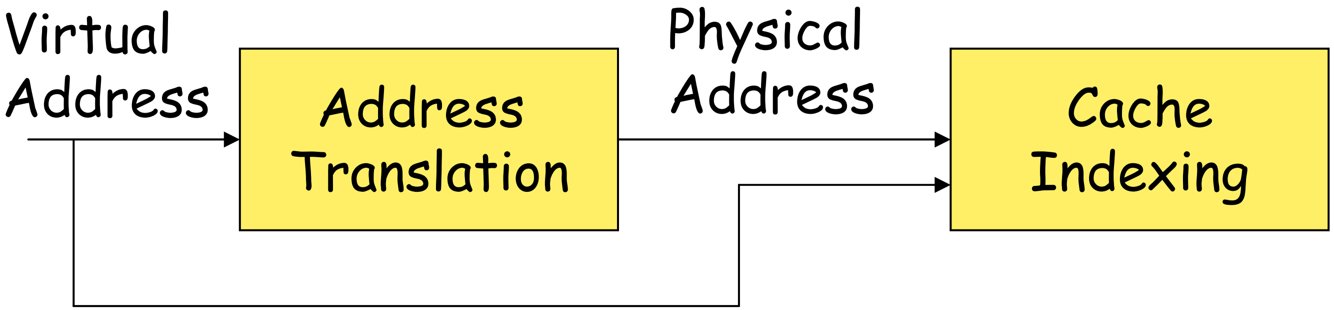
- Write Back?
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead, copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read



6. Avoiding Address Translation during Indexing of the Cache

Block address		Block offset
Tag	Index	

Virtual Cache

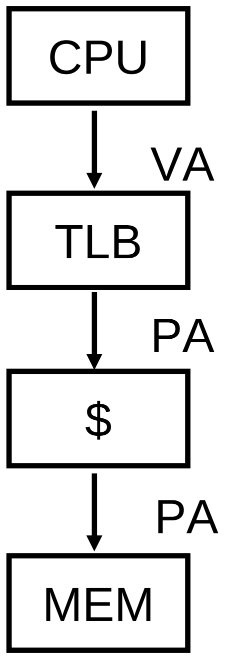


- Using virtual address for cache
  - index
  - tag comparison

➔ virtual cache (eliminate address translation times)

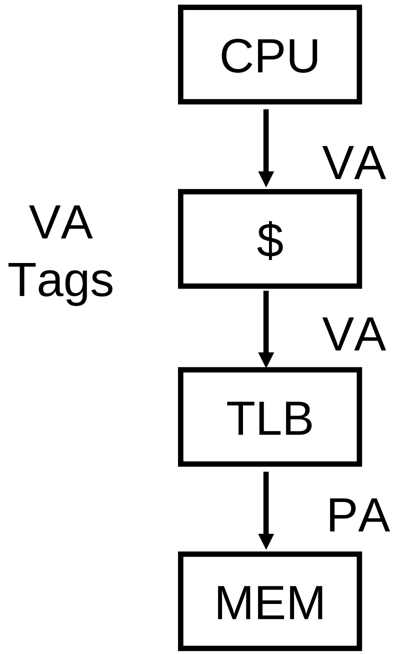
Virtually Addressed Caches

$ means cache



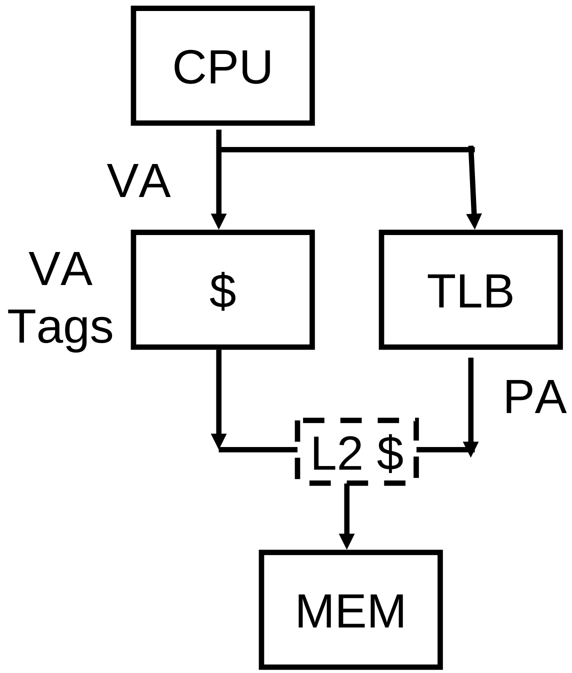
CPU
VA
TLB
PA
$
PA
MEM

Conventional Organization



VA Tags
CPU
VA
$
VA
TLB
PA
MEM

Virtually Addressed Cache
Translate only on miss
Synonym (Alias) Problem



CPU
VA
VA Tags
$
TLB
PA
L2 $
MEM

Overlap $ access with VA translation: requires $ index to remain invariant across translation

Virtual Addressed Caches

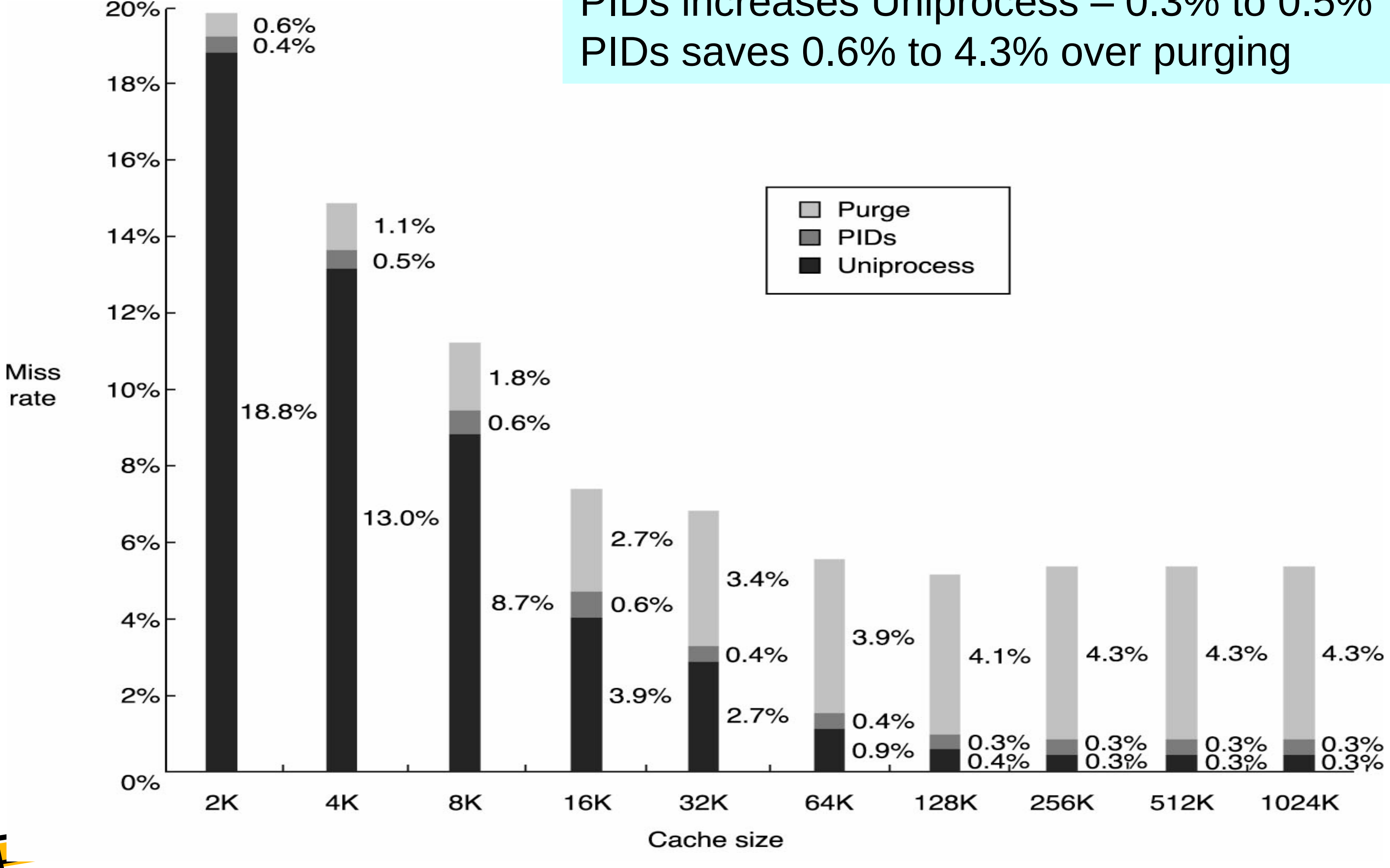
- Parallel rather than sequential access
  - Physical addressed caches access the TLB to generate the physical address, then do the cache access
- Avoid address translation during cache index
  - Implies virtual addressed cache
  - Address translation proceeds in parallel with cache index
    - If translation indicates that the page is not mapped - then the result of the index is not a hit
    - Or if a protection violation occurs - then an exception results
    - All is well when neither happen
- Too good to be true?

Why not Virtual Cache?

- Protection – necessary part of the virtual to physical address translation
  - Copy protection information on a miss, add a field to hold it, and check it on every access to virtually addressed cache.
- Task switch causes the same virtual address to refer to different physical address
  - Hence cache must be flushed
    - Creating huge task switch overhead
    - Also creates huge compulsory miss rates for new process
  - Use PID's as part of the tag to aid discrimination

Miss Rate of Virtual Caches

PIDs increases Uniprocess – 0.3% to 0.5%
PIDs saves 0.6% to 4.3% over purging



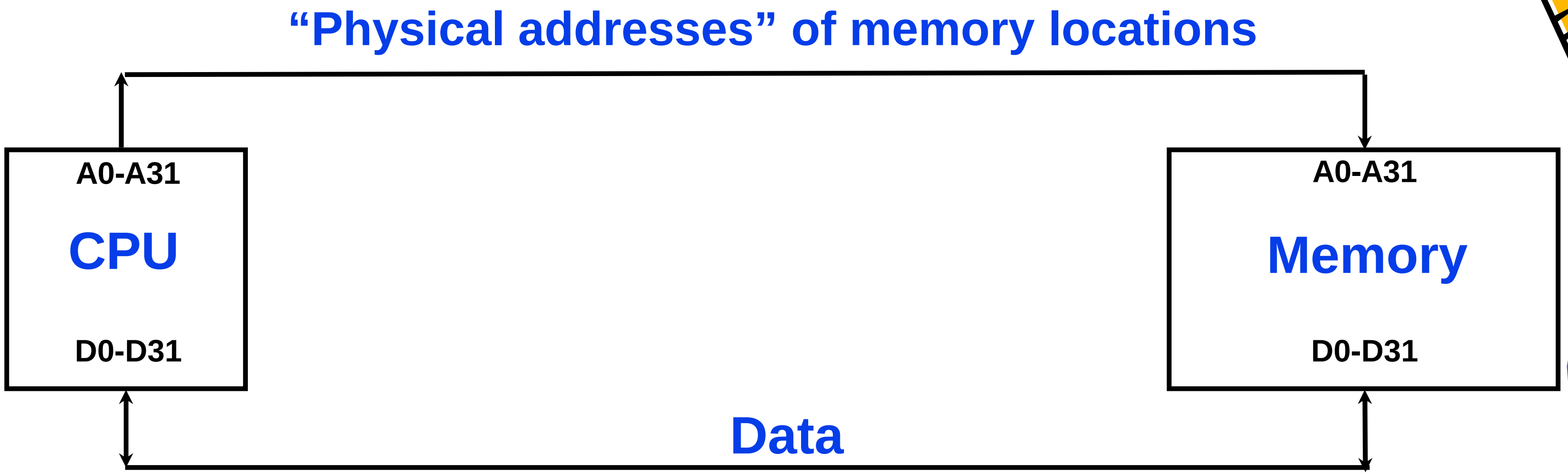
Why not Virtual Cache? (Cont.)

- Synonyms or Alias
  - OS and User code have different virtual addresses which map to the same physical address (facilitates copy-free sharing)
  - Two copies of the same data in a virtual cache → consistency issue
  - Anti-aliasing (HW) mechanisms guarantee single copy
    - On a miss, check to make sure none match PA of the data being fetched (must VA → PA); otherwise, invalidate
  - SW can help - e.g. SUN's version of UNIX
    - Page coloring - aliases must have same low-order 18 bits
- I/O – use PA
  - Require mapping to VA to interact with a virtual cache

Outline

- Review
- Cache
- Cache Performance
- 6 Basic Cache Optimization
- Virtual Memory

The Limits of Physical Addressing



All programs share one address space:
The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource

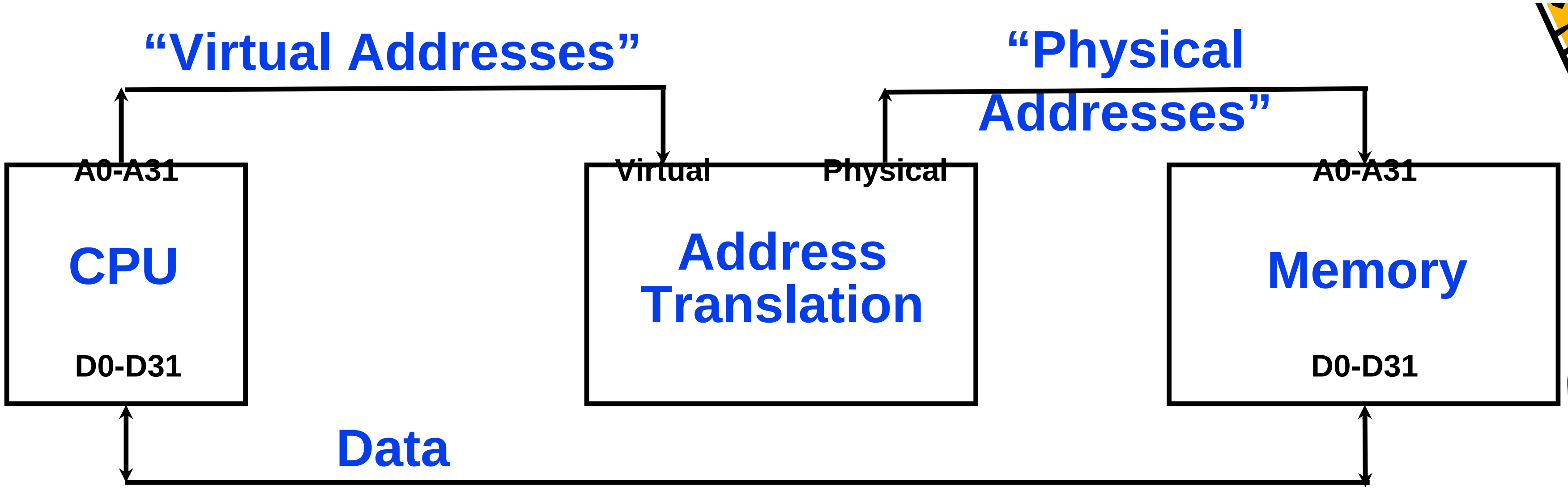
Virtual Memory

- Recall: Many processes use only a small part of address space.
- Virtual memory divides physical memory into blocks (called page or segment) and allocates them to different processes
- With virtual memory, the CPU produces virtual addresses that are translated by a combination of HW and SW to physical addresses, which accesses main memory. The process is called memory mapping or address translation
- Today, the two memory-hierarchy levels controlled by virtual memory are DRAMs and magnetic disks

Virtual Memory (Cont.)

- Permits applications to grow bigger than main memory size
- Helps with multiple process management
  - Each process gets its own chunk of memory
  - Permits protection of 1 process' chunks from another
  - Mapping of multiple chunks onto shared physical memory
  - Mapping also facilitates relocation (a program can run in any memory location, and can be moved during execution)
  - Application and CPU run in virtual space (logical memory, 0 – max)
  - Mapping onto physical space is invisible to the application
- Cache vs. VM
  - Block becomes a page or segment
  - Miss becomes a page or address fault

Solution: Add a Layer of Indirection



"Virtual Addresses"

"Physical Addresses"

A0-A31

CPU

D0-D31

Virtual

Physical

Address Translation

A0-A31

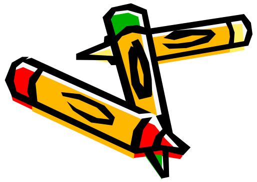
Memory

D0-D31

Data

User programs run in an standardized **virtual** address space

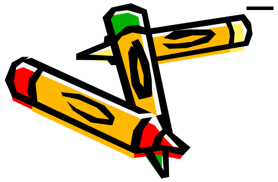
Address Translation hardware managed by the operating system (OS) maps virtual address to physical memory



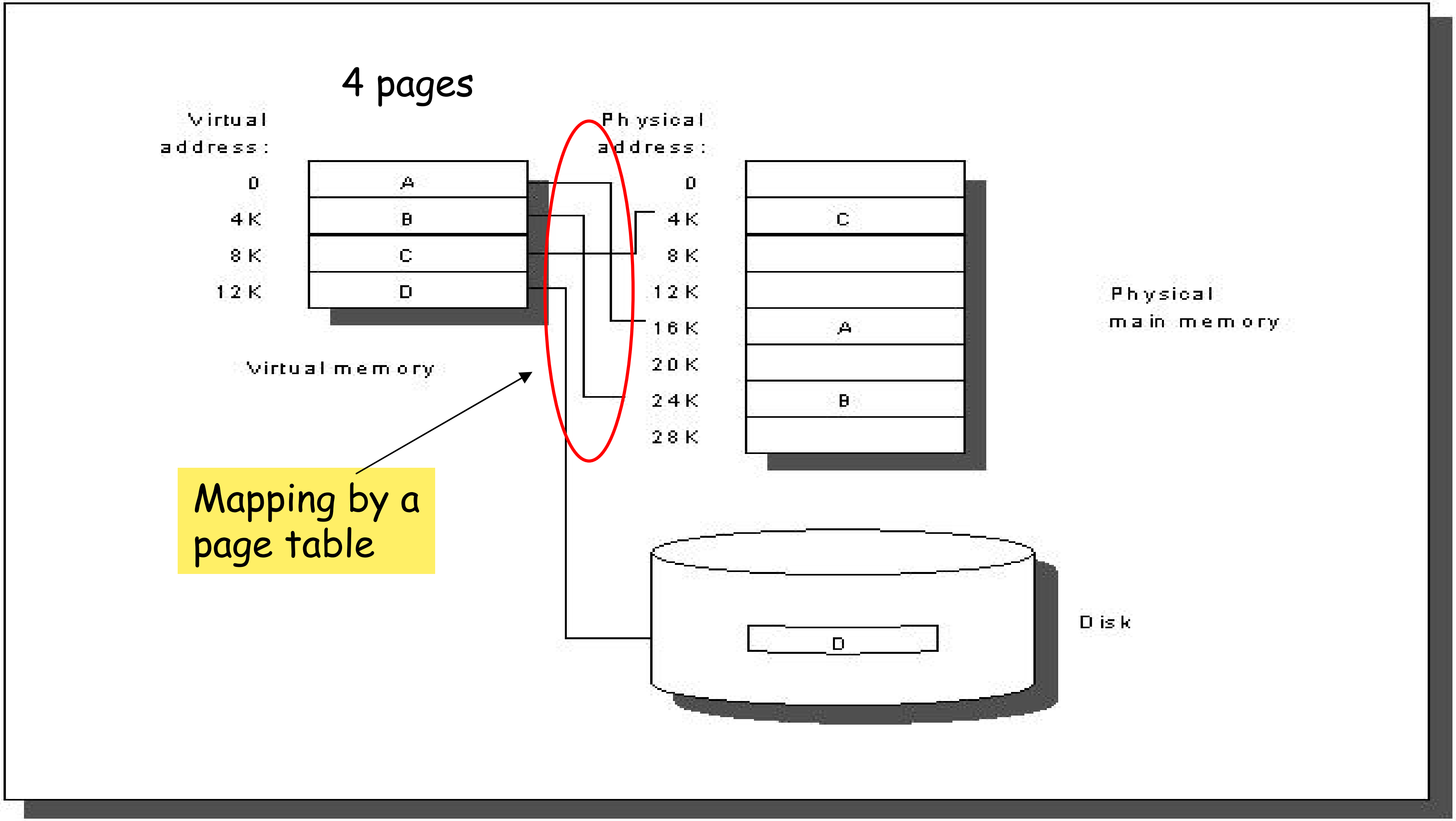
Hardware supports "modern" OS features: **Protection, Translation, Sharing**

Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program (“Working Set”) must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs
- Sharing:
  - Can map same physical page to multiple users (“Shared memory”)

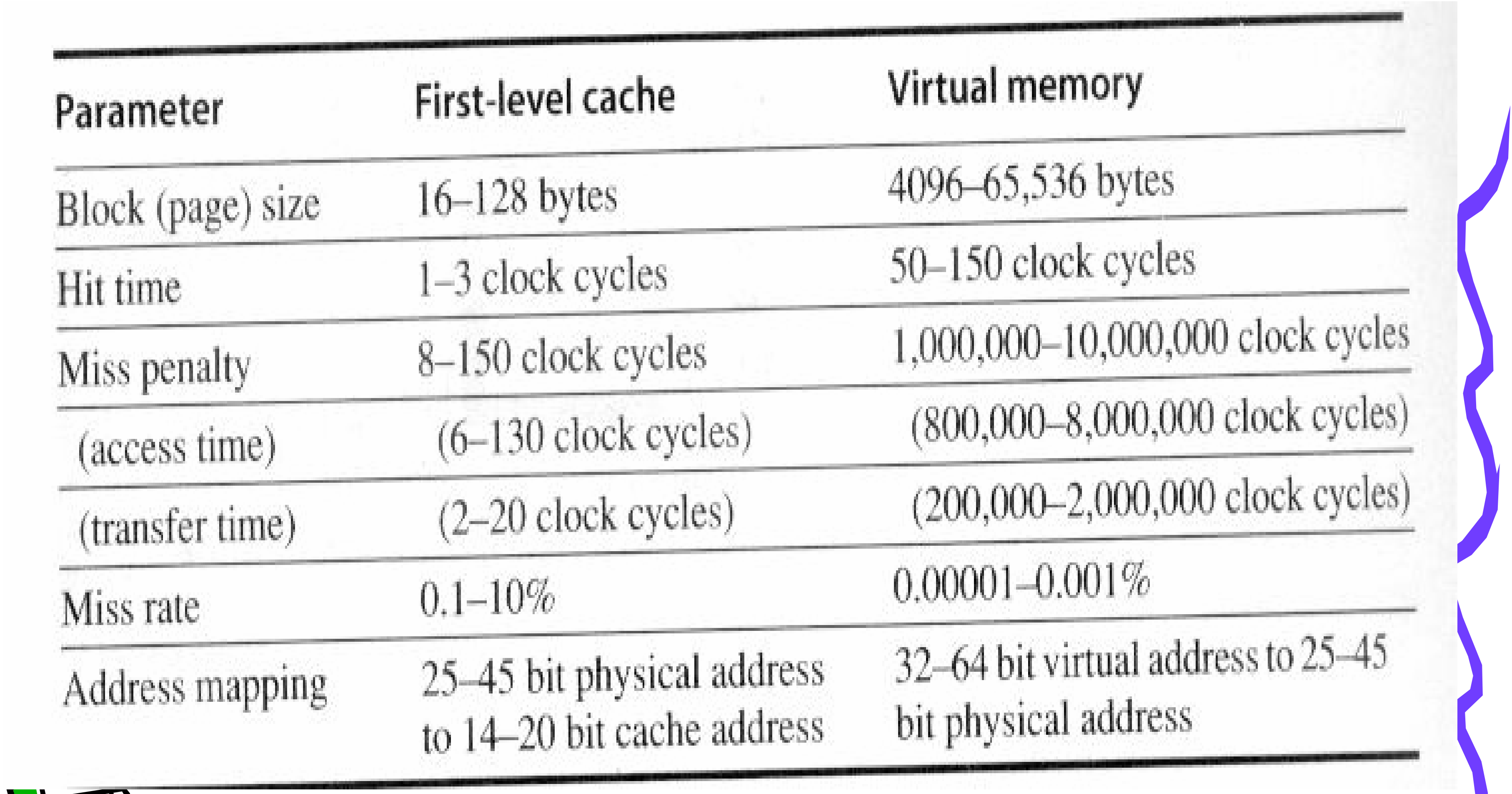


Virtual Memory



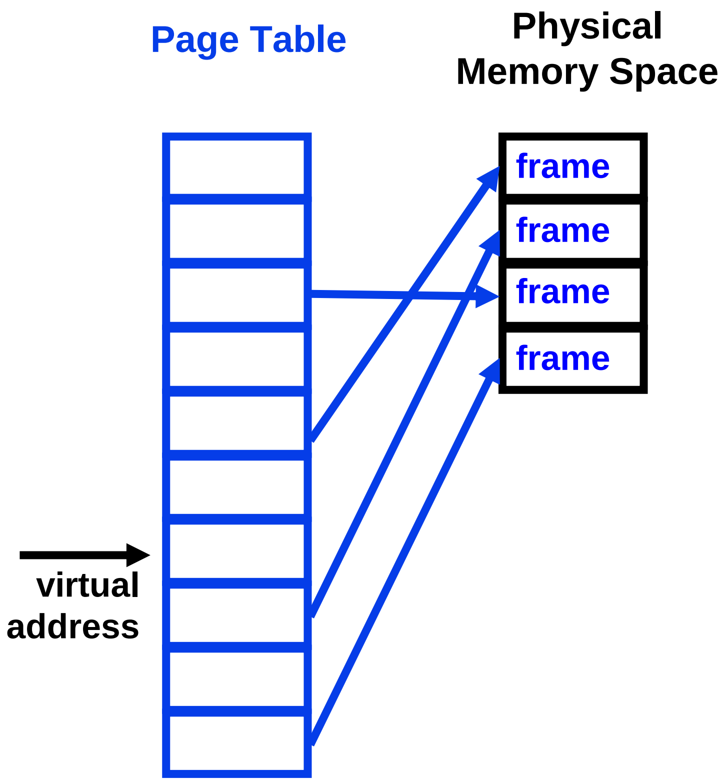
Mapping by a page table

Typical Page Parameters



Parameter	First-level cache	Virtual memory
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	50–150 clock cycles
Miss penalty	8–150 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–130 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–20 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1–10%	0.00001–0.001%
Address mapping	25–45 bit physical address to 14–20 bit cache address	32–64 bit virtual address to 25–45 bit physical address

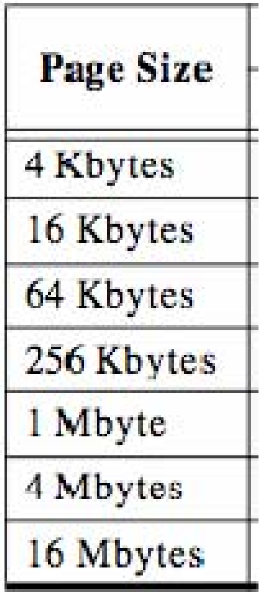
Page Tables Encode Virtual Address Spaces



OS manages the page table for each ASID

A virtual address space is divided into blocks of memory called **pages**

A machine usually supports pages of a few sizes (MIPS R4000):

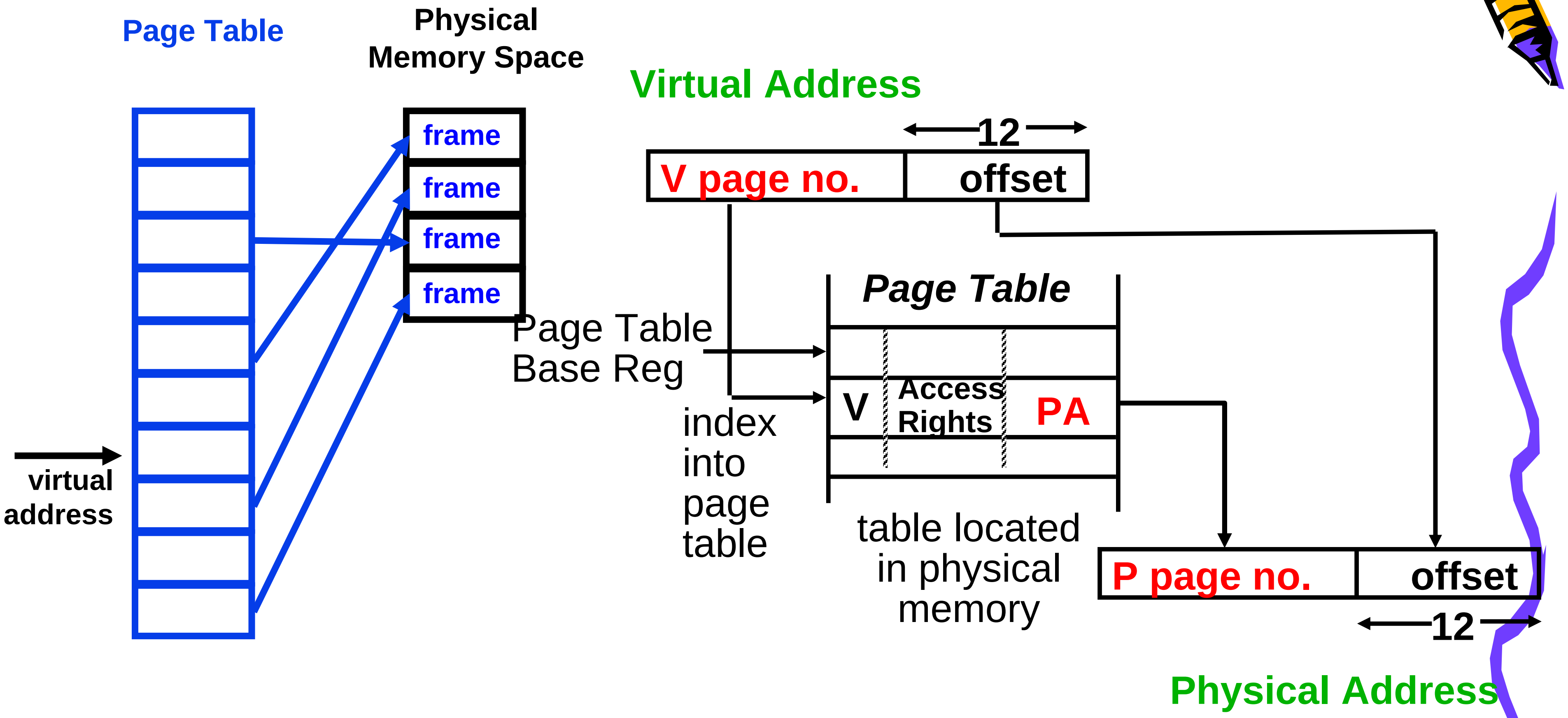


Page Size
4 Kbytes
16 Kbytes
64 Kbytes
256 Kbytes
1 Mbyte
4 Mbytes
16 Mbytes

A page table is indexed by a **virtual address**

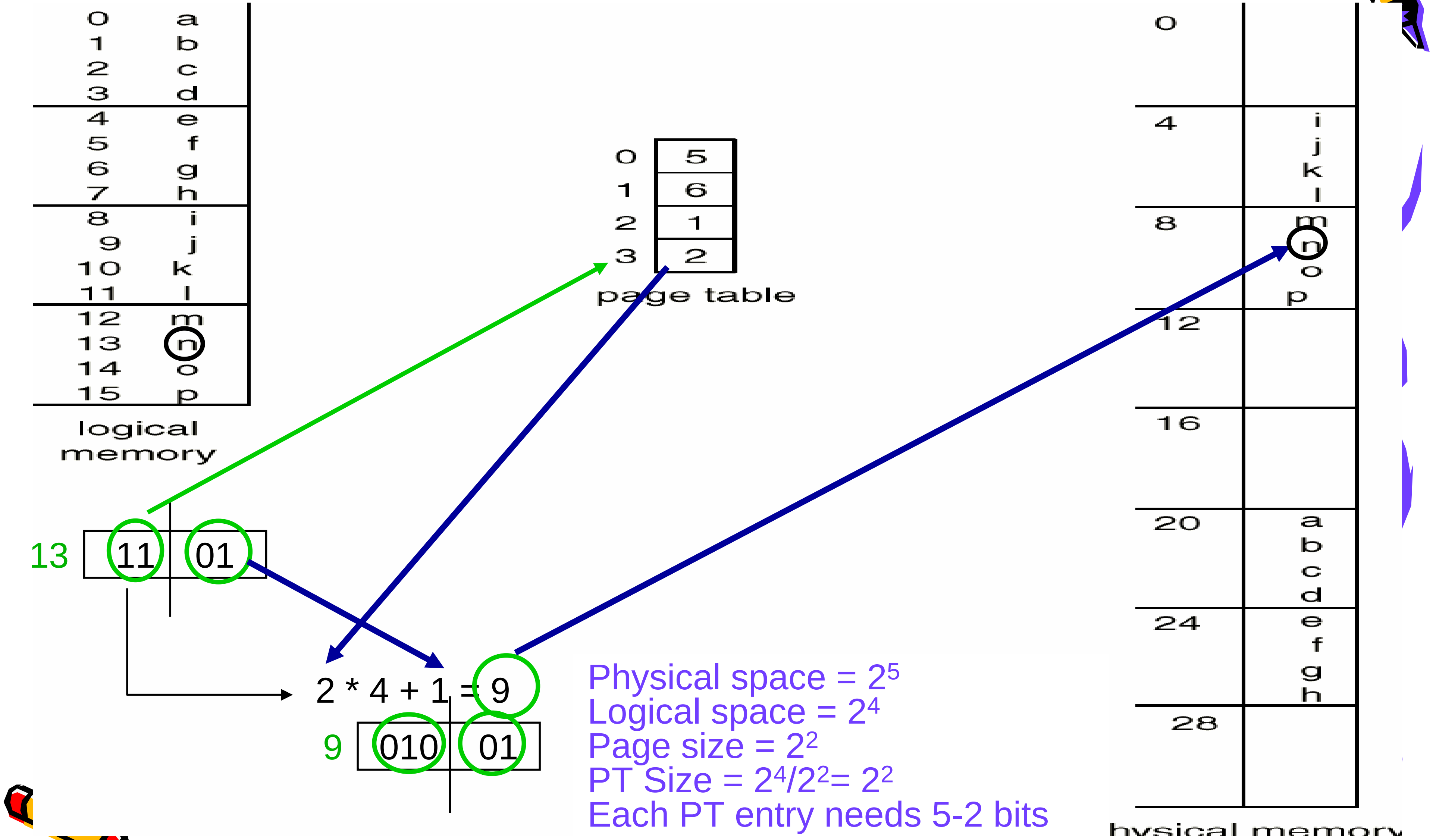
A valid page table entry codes **physical memory “frame”** address for the page

Details of Page Table



- **Page table maps virtual page numbers to physical frames (“PTE” = Page Table Entry)**
- **Virtual memory => treat memory ≈ cache for disk**

Example



Physical space = 2^5

Logical space = 2^4

Page size = 2^2

PT Size = $2^4/2^2= 2^2$

Each PT entry needs 5-2 bits

Cache vs. VM Differences

- Replacement
  - Cache miss handled by hardware
  - Page fault usually handled by OS
- Addresses
  - VM space is determined by the address size of the CPU
  - Cache space is independent of the CPU address size
- Lower level memory
  - For caches - the main memory is not shared by something else
  - For VM - most of the disk contains the file system
    - File system addressed differently - usually in I/ O space
    - VM lower level is usually called SWAP space

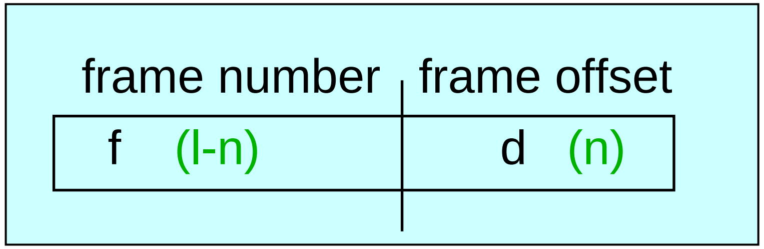
2 VM Styles - Paged or Segmented?

- Virtual systems can be categorized into two classes: pages (fixed-size blocks), and *segments* (variable-size blocks)

	Page	**Segment**
Words per address	*One*	*Two (segment and offset)*
Programmer visible?	*Invisible to application programmer*	*May be visible to application programmer*
Replacing a block	*Trivial (all blocks are the same size)*	*Hard (must find contiguous, variable-size, unused portion of main memory)*
Memory use inefficiency	*Internal fragmentation (unused portion of page)*	*External fragmentation (unused pieces of main memory)*
Efficient disk traffic	*Yes (adjust page size to balance access time and transfer time)*	*Not always (small segments may transfer just a few bytes)*

Virtual Memory - The Same 4 Questions

- Block Placement
  - Choice: lower miss rates and complex placement or vice versa
    - Miss penalty is huge, so choose low miss rate → place anywhere
    - Similar to fully associative cache model
- Block Identification - both use additional data structure
  - Fixed size pages - use a page table
  - Variable sized segments - segment table



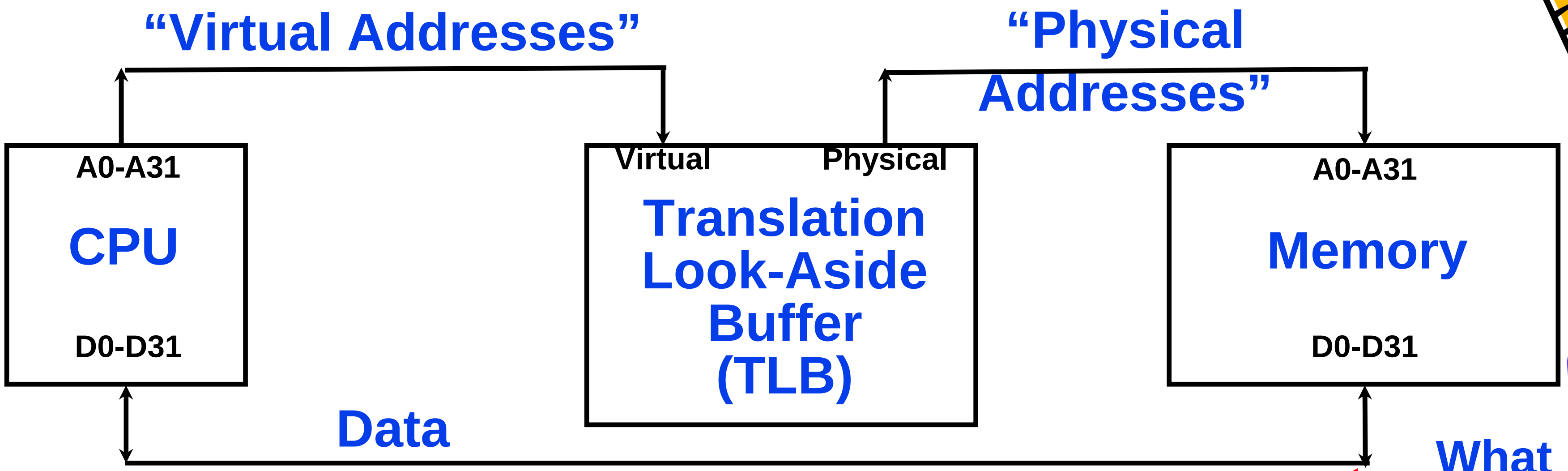
The Same 4 Questions for VM

- Block Replacement -- LRU is the best
  - However true LRU is a bit complex – so use approximation
    - Page table contains a use tag, and on access the use tag is set
    - OS checks them every so often - records what it sees in a data structure - then clears them all
    - On a miss the OS decides who has been used the least and replace that one
- Write Strategy -- always write back
  - Due to the access time to the disk, write through is silly
  - Use a dirty bit to only write back pages that have been modified

Techniques for Fast Address Translation

- Page table is kept in main memory (kernel memory)
  - Each process has a page table
- Every data/instruction access requires two memory accesses
  - One for the page table and one for the data/instruction
  - Can be solved by the use of a special fast-lookup hardware cache called associative registers or translation look-aside buffers (TLBs)
- If locality applies then cache the recent translation
  - TLB = translation look-aside buffer
  - TLB entry: virtual page no, physical page no, protection bit, use bit, dirty bit

MIPS Address Translation

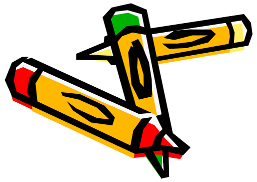


What is the table of mappings that it caches?

Translation Look-Aside Buffer (TLB)

A small fully-associative cache of mappings from virtual to physical addresses

TLB also contains protection bits for virtual address

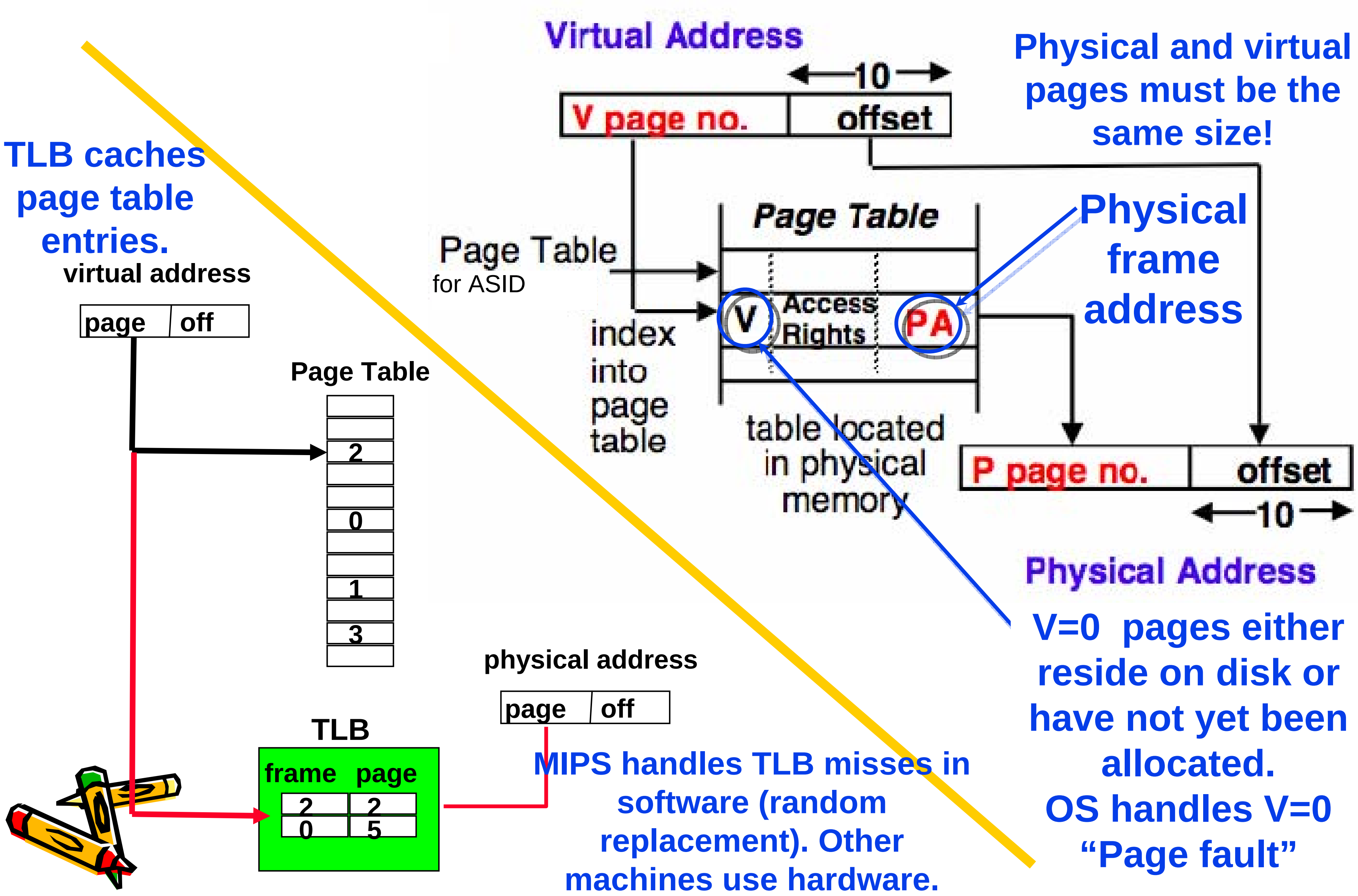


Fast common case: Virtual address is in TLB, process has permission to read/write it.

TLB

- The TLB must be on chip; otherwise it is worthless
  - Fully associative – parallel search
- Typical TLB's
  - Hit time - 1 cycle
  - Miss penalty - 10 to 30 cycles
  - Miss rate - .1% to 2%
  - TLB size - 32 B to 8 KB

The TLB Caches Page Table Entries



TLB caches page table entries.

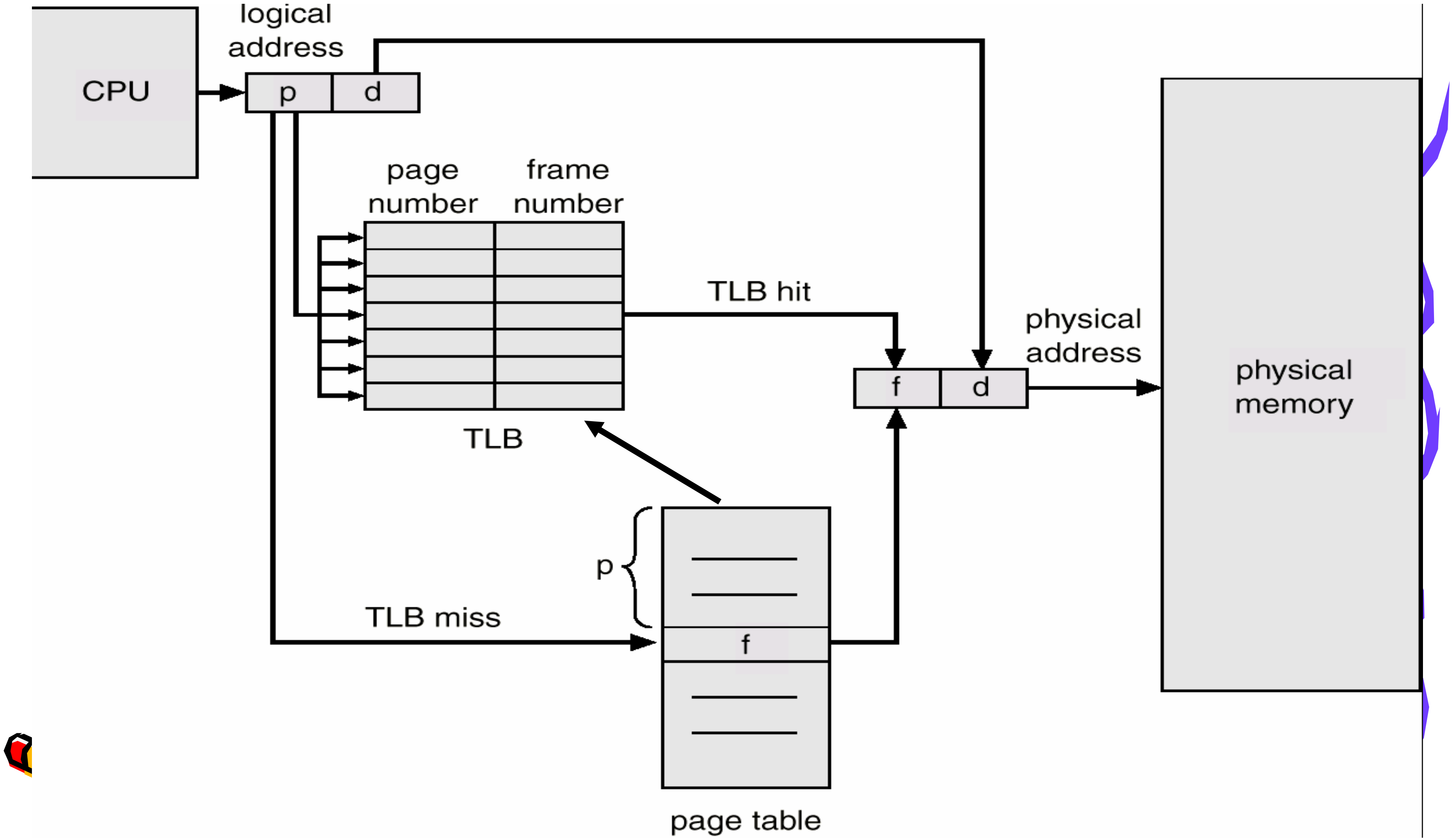
Physical and virtual pages must be the same size!

Physical frame address

V=0 pages either reside on disk or have not yet been allocated.
OS handles V=0 “Page fault”

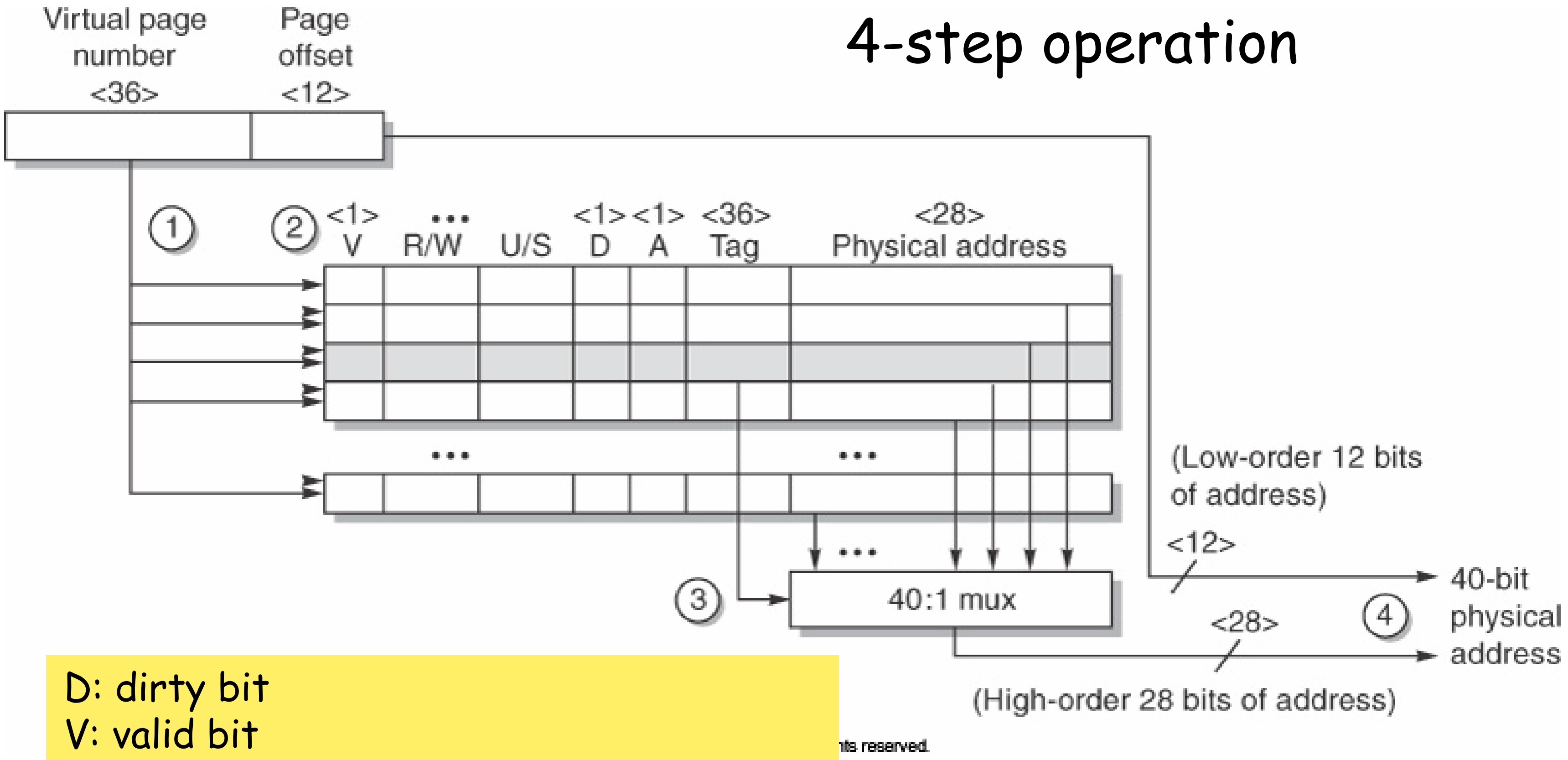
MIPS handles TLB misses in software (random replacement). Other machines use hardware.

Paging Hardware with TLB



Opteron data TLB Organization

4-step operation



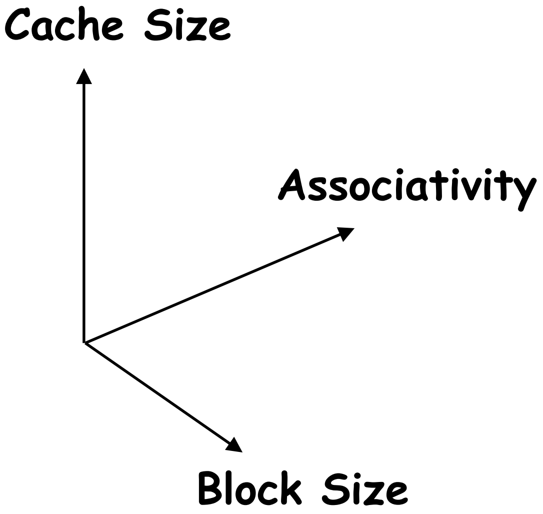
D: dirty bit
V: valid bit
Step 1&2: send virtual address to all tags

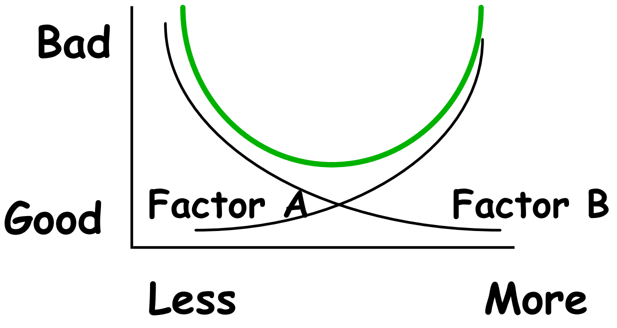
Page Size

- An architectural choice…
- Large pages are good:
  - Reduces page table size
  - Amortizes the long disk access
  - If spatial locality is good then hit rate will improve
  - Reduce the number of TLB miss
- Large pages are bad:
  - More internal fragmentation
    - If everything is random each structure's last page is only half full
  - Process start up time takes longer

Summary (1/3): The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Summary (2/3): Caches

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity.
- Nightmare Scenario: ping pong effect!
- Write Policy: Write Through vs. Write Back
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): affects Compilers, Data structures, and Algorithms

Summary (3/3): TLB, Virtual Memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
  1) Where can block be placed?
  2) How is block found?
  3) What block is replaced on miss?
  4) How are writes handled?
- Today VM allows many processes to share single memory without having to swap all processes to disk;
- Today VM protection is more important than memory hierarchy benefits, but computers insecure

Why Protection?

- Multiprogramming forces us to worry about it
  - A computer is shared by several programs simultaneously
- Hence lots of processes
  - Hence task switch overhead
  - HW must provide savable state
  - OS must promise to save and restore properly
  - Most machines task switch every few milliseconds
  - A task switch typically takes several microseconds
- Process protection
  - Each process has its own status state such that one process cannot modify another