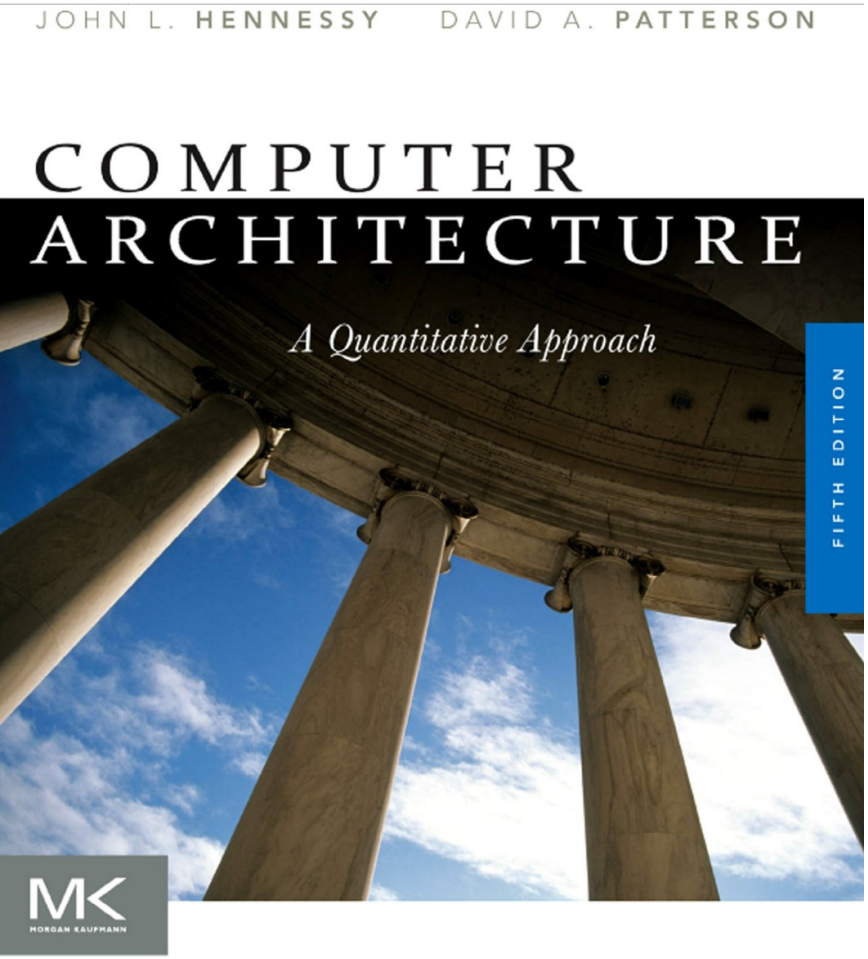
Computer Architecture

Memory Hierarchy Design

Dr. Mohammad Reza Selim

Resources

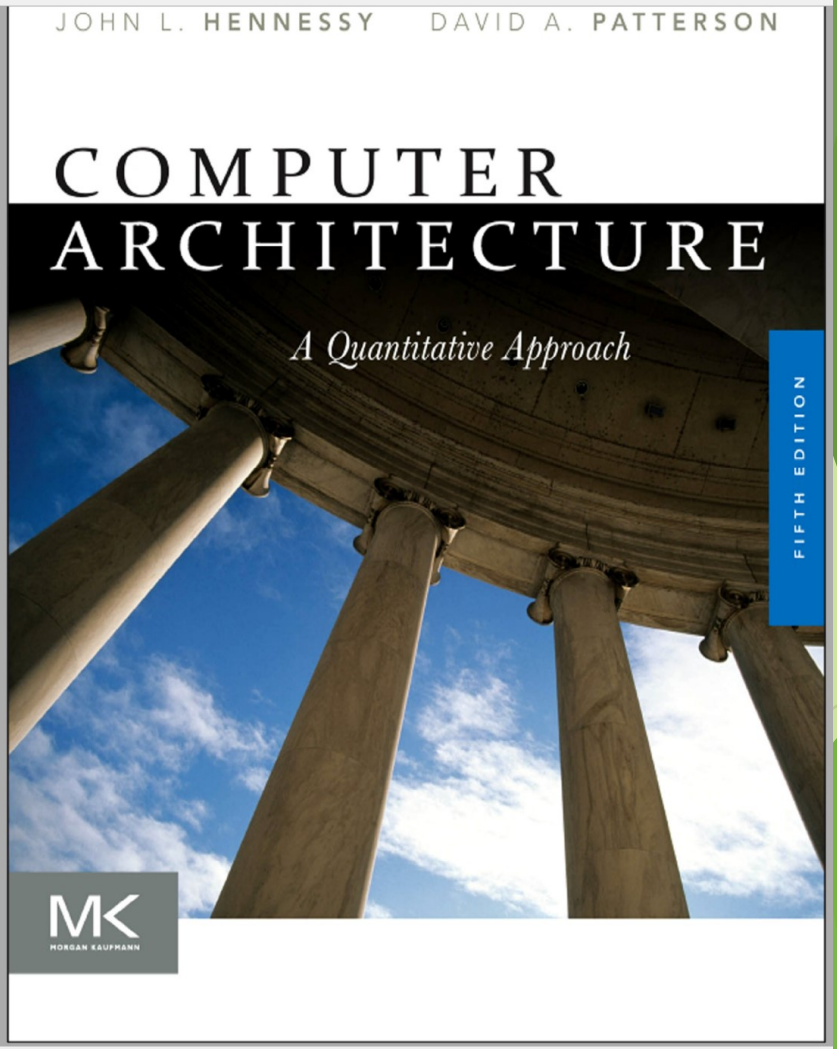


- Text Book:
  - Computer Architecture – A Quantitative Approach, 5th or Later Edition
- Other Resources
  - http://ocw.uc3m.es/ingenieria-informatica/computer-architecture/lecture-notes-1/
  - http://www.icl.utk.edu/~luszczek/teaching/courses/fall2013/cosc530/
  - https://www.youtube.com/playlist?list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX
  - https://www.youtube.com/watch?v=GTObrKKbRww&list=PLAwxTw4SYaPkKfusBLVfklgfdcB3BNpwX
  - https://en.wikipedia.org/

Resources (Memory Hierarchy Design)

Bookmarks

- Front Cover
- In Praise of Computer Architecture: A Quantitative ApproachFifth Edition
- Computer Architecture: A Quantitative Approach
- Copyright
- Dedication
- Foreword
- Table of Contents
- Preface
- Acknowledgments
- 1 Fundamentals of Quantitative Design and Analysis
- 2 Memory Hierarchy Design
- 3 Instruction-Level Parallelism and Its Exploitation
- 4 Data-Level Parallelism in Vector, SIMD, and GPU Architectures
- 5 Thread-Level Parallelism
- 6 Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism
- Appendix A. Instruction Set Principles
- Appendix B. Review of Memory Hierarchy
- Appendix C. Pipelining: Basic and Intermediate Concepts
- References
- Index
- Translation between GPU terms in book and official NVIDIA and OpenCL terms



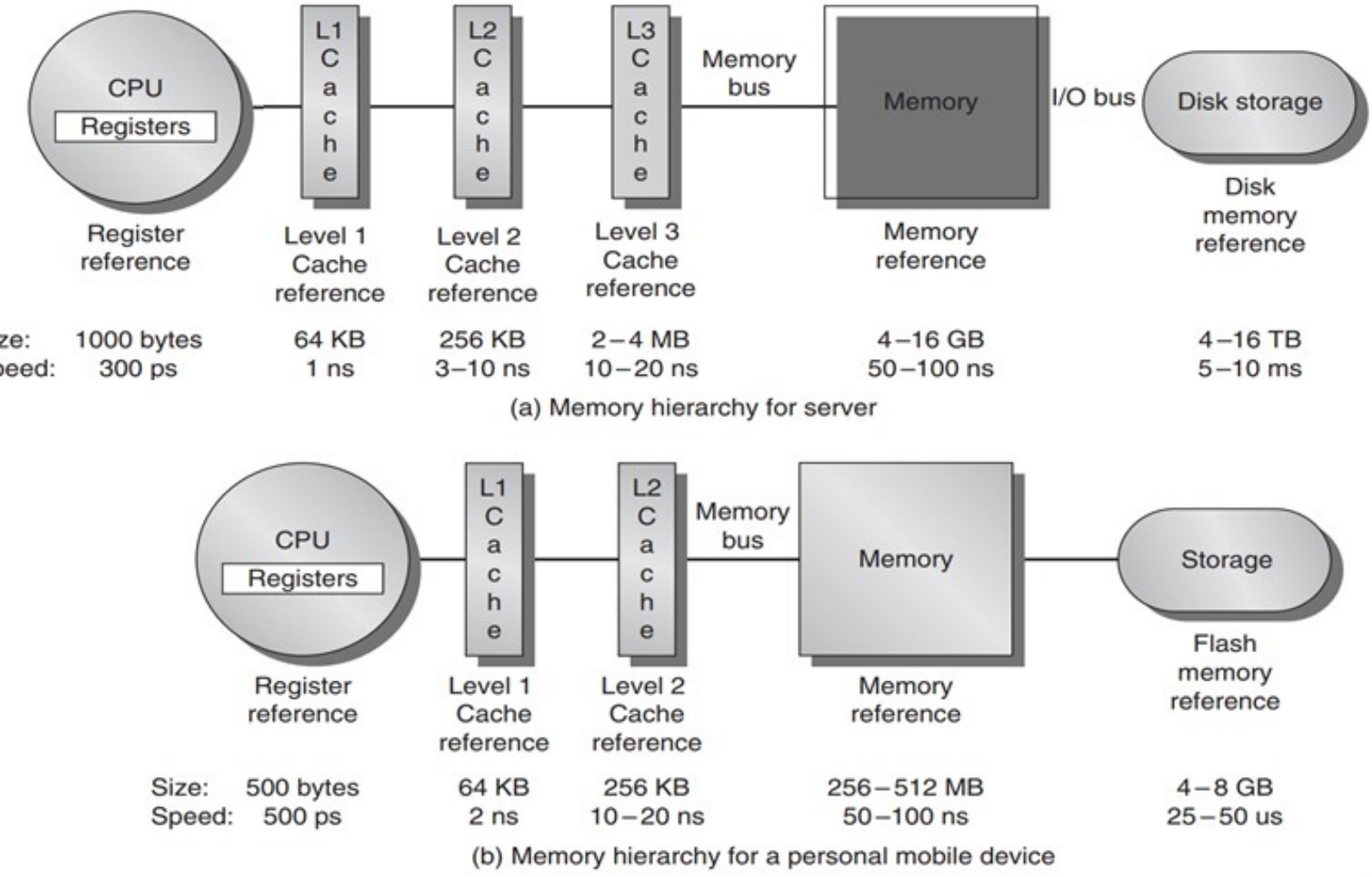
Lecture Outline

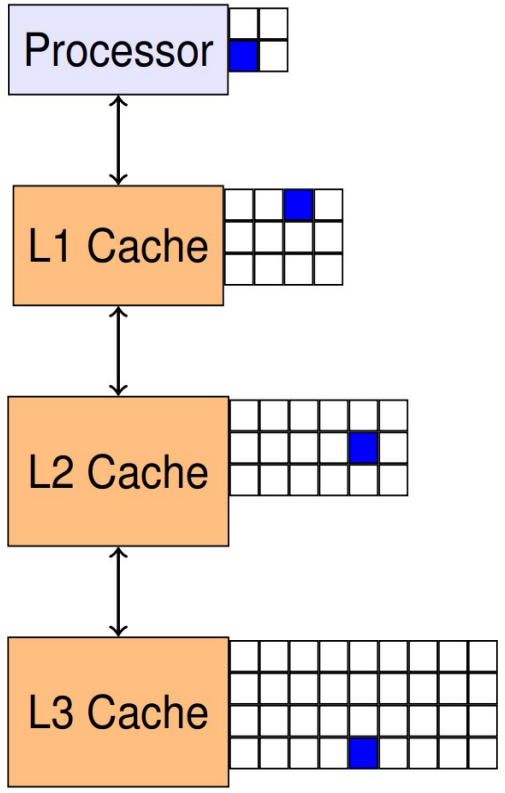
- Introduction to memory and memory hierarchy
- Why memory hierarchy (specially, cache memory) is necessary?
- Cache memory operations:
  - Block Placement
  - Block Identification
  - Block Replacement
  - Write Strategies

Memory Hierarchy Design

Introduction to Memory and Memory Hierarchy

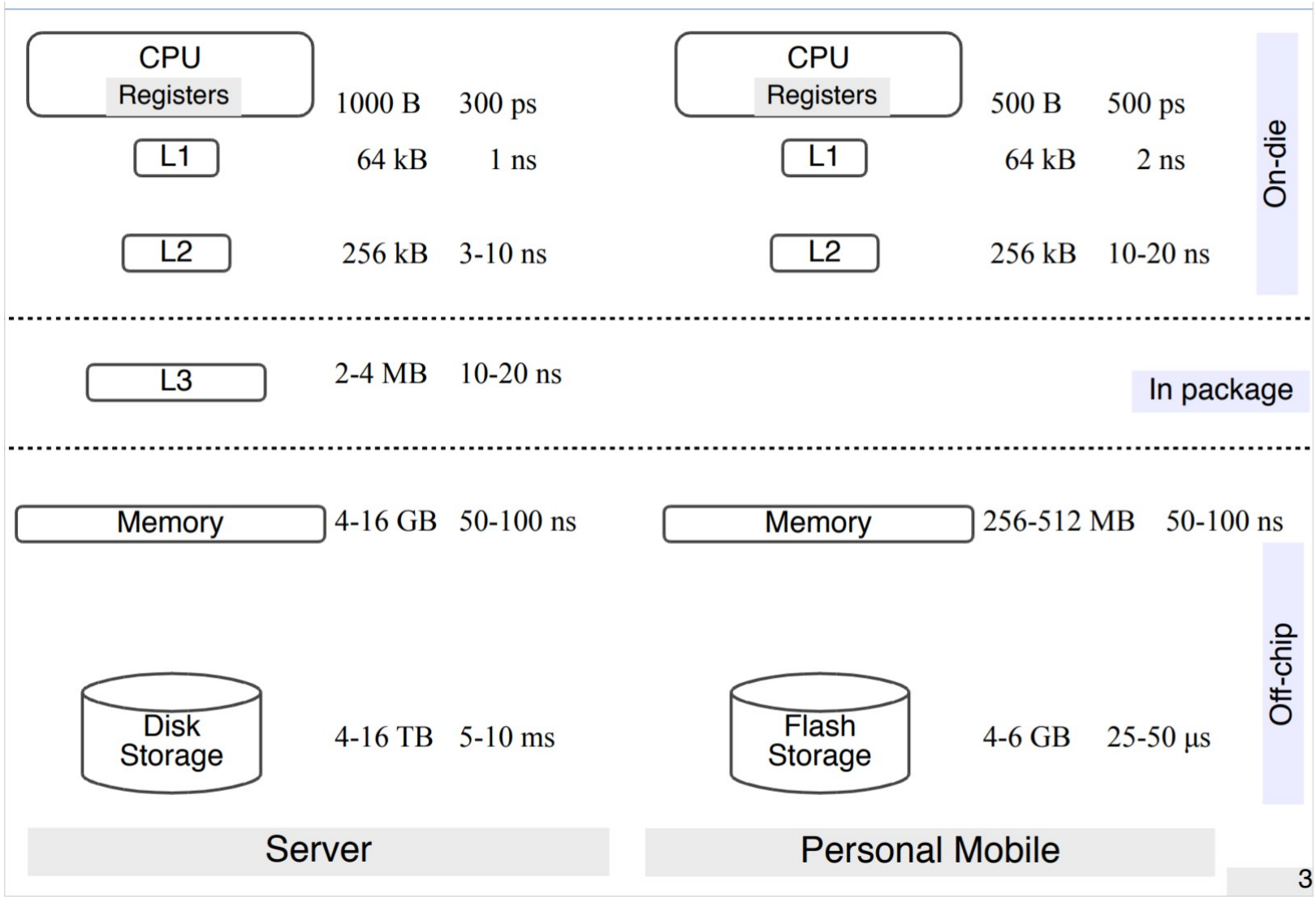
Memory Hierarchy



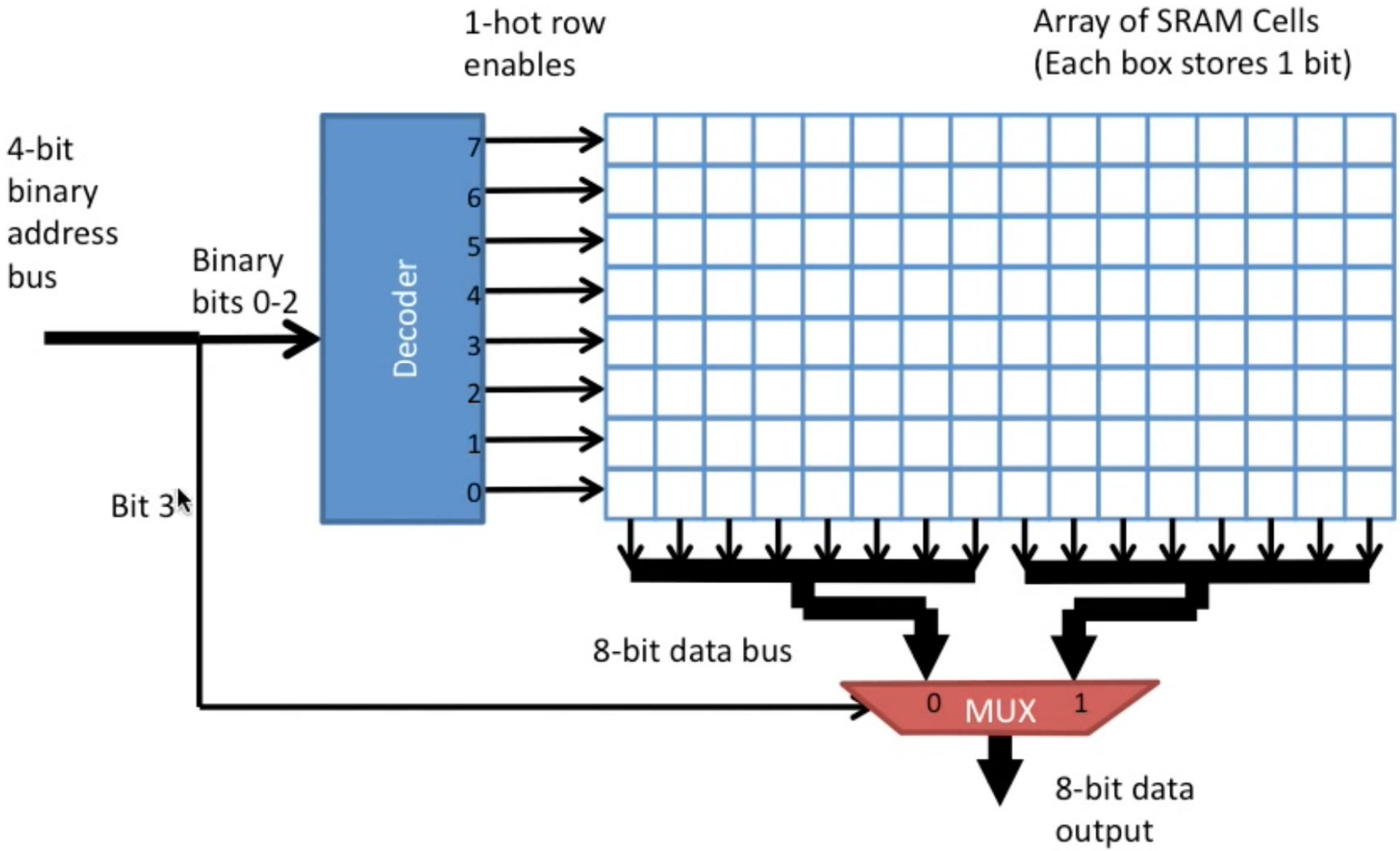


re The levels in a typical memory hierarchy in a server computer shown on a) and in a personal mobile device (PMD) on the bottom (b).

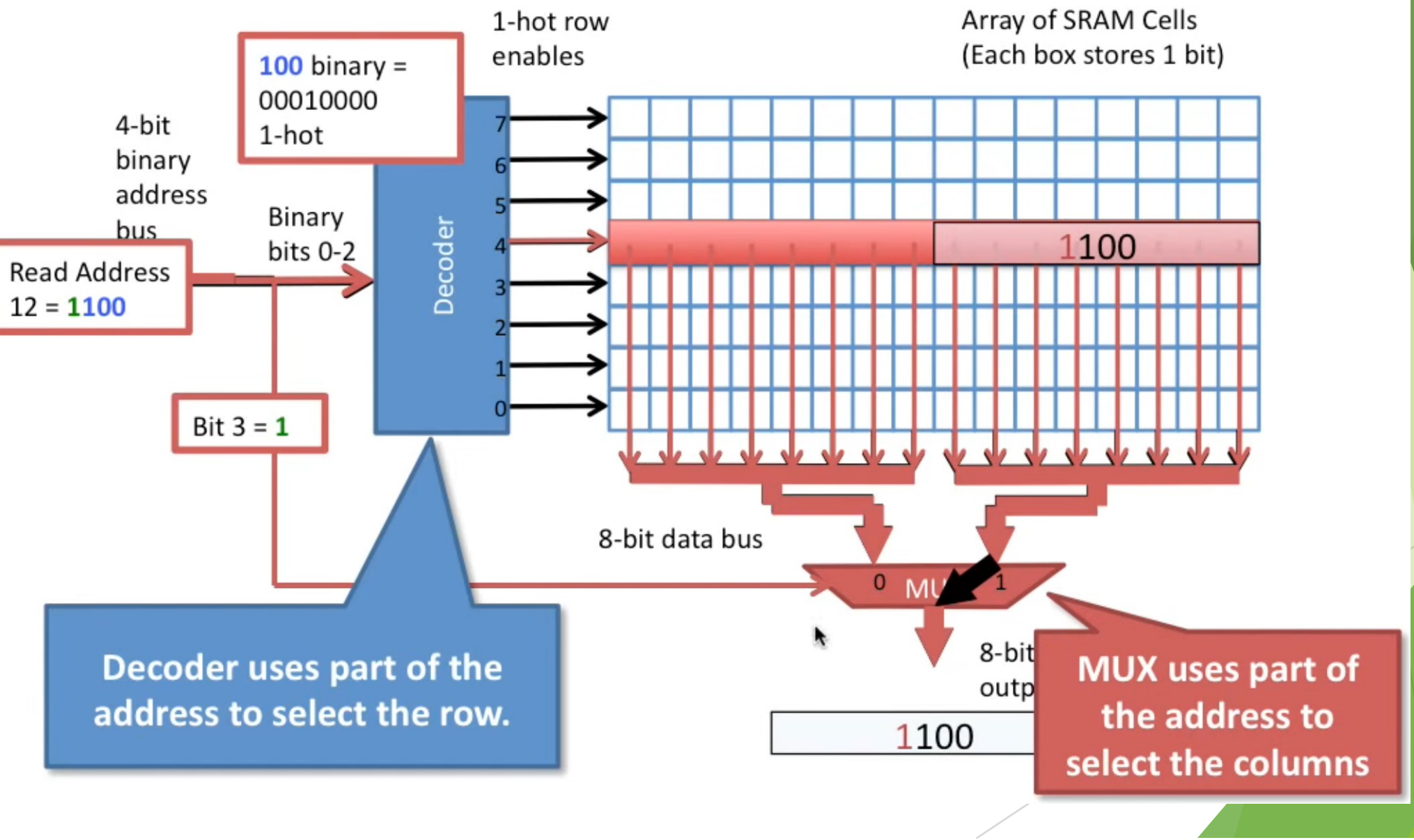
Memory Hierarchy



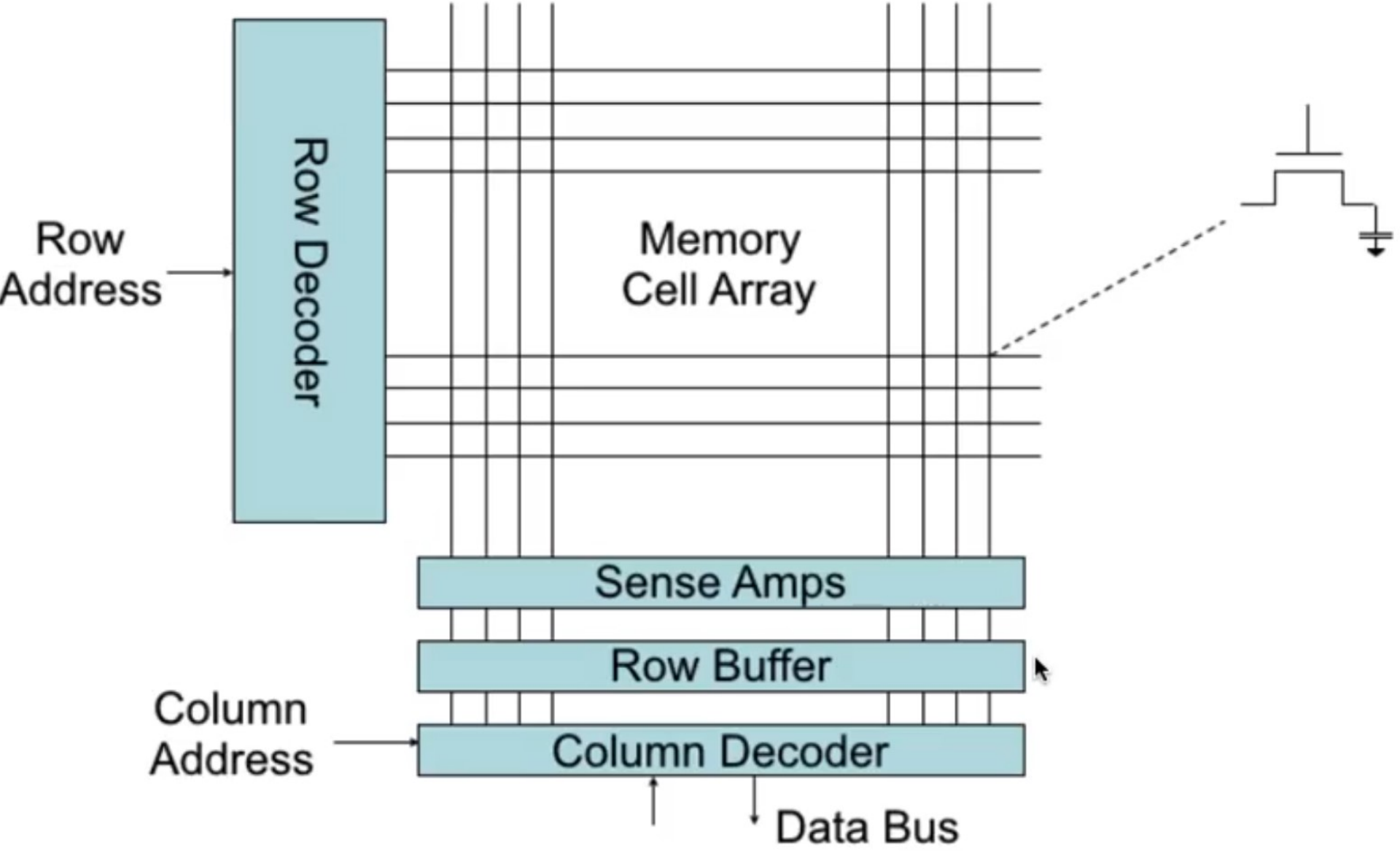
Memory (SRAM)


1-hot row enables
Array of SRAM Cells (Each box stores 1 bit)
4-bit binary address bus
Binary bits 0-2
Decoder
7
6
5
4
3
2
1
0
Bit 3
8-bit data bus
0
MUX
1
8-bit data output

Memory (SRAM Read Operation)



Memory (DRAM)


Row Address
Row Decoder
Memory Cell Array
Sense Amps
Row Buffer
Column Address
Column Decoder
Data Bus

Memory Hierarchy Design

Why memory hierarchy?

Why Memory Hierarchy?

- Goal: (Programmers want) unlimited amount of memory with low latency
- Fast memory technology is more expensive per bit than slower memory
- Solution: Use principle of Locality and organize memory system into a hierarchy
  - – Entire addressable memory space available in largest, slowest, cheapest memory
  - – Incrementally smaller and faster memories, each containing a subset of the memory below it, proceed in steps up toward the processor
- Temporal and spatial locality insures that nearly all memory references can be found in smaller memories
  - – Gives the illusion of a large, fast memory being presented to the processor

Principle of Locality / Locality of Reference

- *principle of locality*: Programs tend to reuse data and instructions they have used recently.
  - A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code.
- *Temporal locality* - recently accessed items are likely to be accessed in the near future.
  - Examples: Loops, variable reuse, etc
- *Spatial locality* - items whose addresses are close together tend to be referenced close together in time.
  - Examples: Sequential execution of instructions, arrays, etc
- If a consecutive sequence of content of main memory is transferred to cache at a time, system can be benefited by both of localities.

Why Memory Hierarchy?

Large gap in performance between Processor and Main Memory

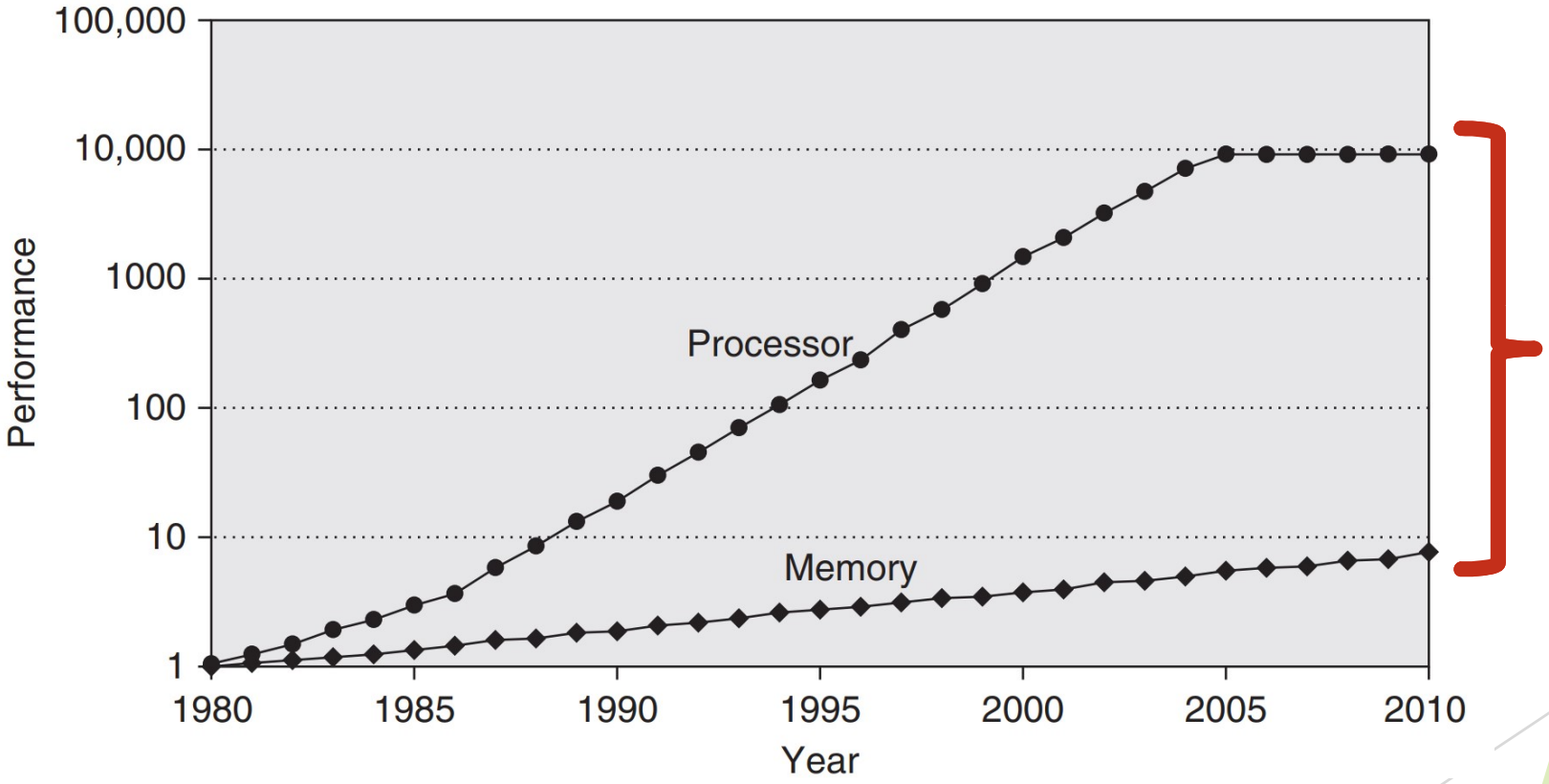


Figure: The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency).

Why Memory Hierarchy?

Becomes more crucial with **multi-core** processors.

- Aggregate peak bandwidth grows with # cores:
- Intel Core i7 (older generation) can generate two references per core per clock
- Four cores and 3.2 GHz clock
  - 25.6 billion 64-bit data references/second +
  - 12.8 billion 128-bit instruction references
  - = 409.6 GB/s!
  - DRAM bandwidth is only 6% of this (25 GB/s)
  - Requires: – Multi-port, pipelined caches
    - – Two levels of cache per core
    - – Shared third-level cache on chip

Why Memory Hierarchy?

Power consumption is also an issue

- High-end microprocessors have >10 MB on-chip cache
  - – Consumes large amount of area and power budget
  - – Both static (idle) and dynamic power is an issue
- Personal/mobile devices have – 20-50x smaller power budget
  - – 25%-50% is consumed by cache memory

Cache Memory

- Cache memory – first level of memory receiving address from processor
- Cache hit – found in top level cache
- Cache miss – not found in top level cache
- Latency – Time to get the first word
- Bandwidth – Time to get the rest of the block
- Performance = 1/Latency
- In-order execution – Maintains program order
- Out-of-order execution – Independent instructions need not maintain order. It is a capability of a processor
- Stall – means CPU is paused (on miss)
- Memory stall cycles – no. of cycles processor is stalled waiting for memory access
- Miss penalty – cost/stall cycles per miss
- Miss rate – number of missed accesses per no. of accesses. One of the most important (but not only) measures of cache design.

Cache Memory (Cont.)

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

This equation assumes that the CPU clock cycles include the time to handle a cache hit and that the processor is stalled during a cache miss.

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

The advantage of the last form is that the components can be easily measured.

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

Cache Memory (Cont.)

Example Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2%, how much faster would the computer be if all instructions were cache hits?

Answer First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}\\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}\\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.5) \times 0.02 \times 25\\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term (1 + 0.5) represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle}\\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{Clock cycle}}\\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster.

Memory Hierarchy Design

Cache Memory Operations

Four questions about Cache Memory

- 1. Where is a block placed in the upper level?
  - Block placement.
- 2. How is a block found in the upper level?
  - Block identification.
- 3. Which block must be replaced on a miss?
  - Block replacement.
- 4. What happens on a write?
  - Write strategy

Memory Hierarchy Design

Cache Memory Operations:
Block Placement

Block Placement

- Direct mapping:
  - Placement -> block MOD no of blocks
- Fully associative mapping:
  - Placement -> Anywhere.
- Set associative mapping:
  - Set placement -> block MOD no of sets
  - Block placement within set -> Anywhere
- n-way set associative:
  - n blocks in a set

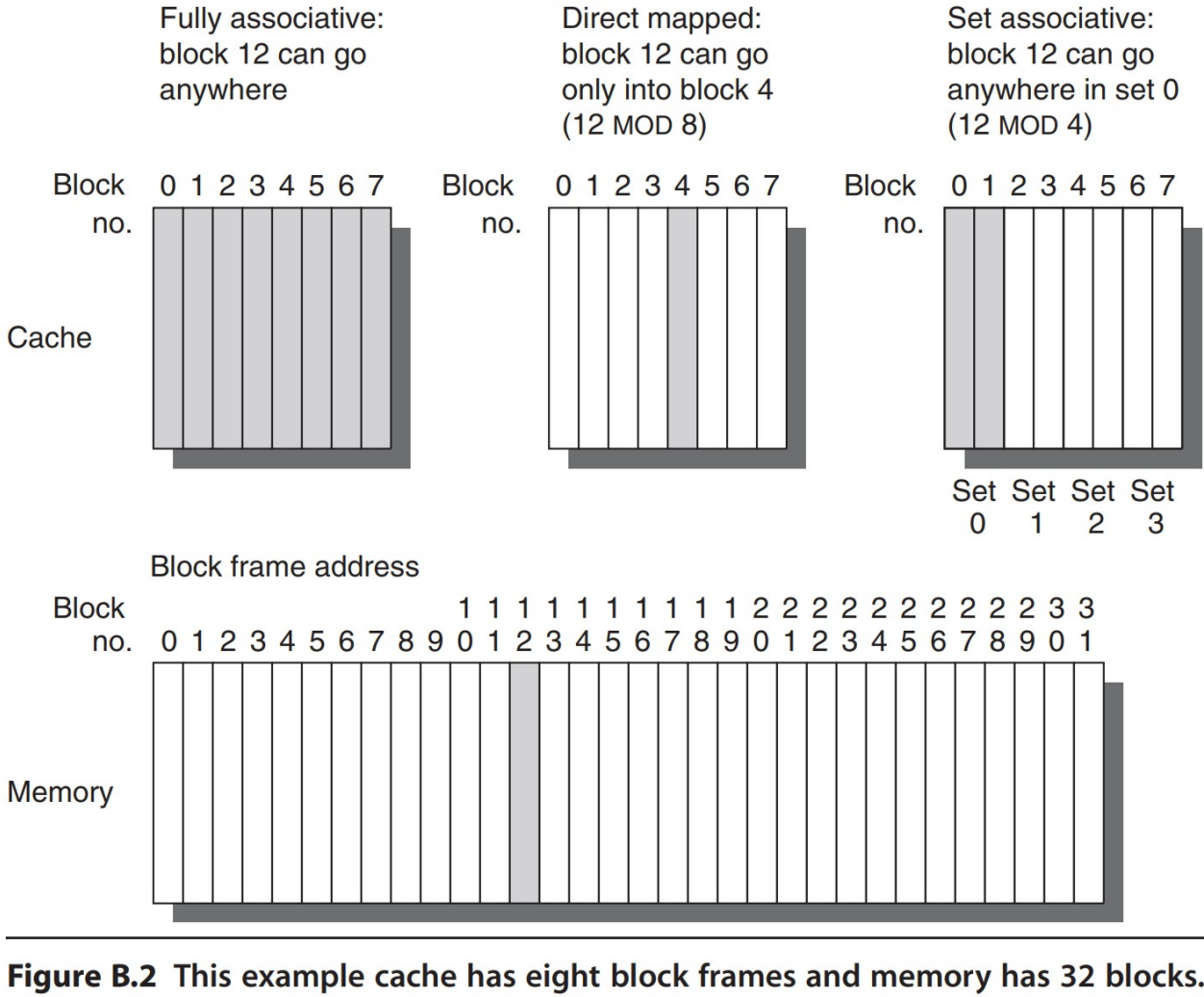


Figure B.2 This example cache has eight block frames and memory has 32 blocks.

Memory Hierarchy Design

Cache Memory Operations:
Block Identification

Block Identification

- Physical address coming out of the CPU (after converting virtual address) and entering the cache has three parts Tag, Index and Block Offset.
- Block address:
  - Tag: Identifies block within a set
    - Validity bit in every entry to signal whether content is valid.
  - Index: Selects the set.
- Block offset:
  - Selects data within block.
- Higher associativity means:
  - Less index bits.
  - More tag bits

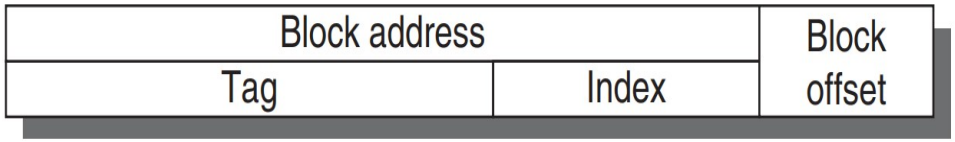


Figure B.3 The three portions of an address in a set associative or direct-mappe cache. The tag is used to check all the blocks in the set, and the index is used to selec the set. The block offset is the address of the desired data within the block. Fully asso ciative caches have no index field.

Block Identification (Cont.)

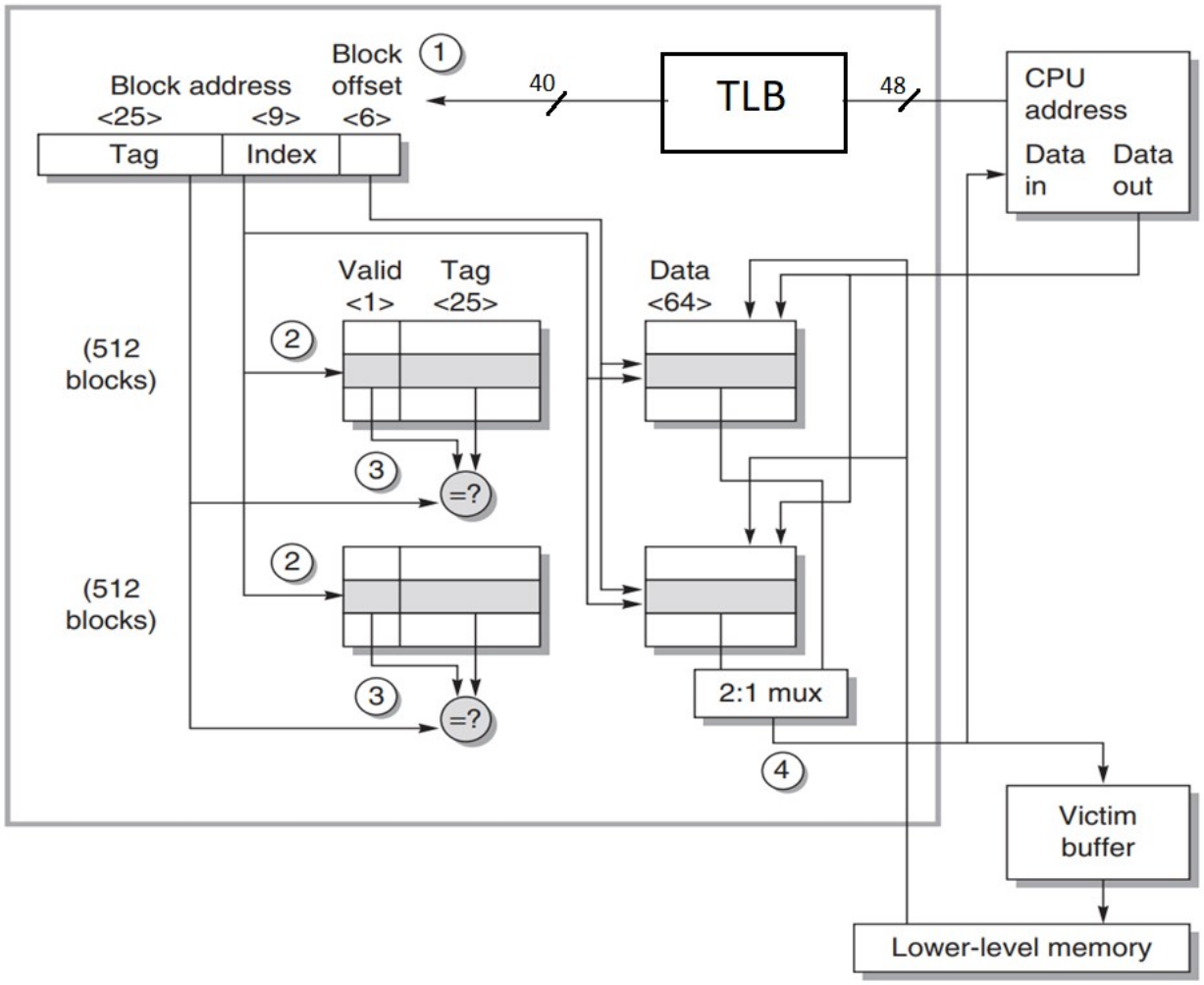


Figure B.5 **The organization of the data cache in the Opteron microprocessor.** The 64 KB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache.

Memory Hierarchy Design

Cache Memory Operations:
Block Replacement

Block Replacement

Random
- Easy to implement.

LRU: Least Recently Used
- Increasing complexity as associative increases.

FIFO
- Approximates LRU with a lower complexity.

	Associativity								
	Two-way			Four-way			Eight-way		
ze	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
6 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
4 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
6 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

gure B.4 Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out placement for several sizes and associativities. There is little difference between LRU and random for the largest ze cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller che sizes.

Memory Hierarchy Design

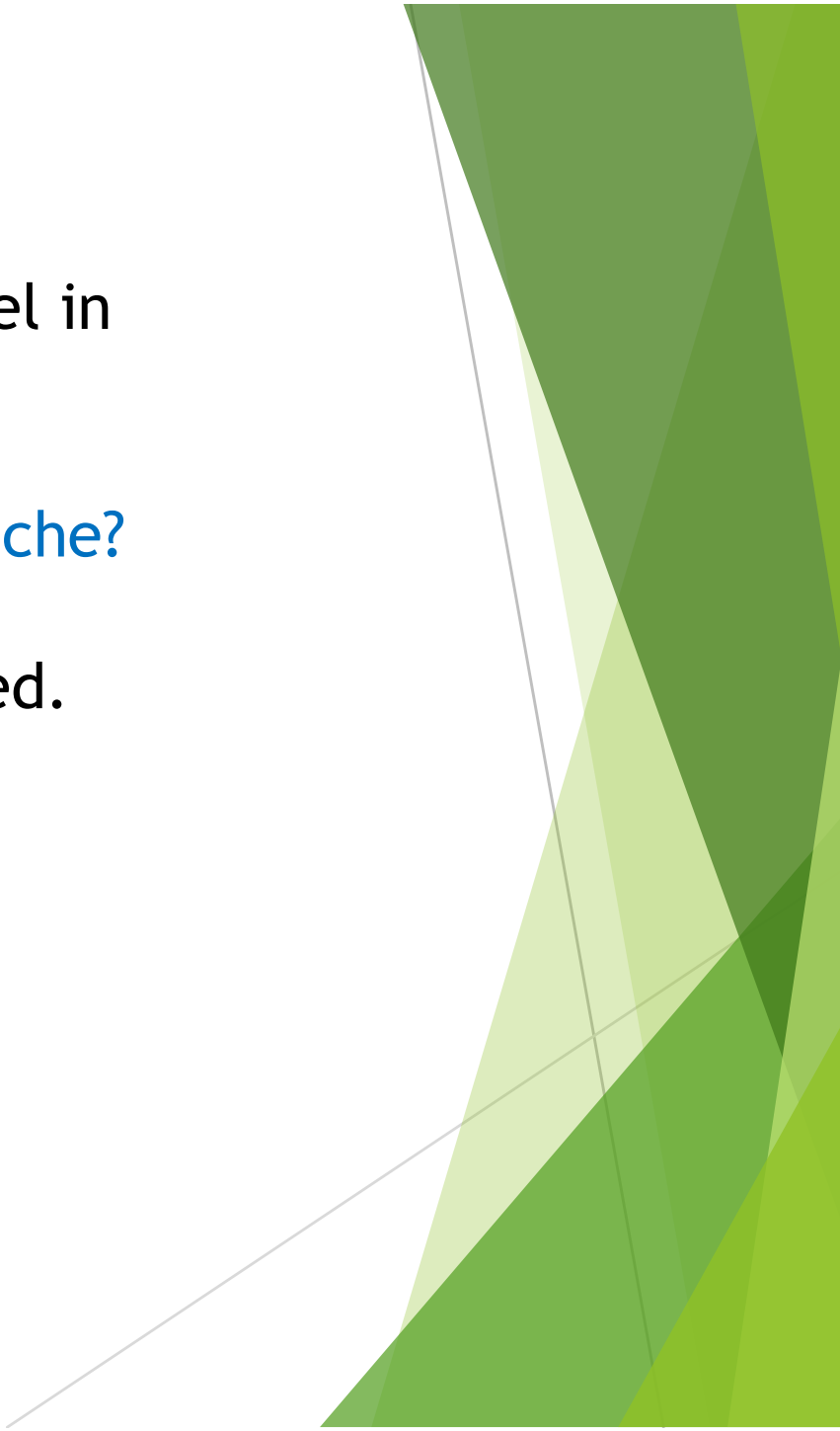
Cache Memory Operations:
Write Strategy

Write Strategy

- Write Through
  - All writes sent to lower level memory.
  - Easy to implement.
  - Simplifies data coherency (all levels has same copy)
  - Performance issues
- Write-back
  - Write hits do not go to lower level memory.
  - Writes at a speed of cache memory
  - uses less memory bandwidth, making write-back attractive in multiprocessors
  - Power efficient, making it attractive for embedded applications
  - Propagation and serialization problems.
  - More complex.

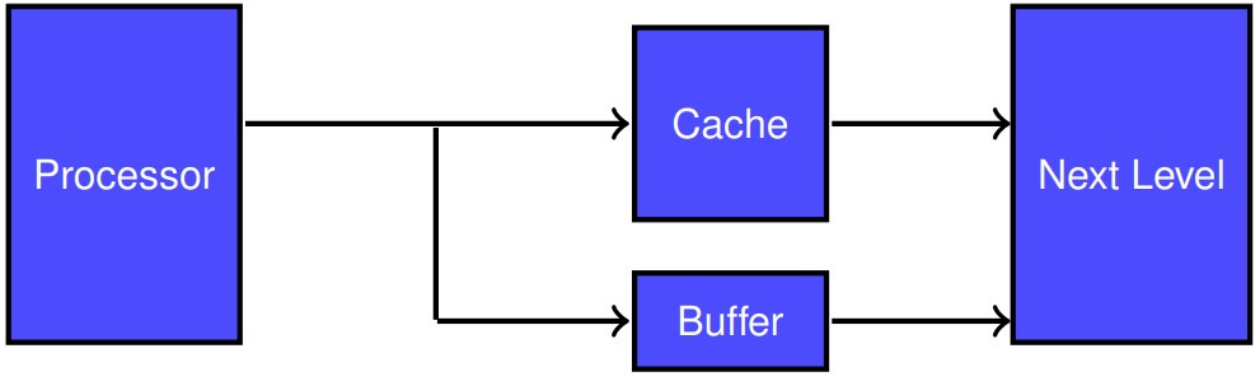
Write Strategy (Cont.)

- Where is write done?
  - write-through: In cache block and next level in memory.
  - write-back: Only in cache block.
- What happens when a block is evicted from cache?
  - write-through: Nothing else.
  - write-back: Next level in memory is updated.
- Debugging:
  - write-through: Easy.
  - write-back: Difficult.
- Miss causes write?
  - write-through: No.
  - write-back: Yes.
- Repeated write goes to next level?
  - write-through: Yes.
  - write-back: No.



Write Strategy (Cont.)

- Write Stall
  - Processor must wait for writes to complete during write-through
- Write Buffer / Victim Buffer
  - Allows the processor to continue as soon as the data are written to the buffer
  - Thereby overlapping processor execution with memory updating.
  - Reduces write stalls



Write Strategy (Cont.)

- Write Miss
  - The block to be updated is absent in the cache
  - Data is not needed by the CPU, like read miss
  - Two alternative policies, shown bellow:
  - Write Allocate
    - block is allocated on a write miss, followed by a write hit
    - natural option, write misses act like read misses.
  - No Write Allocate
    - the block is modified only in the lower-level memory
    - do not affect the cache.
- Either write-miss policy can be used with write-through or write-back.
- Normally, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache.
- Write-through caches often use no-write allocate. Because, subsequent writes to that block still go to the lower-level memory. (so what's to be gained?)

Write Strategy (Cont.)

xample Assume a fully associative write-back cache with many cache entries that starts empty. Below is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits since 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Memory Hierarchy Design

Separate vs Unified Cache

Separate vs Unified Cache

- Single unified cache supplying both instruction and data can be a bottleneck.
  - For example, when a load or store instruction is executed, both data and instruction words required simultaneously.
  - Hence, a single cache would present a structural hazard for loads and stores, leading to stalls.
- Solution: Separate cache dedicated to instructions and data
- Can have separate ports for both, thereby doubling the bandwidth between the memory hierarchy and the processor.
- Offer the opportunity to optimize each cache separately: Different capacities, block sizes, and associativities may lead to better performance.
- Separate caches are found in most recent processors, including Intel Core i
- Drawback: Fixes cache space devoted to each type

Separate vs Unified Cache (Cont.)

Size (KB)	Instruction cache	Data cache	Unifie cache
8	8.16	44.0	63.0
16	3.82	40.9	51.0
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.30	35.3	36.2
256	0.02	32.6	32.9

Figure B.6 Miss per 1000 instructions **for instruction, data, and unified caches of di ferent sizes.** The percentage of instruction references is about 74%. The data are fo two-way associative caches with 64-byte blocks

Separate vs Unified Cache (Cont.)

Example Which has the lower miss rate: a 16 KB instruction cache with a 16 KB data cache or a 32 KB unified cache? Use the miss rates in Figure B.6 to help calculate the correct answer, assuming 36% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 200 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of Chapter 3, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Answer First let's convert misses per 1000 instructions into miss rates.

Since every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate}_{16\text{ KB instruction}} = \frac{3.82/1000}{1.00} = 0.004 = \frac{\text{number of instr. misses}}{\text{no. of access to instr. cache}} = 3.82/1000$$

Since 36% of the instructions are data transfers, the data miss rate is

$$\text{Miss rate}_{16\text{ KB data}} = \frac{40.9/1000}{0.36} = 0.114 = \frac{\text{no of data misses}}{\text{no of access to data cache}} = 40.9/360$$

Separate vs Unified Cache (Cont.)

ne unified miss rate needs to account for instruction and data accesses:

$$\text{iss rate}_{32\text{ KB unified}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

43.3/1360

s stated above, about 74% of the memory accesses are instruction references.
nus, the overall miss rate for the split caches is

$$4\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

us, a 32 KB unified cache has a slightly lower effective miss rate than two
KB caches.

Important Formulas

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

Separate vs Unified Cache (Cont.)

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&= \%\ \text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty})\\&\quad + \%\ \text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&= 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200)\\&= (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.188 = 7.52\\&\text{Average memory access time}_{\text{unified}}\\&= 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200)\\&= (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—have a better average memory access time than the single-ported unified cache despite having a worse effective miss rate.

Memory Hierarchy Design

Case Study (AMD Opteron)

Case Study (AMD Opteron)

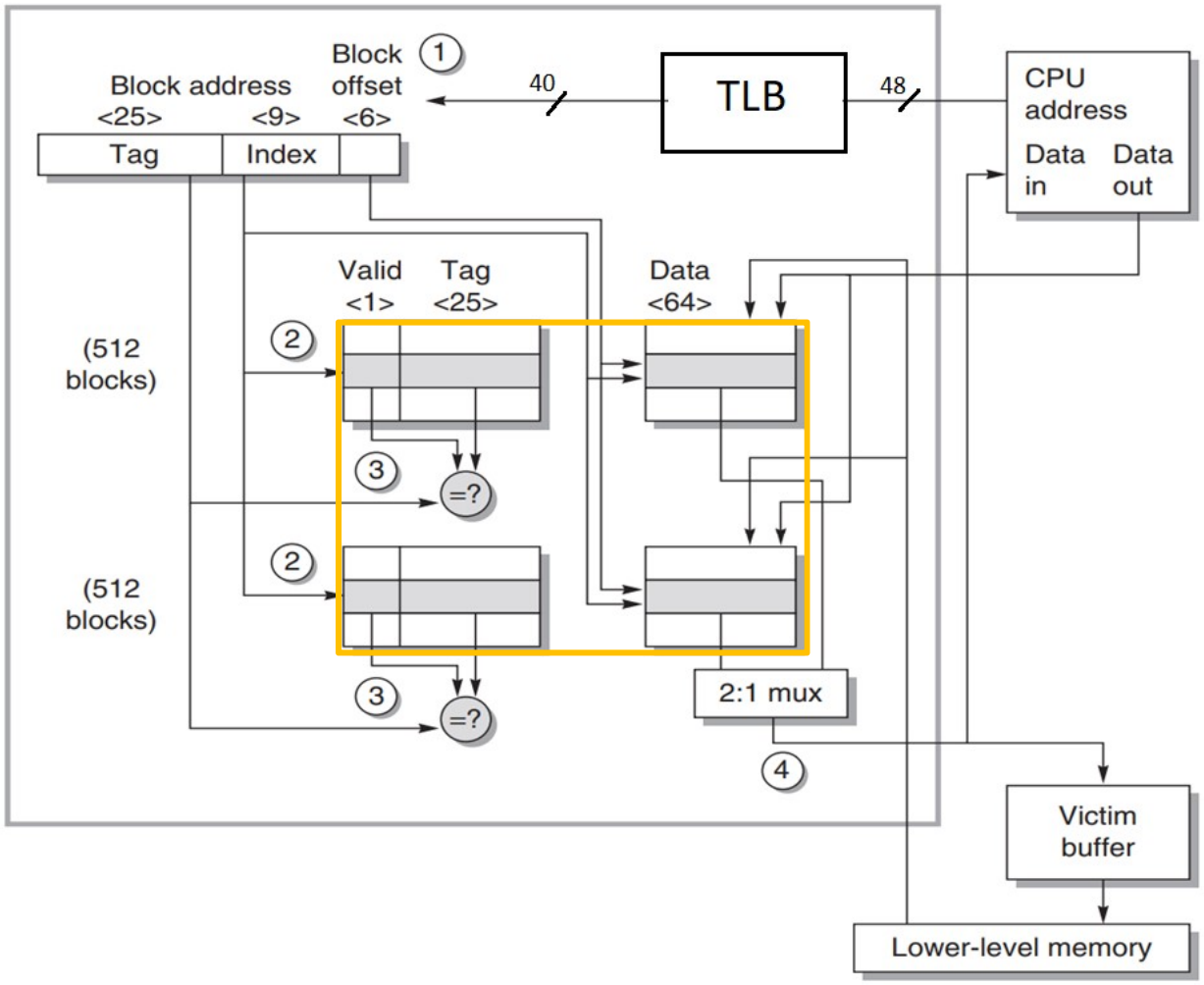


Figure B.5 **The organization of the data cache in the Opteron microprocessor.** The 64 KB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache.

Case Study (AMD Opteron) Cont.

- Cache Size: A 64 KB instruction and a 64 KB data cache. Holds two groups of 4096 64 bit-words.
- Block size: 64-bytes
- Placement: two-way set associative
- Replacement: least recently used (LRU)
- Write Strategy: write-back, and write allocate on a write miss.

Case Study (AMD Opteron) Cont.

- A 48-bit virtual address coming out of the CPU is translated into a 40-bit physical address (by the TLB)
- The 40-bit physical address coming into the cache is divided into:
  - 34-bit block address and 6-bit block offset (Block size is 2^6 =64 bytes).
  - The block address is further divided into an address tag (25 bit) and cache index (9 bit).
- The size of the index depends on cache size, block size, and set associativity

For the Opteron cache the set associativity is set to two, and we calculate the index as follows:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{65{,}536}{64 \times 2} = 512 = 2^9$$

Hence, the index is 9 bits wide, and the tag is 34 – 9 or 25 bits wide.

Case Study (AMD Opteron) Cont.

- Read Hit
  - The Opteron allows 2 clock cycles for four steps (shown in figure) to read a value from the cache.

Case Study (AMD Opteron) Cont.

- Write Hit
  - First 3 steps are same as read hit.
  - In last step the data is modified in the cache (since it uses Write-back) and the dirty bit is set. Keeps one dirty bit per block like valid bit.
  - modified data and address are sent to the victim buffer.
  - Victim buffer has space for eight victim blocks
  - victim blocks are written to the next level of the hierarchy in parallel with other cache actions.
  - If the victim buffer is full, the cache must wait, until the pending writes are completed

Case Study (AMD Opteron) Cont.

- Read Miss
  - Cache sends a signal to the processor that data yet not available,
  - A block of 64 bytes are read from the next level of the hierarchy.
    - The latency is 7 clock cycles for first 8 bytes (one word) of the block
    - 2 clock cycles per 8 bytes for the rest of the block.
  - Since the data cache is set associative, which block to replace?
    - Selects the block that was referenced longest ago (LRU),
    - So every read/write must update the LRU bit.
    - Replacing a block means updating the data, the address tag, the valid bit, and the LRU bit.

Case Study (AMD Opteron) Cont.

- Write Miss
  - Very similar to a read miss, since the Opteron allocates a block on a read or a write miss (uses Write allocate)

Metrics for Cache Optimization

- Average memory access time = Hit time + Miss rate × Miss penalty
- Reduce hit time
  - Avoid cache indexing dependent on address translation
  - – Small and simple first-level caches
  - – “way-prediction”
  - – Side effect: reduction in power consumption
- Reduce miss penalty
  - Use Multi-level caches.
  - Prioritize reads over writes
  - “Critical word first”
  - Merging write buffers
- Reduce miss rate
  - Increase block size.
  - Increase cache size.
  - Increase associativity
  - Compiler optimization
  - Side effect: reduced power

Metrics for Cache Optimization (Cont.)

- Increase cache bandwidth
  - – Pipelined caches
  - – Multibanked caches
  - – Nonblocking caches
- Reduce miss penalty and/or rate via parallelism
  - – Hardware prefetching
  - – Software and compiler prefetching
  - – Side effect: increased power due to unused data

Types of Cache Misses (3C/4C)

- *Compulsory or cold-start misses or first-reference misses* - The very first access t a block results in a miss
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict or Collision misses*— For set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) occur because a block may be discarded and later retrieved if too many blocks map to its set.
  - The idea is that hits in a fully associative cache that become misses in an *n*-way set-associative cache are due to more than *n* requests on some popular sets.
- *Coherency* misses - due to cache flushes to keep multiple caches coherent in a multiprocessor
- Compulsory misses are independent of cache size, while capacity misses decrease capacity increases, and conflict misses decrease as associativity increases.

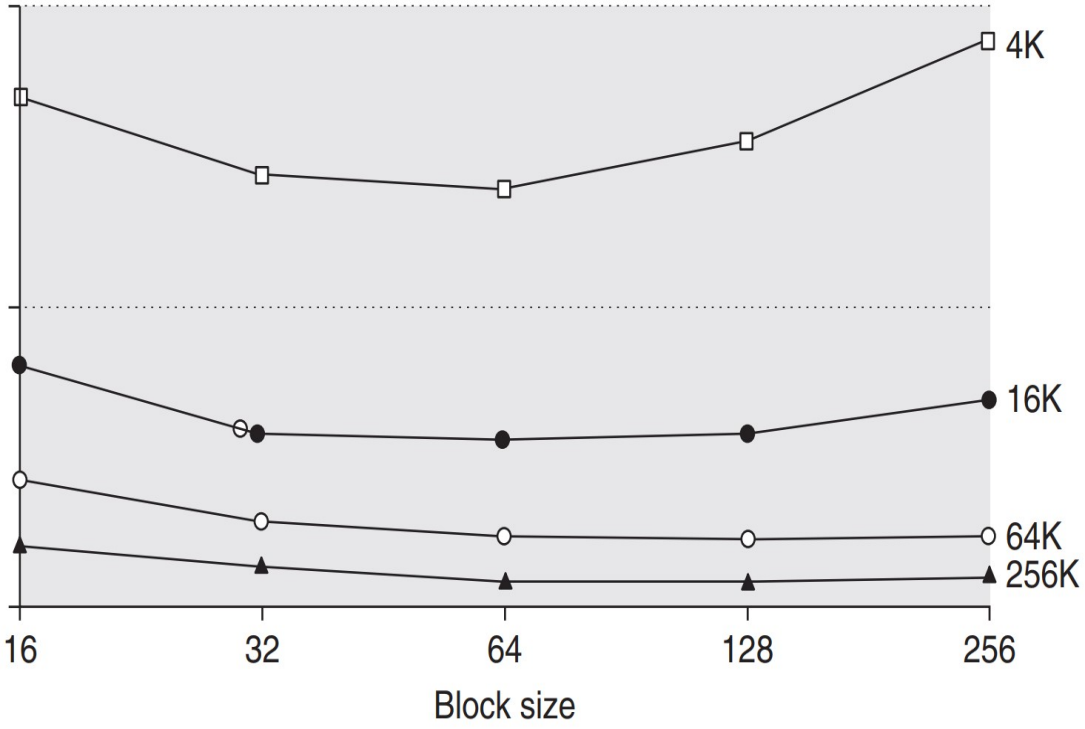
Basic Cache Optimizations

1. Increase Block Size

- Lower miss rate -> Better exploitation of spatial locality.
- Slightly reduced static power -> smaller tag
- Higher miss penalty -> Upon miss, larger blocks need to be transferred.
- Sometimes increased capacity and conflict misses -> cache has less blocks
- Need to balance:
  - Memory with high latency and high bandwidth -> Increase block size since the cache gets many more bytes per miss for a small increase in miss penalty
  - Memory with low latency and low bandwidth -> Decrease block size there is little time saved from a larger block.

Basic Cache Optimizations

1. Increase Block Size (Cont.)



0 Miss rate versus block size for five different-sized caches. Note that miss
ly goes up if the block size is too large relative to the cache size.

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

Figure B.12 Average memory access time versus block size for five differ
caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smalles
time per cache size is boldfaced.

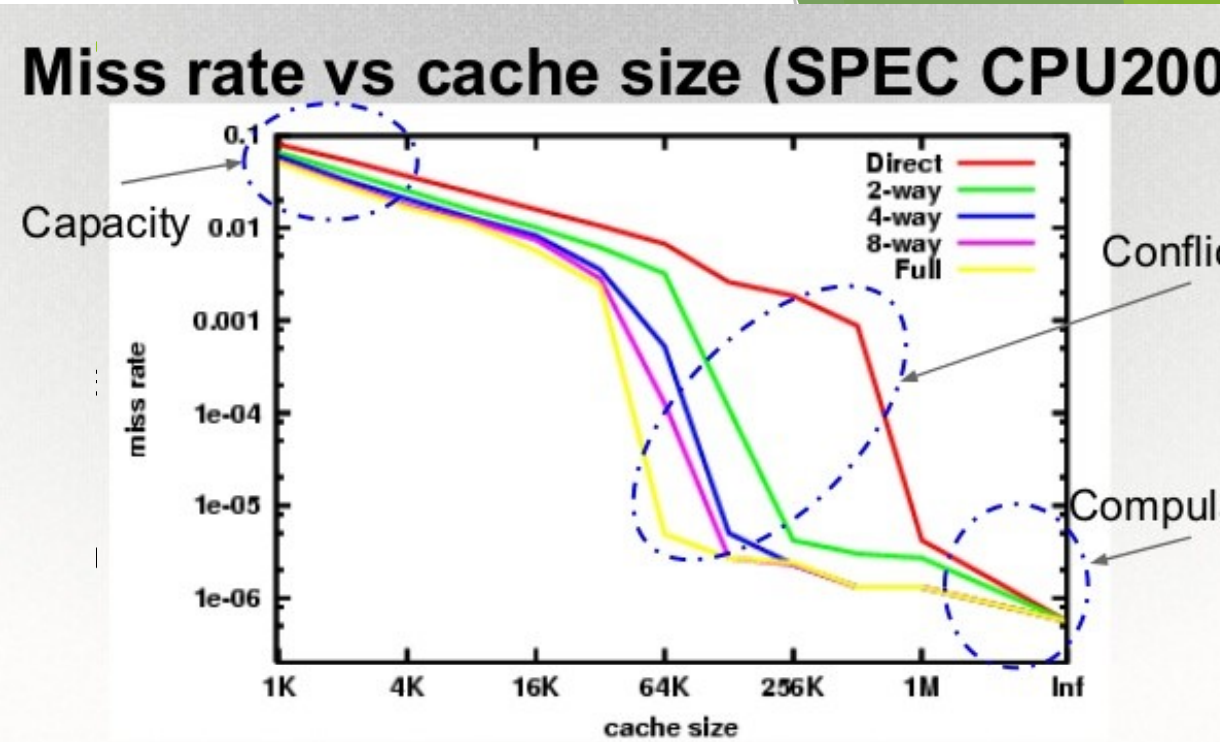
Basic Cache Optimizations

2. Increase Cache Size

- Lower miss rate -> More data fit in cache.
- It may increase hit time.
  - More time needed to find a block.
- Higher cost.
- Higher static and dynamic power consumption
- Popular in Off-chip cache
- Need to find a balance in on-chip caches

Basic Cache Optimizat

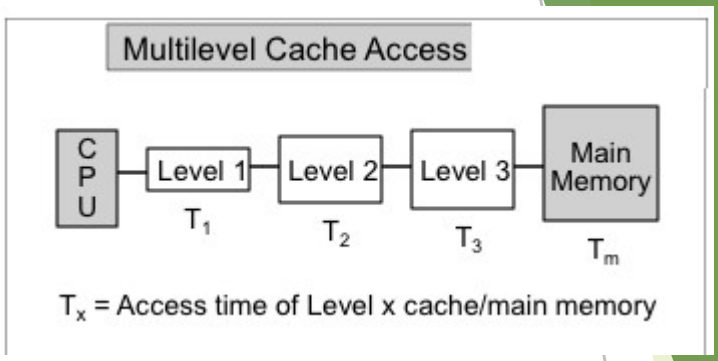
3. Increase Associativit



- Lower miss rate.
  - Less conflicts as more ways in the sar
- May increase hit time.
  - More time needed to find a block (Mc
- Increased power
- Consider Formula:
  - Average memory access time = Hit time + Miss rate × Miss penalty
- Increasing associativity reduces miss rate while increasing hit time.
  - Hence, the pressure of a fast processor clock cycle (related with reduced hit time) encourages simple cache designs (lower associativity).
- (As per the formula) for higher associativity, increased hit time can outweigh the benefits of reduced miss rate, that is, average memory access time can be increased rather than decrease.

Basic Cache Optimizations

4. Multi-Level Caches (1)



- Goal is to reduce miss penalty.
- Large performance gap between CPU and main memory.
- Caches should small and simple to be fast enough to keep pace with the processor
- But Small caches have large capacity misses and larger caches become slow.
- Solution to this conflicting case is to use **more level of caches**
- **First level** as simple, small and fast enough to match the clock cycle time of the fast CPU [low hit time]
- **2nd level** is large enough to capture many accesses that would go to main memory giving low miss penalty.

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

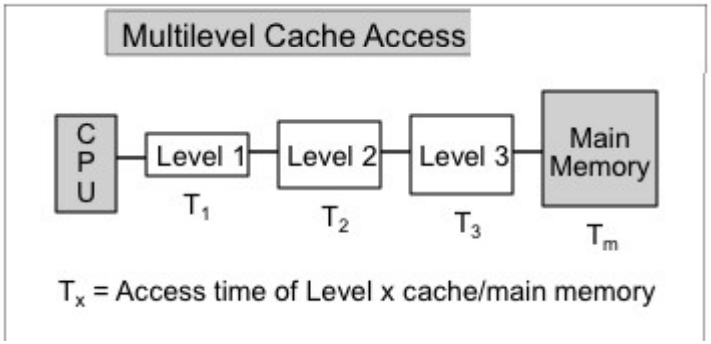
$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

Basic Cache Optimizations

4. Multi-Level Caches (2)

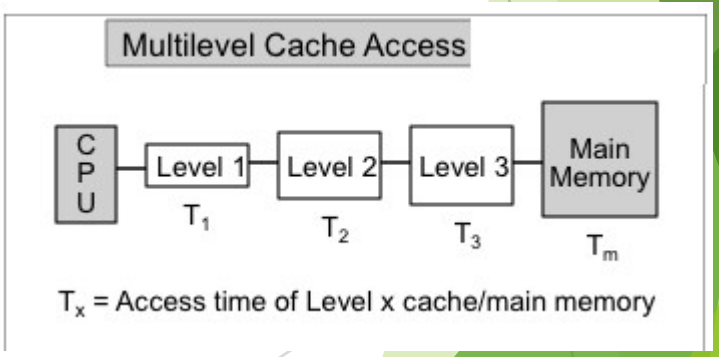
- **Local Miss Rate**: No. of misses in a cache level divided by the no. of memory access to this level
- **Global Miss Rate**: No. of misses in a cache level divided by the no. of memory access generated by the CPU



Basic Cache Optimizations

4. Multi-Level Caches (3)

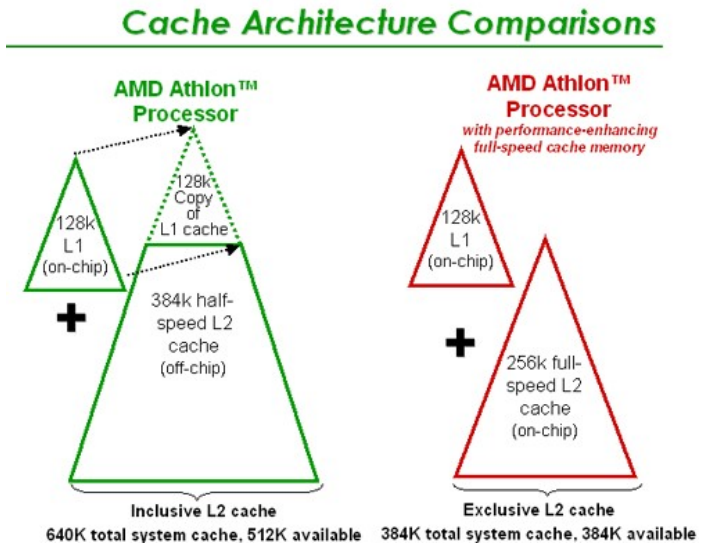
- Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time?
  - If a CPU generates 100 memory accesses out of which 80 is captured by the L1 cache and 10 by the L2 cache and 5 by the L3 cache, what will be the local and global miss rates?
    - Local miss rates for L1 and L2, respectively, is 40/1000 and 20/40
    - Global miss rate = 20/1000
  - Average memory access time = Hit time$_{L1}$ + Miss rate$_{L1}$ × (Hit time$_{L2}$ + Miss rate$_{L2}$ × Miss penalty$_{L2}$)
    =1 + 4% × (10 + 50% × 200) = 1 + 4% × 110 = 5.4 clock cycles



Basic Cache Optimizations

4. Multi-Level Caches (4)

- **Inclusive Cache:**
  - Contents of L1 is the subset of that of L2. Effective size = size of L2
  - Cache miss in L1 results a replacement of an L1 block with an L2 block
- **Exclusive Cache:**
  - Contents are different. Effective size = size of L2 + size of L1
  - Cache miss in L1 results in a swap of blocks between L1 and L2 instead of
  - AMD Opteron chip obeys the exclusion property using two 64 KB L1 caches and 1 MB L2 cache



Basic Cache Optimizations

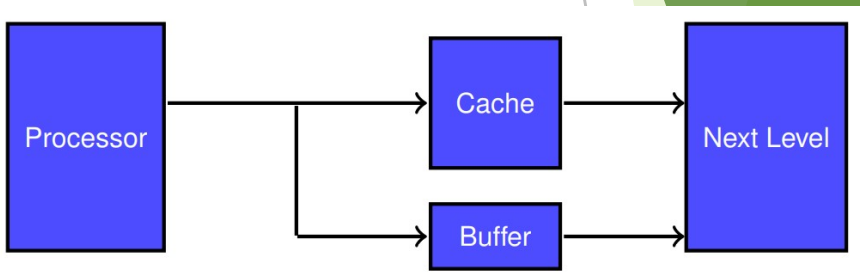
5. Prioritize Read Misses over Writes

- Goal is to reduce miss penalty. Consider a **write-through** cache with a buffer.

Look at this code sequence:

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

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in R2 always be equal to the value in R3?

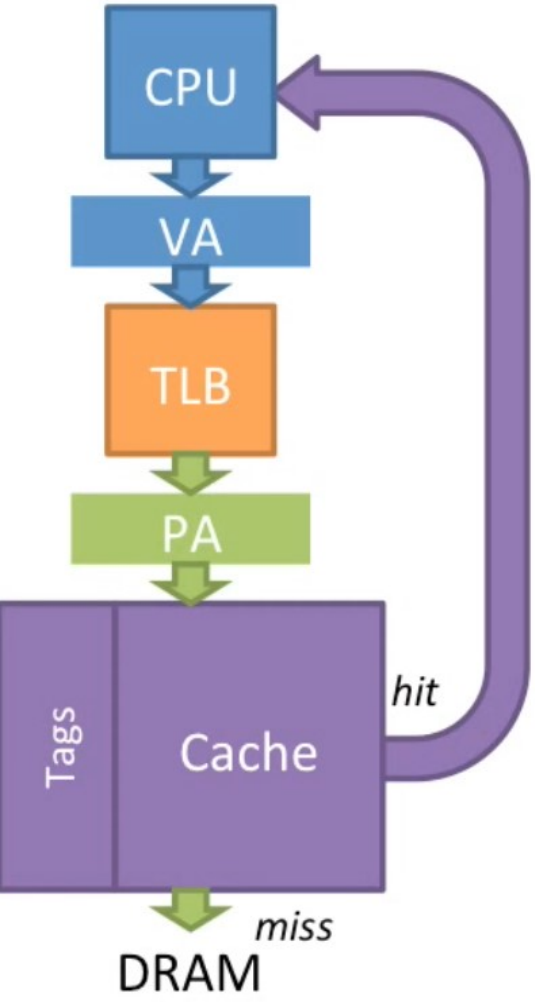


- In the above case, the answer is NO. Why?
- The simplest **un-optimized** way is for the read miss to wait until the write buffer is empty.
- **Optimized** Method: On a read miss,
  - if need to replace a dirty block,
    - copy the dirty block to a buffer (instead of directly writing to memory), replace the dirty block by reading memory, and then complete writing memory. This way the processor can finish faster.
  - Otherwise, check the contents of the write buffer, and if there is no conflict let the read miss continue. Mostly used method.
- write-back cache can be benefited.

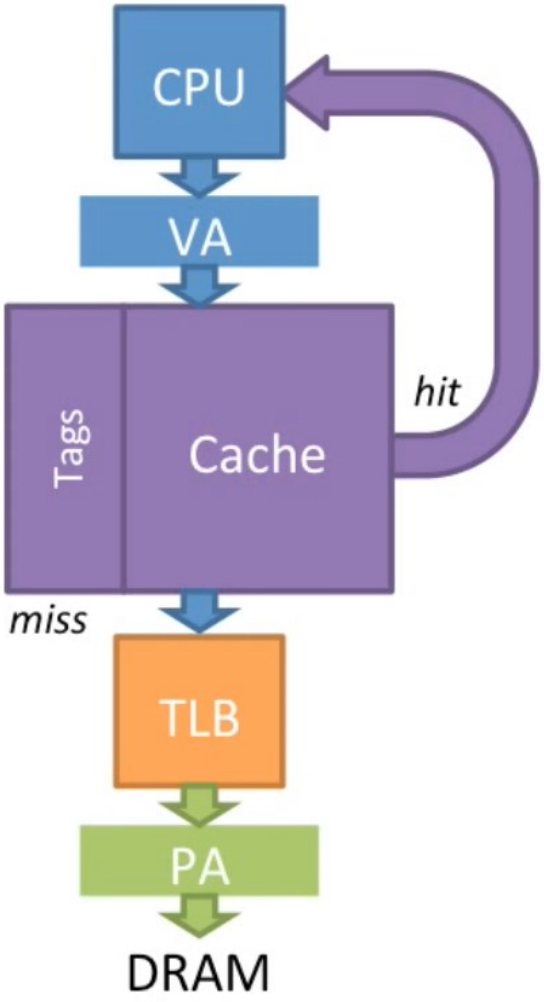
Basic Cache Optimizations

6. Avoiding Address Translation during Indexing to Reduce Hit Time

Physical Cache



Virtual Cache



Physical Cache

Slow: Must do a TLB lookup *before* accessing the cache

Virtual Cache

Fast: TLB lookups *only* when we miss in the cache

Q: Can you have two programs share a virtual cache?

- Yes. The TLB provides protection.
- No. Each needs its own TLB.
- No. The cache is virtual so there is no way to provide protection

A: No.

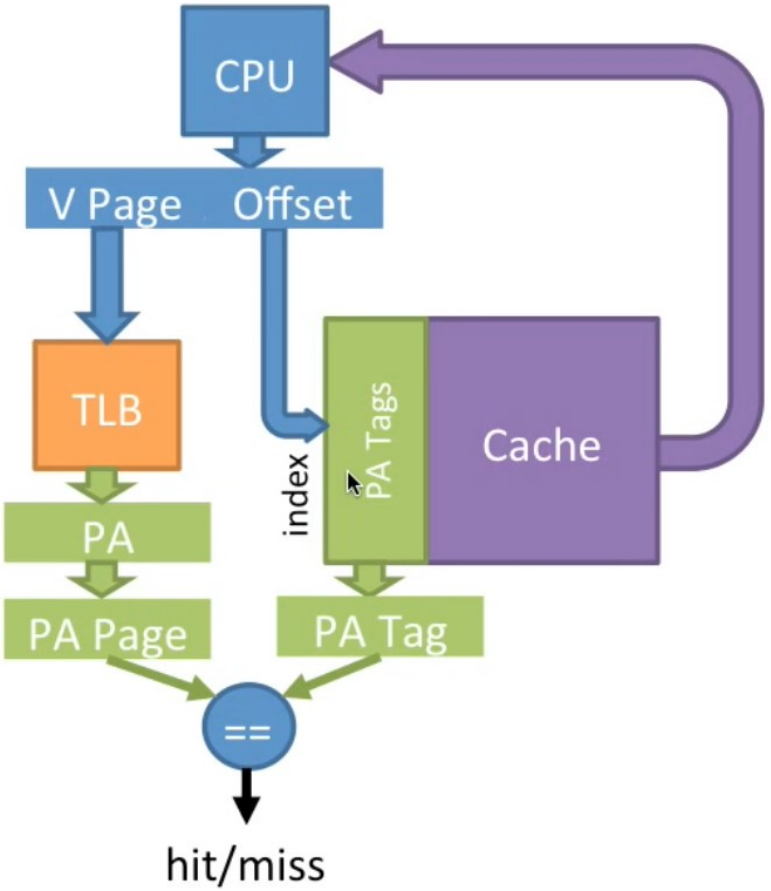
A virtual cache stores data by the virtual program address. There is no translation, so VM-based protection can't keep applications apart!

(You can have a separate bit for the process ID or you have to flush the caches when you switch programs.)

Basic Cache Optimizations

6. Avoiding Address Translation during Indexing to Reduce Hit Time

VIPT: Virtually Indexed, Physically Tagged



TLB translation and **Cache lookup** at the **same time**

- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as the cache

Safe: Cache hit only on PA match

- **But**, can only use non-translated bits to index cache (limit on how large the cache can be)
- Most processors use VIPT for L1 caches today

Basic Cache Optimizations

6. Avoiding Address Translation during Indexing to Reduce Hit Time

Q: With 4kB pages, how many bytes can a direct-mapped (1-way) VIPT cache store?

- 4kB
- 4096kB
- Unlimited
- Need to know the cache line size

A: 4kB

We can only use the page offset bits (12 bits for 4kB pages) to index into the cache. So the index can only address 12 bits of address, or 4kB of data.

Note: if we increase associativity, we can make it larger! E.g., 8-way cache would give us 32kB of cache size. (Each way uses the same index.)

Virtual Memory

Topics

- **Three memory problems:**
  - Not enough RAM
  - Holes in our address space
  - Programs writing over each other
- **What is virtual memory?**
  - Indirection
  - How does it solve the problems?
  - Page Tables and Translation
- **Implementing virtual memory**
  - Where do we store the page tables?
  - Making translation fast
- **Virtual memory and caches**

Virtual Memory

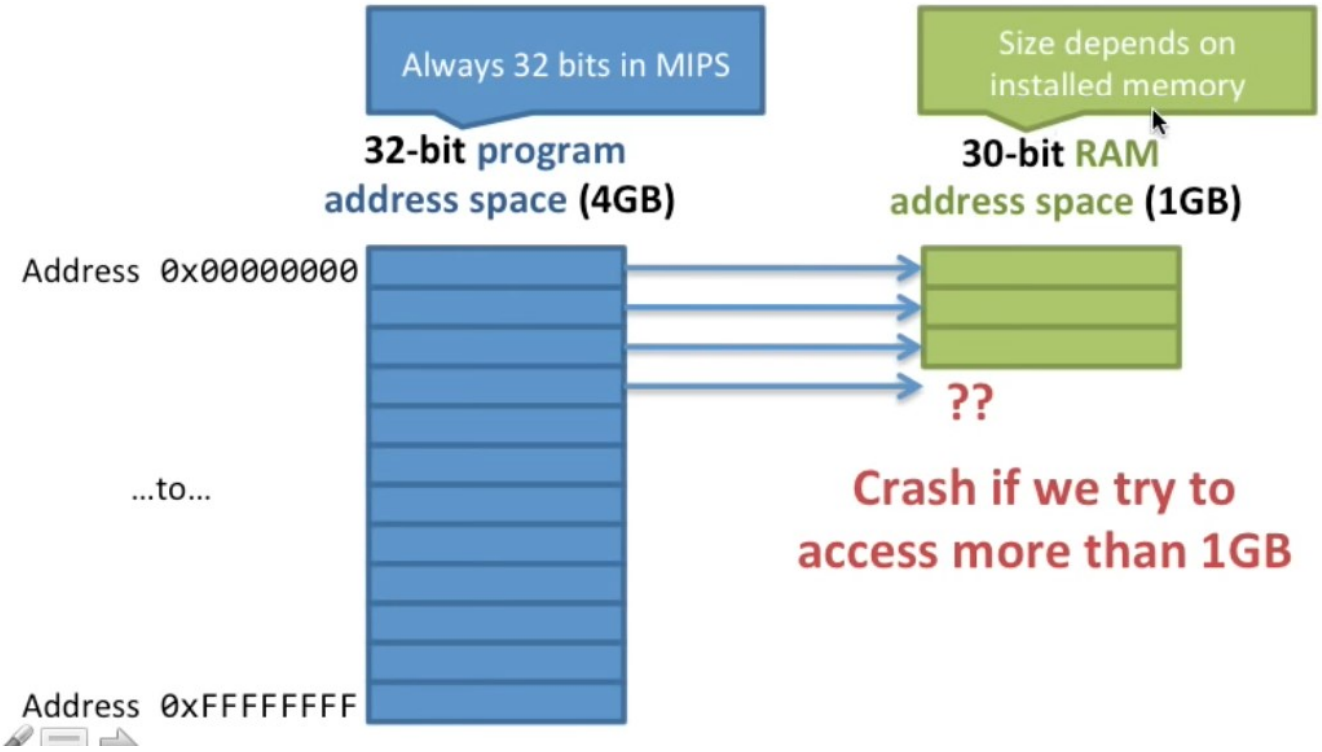
MIPS Computer System

Designer	MIPS Technologies, Imagination Technologies
Bits	64-bit (32 → 64)
Introduced	1985; 36 years ago
Version	MIPS32/64 Release 6 (2014)
Design	RISC
Type	Register-Register
Encoding	Fixed
Branching	Compare and branch
Endianness	Bi
Page size	4 KB
Extensions	MDMX, MIPS-3D
Open	Partly. The R12000 processor has been on the market for more than 20 years and so cannot be subject to patent claims. Therefore, the R12000 and older processors are fully open.
	Registers
General purpose	32
Floating point	32

Virtual Memory

Problem 1: If Not have Enough Memory

- MIPS gives each program its own **32-bit address space**
- Programs can access any byte in their **32-bit address space**
- What if you don't have 4GB (2^{32} bytes) of memory?



Q: How much memory can you access with a 32 bit address?

- 2^{30} bytes = 1GB
- 2^{32} bytes = 4GB
- 2^{32} words = 16GB

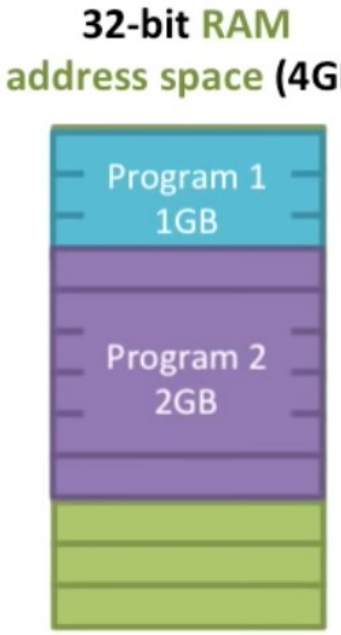
A: 2^{32} bytes = 4GB

A 32-bit address space gives you (theoretically) 4GB of memory you can address. In practice the OS reserves some if it so it is closer to 2GB of usable space.

Virtual Memory

Problem 2: Hoes in Address Space (1)

- How do programs share the memory?
- Where do we put them?

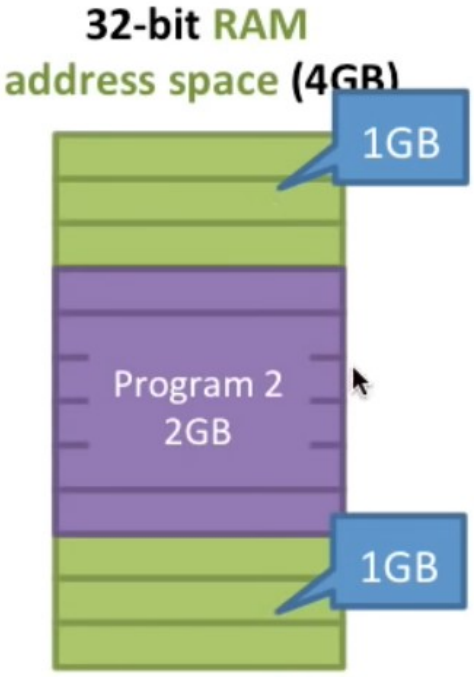


1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)

Virtual Memory

Problem 2: Hoes in Address Space (2)

- How do programs share the memory?
- Where do we put them?

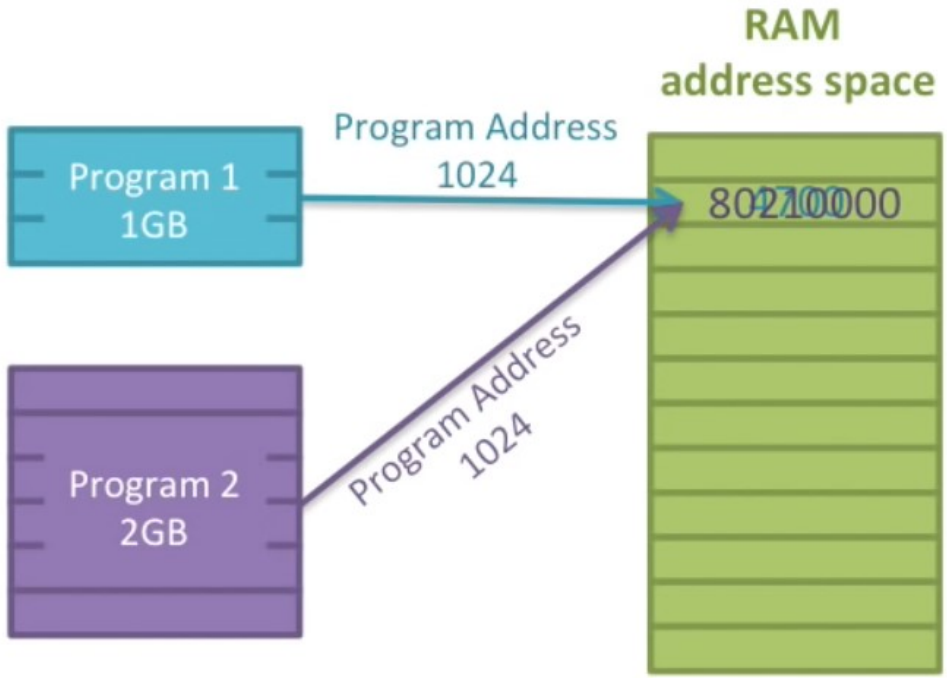


1. Programs 1 and 2 fit
(They use 3GB of memory, leaving **1GB free**.)
2. Quit Program 1
(Program 2 uses 2GB of memory, leaving **2GB free**.)
3. Can't run Program 3
(Even though we have enough free space!)
Memory fragmentation

Virtual Memory

Problem 3:How do we keep program secure

- Each program can access any 32-bit memory address
- What if multiple programs access the same address?
  - `sw R2, 1024(R0)` will write to address 1024 regardless of the program that's running
- They can **corrupt or crash** each other: security and reliability



1. **Program 1** stores your bank balance at address 1024
2. **Program 2** stores your video game score at address 1204
3. **Profit? Loss? Not good!**

Virtual Memory

Problems with physical memory

- If all programs have access to the same 32-bit memory space:
  - Can **crash** if less than 4GB of RAM memory in the system
  - Can **run out of space** if we run multiple programs
  - Can **corrupt** other programs' data

- How do we solve this?
  - Key to the problem: "**same memory space**"
  - Can we give each program it's **own *virtual* memory space**?
  - If so, we can:
    - Separately **map** each **program's memory space** to the **RAM memory** space
    - (and even move it to disk if we run out of memory)

Mapping gives us **flexibility** in how w
use the physical RAM memory

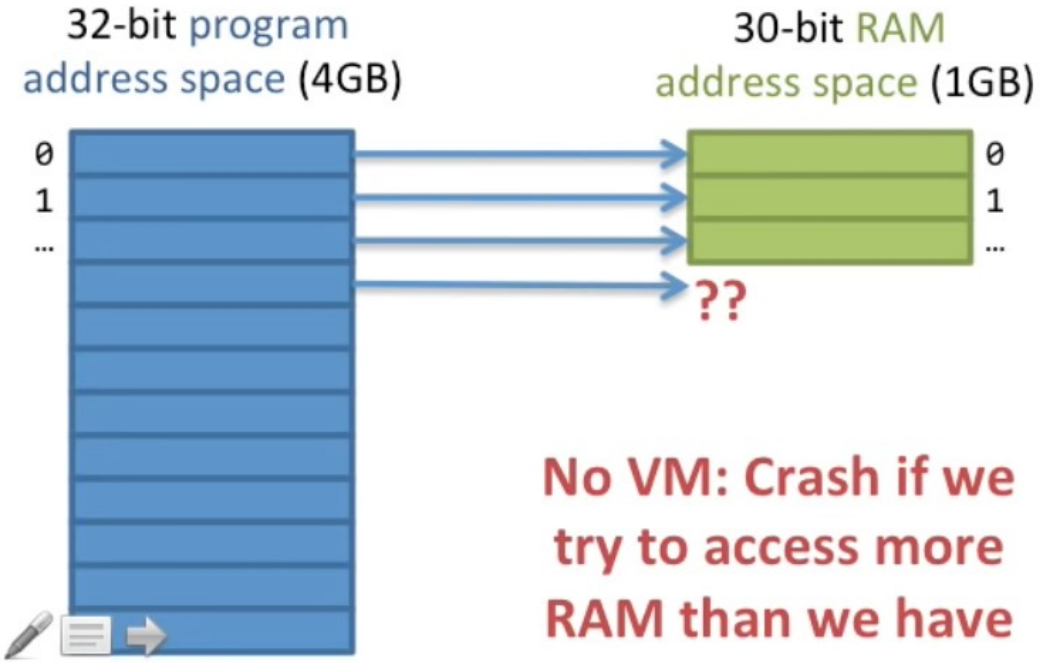
What is a Virtual Memory

Virtual Memory is a layer of Indirection

Virtual memory takes **program addresses** and **maps** them to **RAM addresses**

Without Virtual Memory

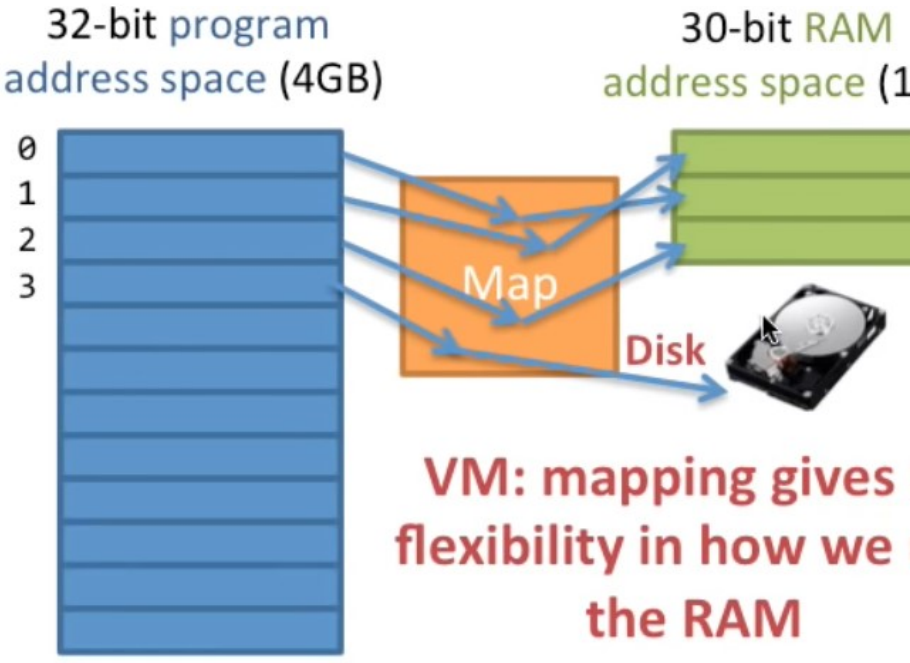
Program Address = RAM Address



No VM: Crash if we try to access more RAM than we have

With Virtual Memory

Program Address Maps to RAM Address

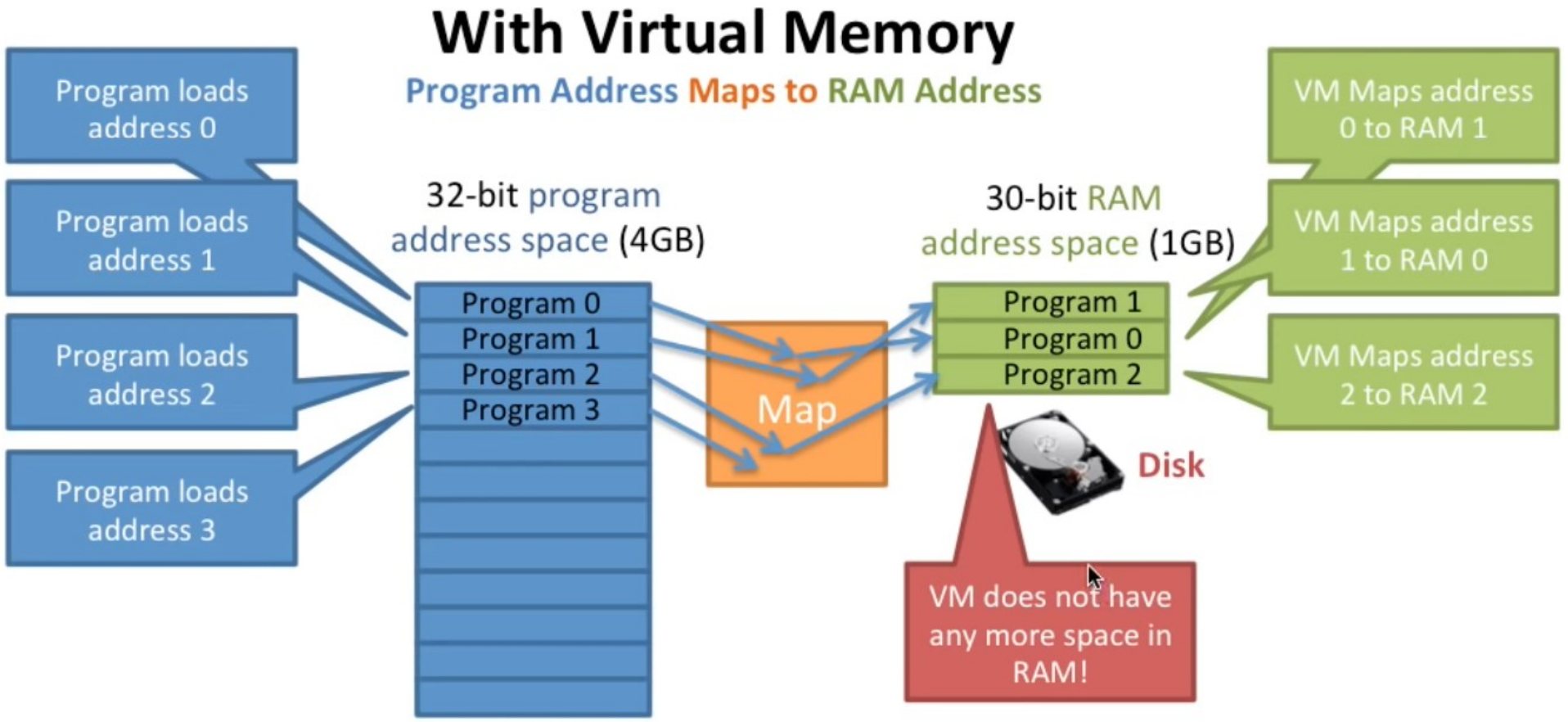


VM: mapping gives flexibility in how we the RAM

Virtual Memory

Solving Problem 1: If Not have Enough Memory (1)

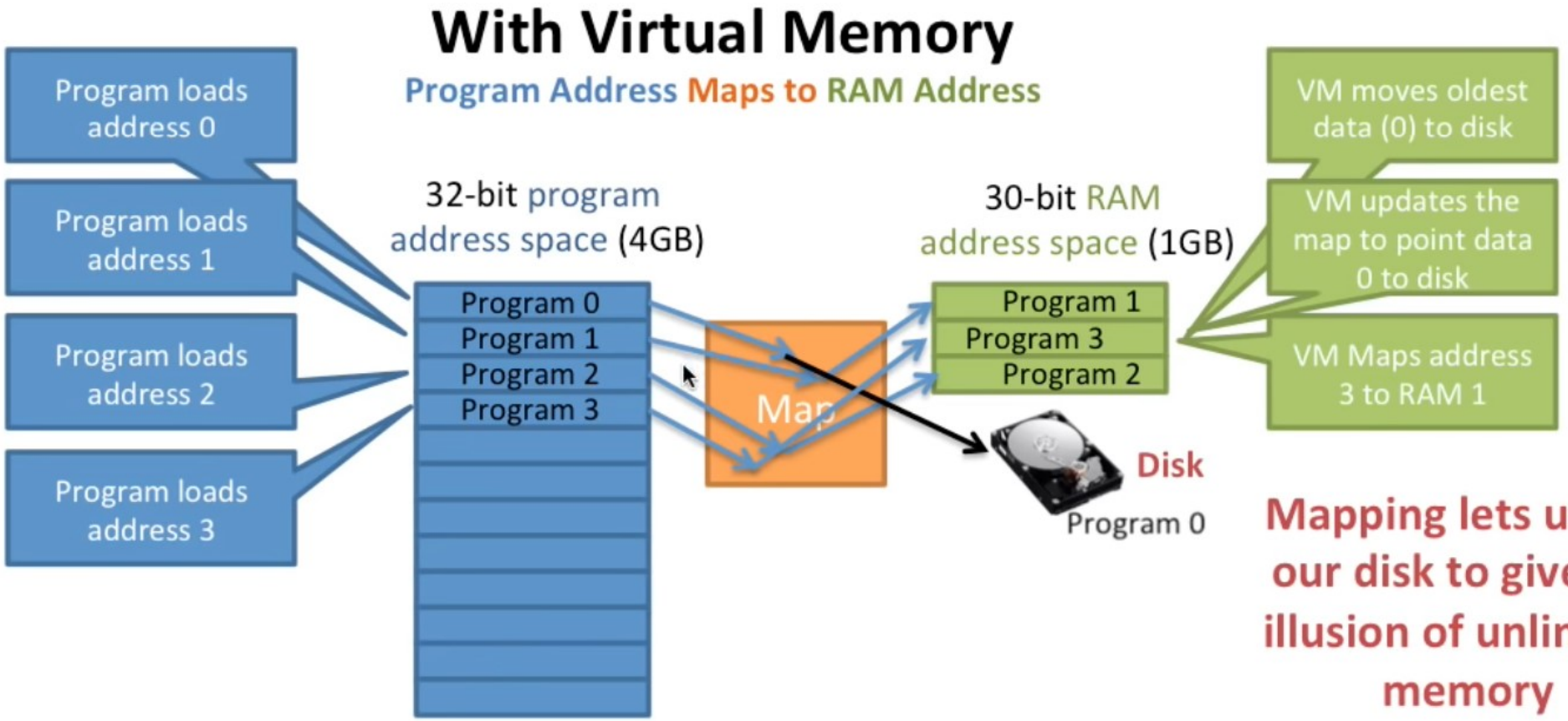
- **Map** some of the **program's address space** to the **disk**
- When we need it, we bring it into memory



Virtual Memory

Solving Problem 1: If Not have Enough Memory (2)

- **Map** some of the **program's address space** to the **disk**
- When we need it, we bring it into memory



Virtual Memory

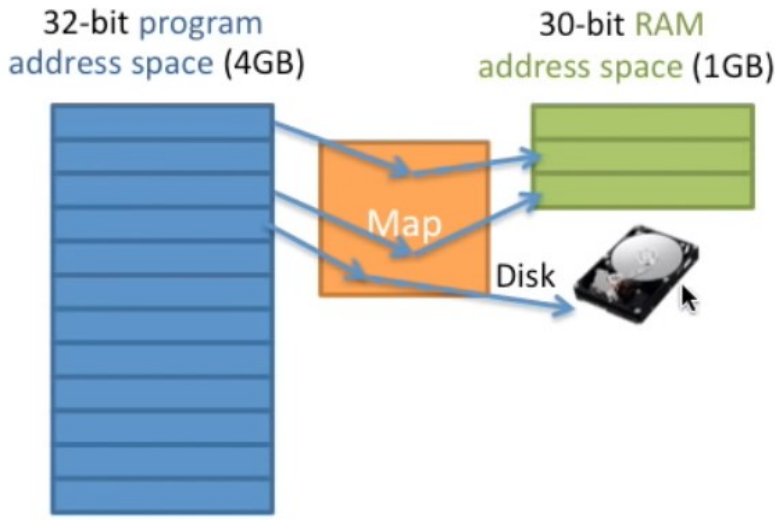
VM and Performance

Q: What is going to happen to the program performance when the data it needs is on the disk and not in memory?

- Better performance: we can use more memory than we have
- Nothing: mapping to memory or disk is just as easy
- Worse performance: reading from disk is slower than RAM

A: Worse performance: reading from disk is slower than RAM

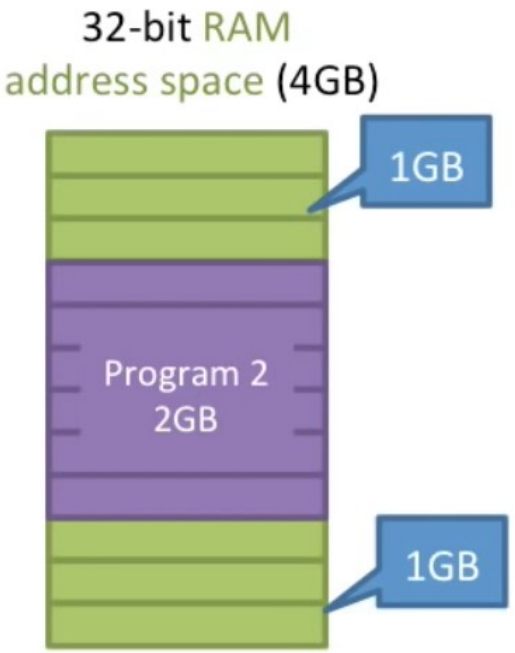
Remember that disks are 1000x slower than RAM. Any time you can't fit your data in memory and have to go to disk you pay a HUGE performance penalty! (This is why buying more RAM makes your computer faster.)



Virtual Memory

Solving Problem 2: Holes in address space (1)

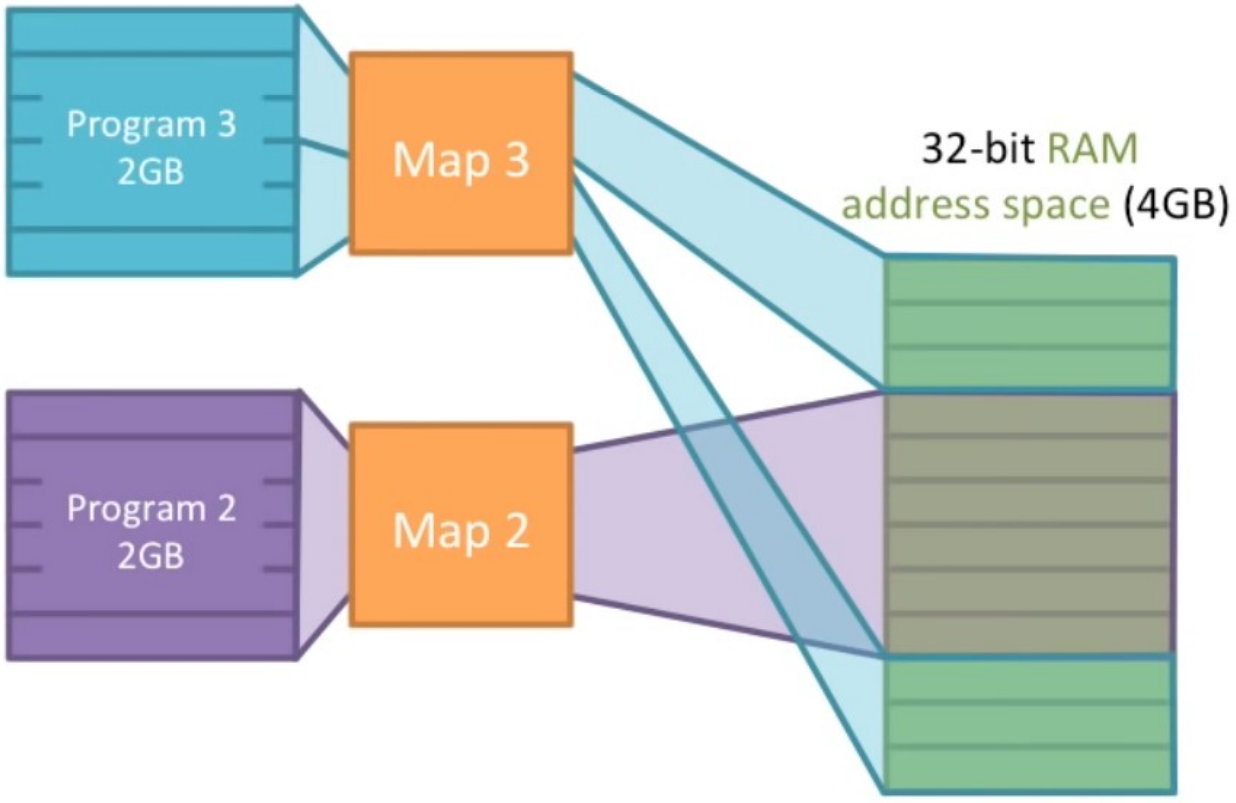
- How do we use the holes left when programs quit?
- We can **map** a program's addresses to RAM addresses however we like



Virtual Memory

Solving Problem 2: Holes in address space (2)

- How do we use the holes left when programs quit?
- We can **map** a program's addresses to RAM addresses however we like



With Virtual Memory

Program Address Maps to RAM Addres

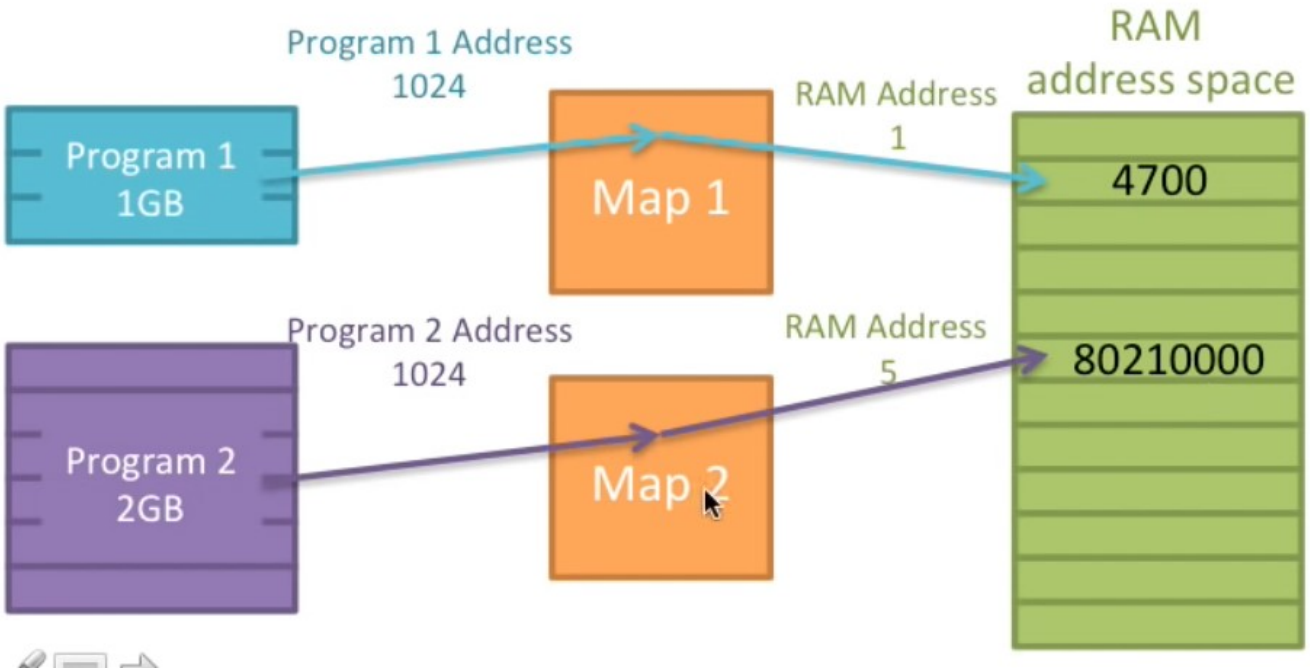
Each program has its own mapping.

Mappings lets us put our program data wherever we want in the RAM.

Virtual Memory

Solving Problem 3:How do we keep program secur

- Program 1's and Program 2's addresses map to different RAM addresses
- Because each program has its own address space, they cannot access each other's data: security and reliability!



With Virtual Memory

Program Address Maps to RAM Addre

1. Program 1 stores your bank bal
at address 1024
1B. VM maps it to RAM address 1

2. Program 2 stores your video ga
score at address 1204
2B. VM maps it to RAM address 5

3. Neither can touch the other's

Virtual Memory

Is program isolation always good?

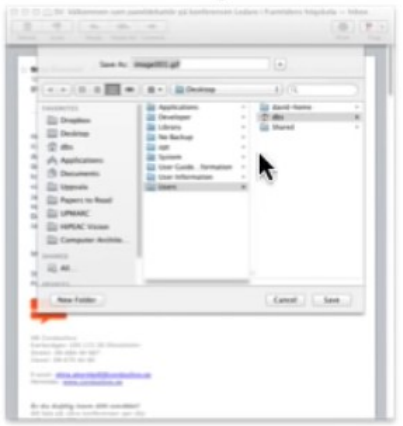
Q: Virtual Memory lets us isolate programs so they can't share/corrupt data. What is a downside of complete isolation?

- Programs can't corrupt each other
- Programs can't share data with each other
- Programs use more space because they have their own address space
- Programs are slower because they always have to check the disk for data

A: Programs can't share data with each other
A lot of data is shared between programs. (Think about shared resources: fonts, graphics, scrollbars, icons and shared functionality: libraries, open/save dialog boxes, etc.)

However, we can use the same mapping to allow programs to share data by simply having their maps point to the same data! (Isn't indirection great?)

Save dialog in Mail



Save dialog in Browser

Save dialog is **shared** across the whole syste

Virtual Memory

How virtual memory works?

- **Basic idea: separate memory spaces**
  - **Virtual memory**: what the **program** sees
    - e.g., ld R4, 1024(R0) accesses virtual address R0+1024=1024
  - **Physical memory**: the **physical RAM** in the computer
    - e.g., if you have 2GB of RAM installed, you have physical addresses 0 to 2^{31}-1

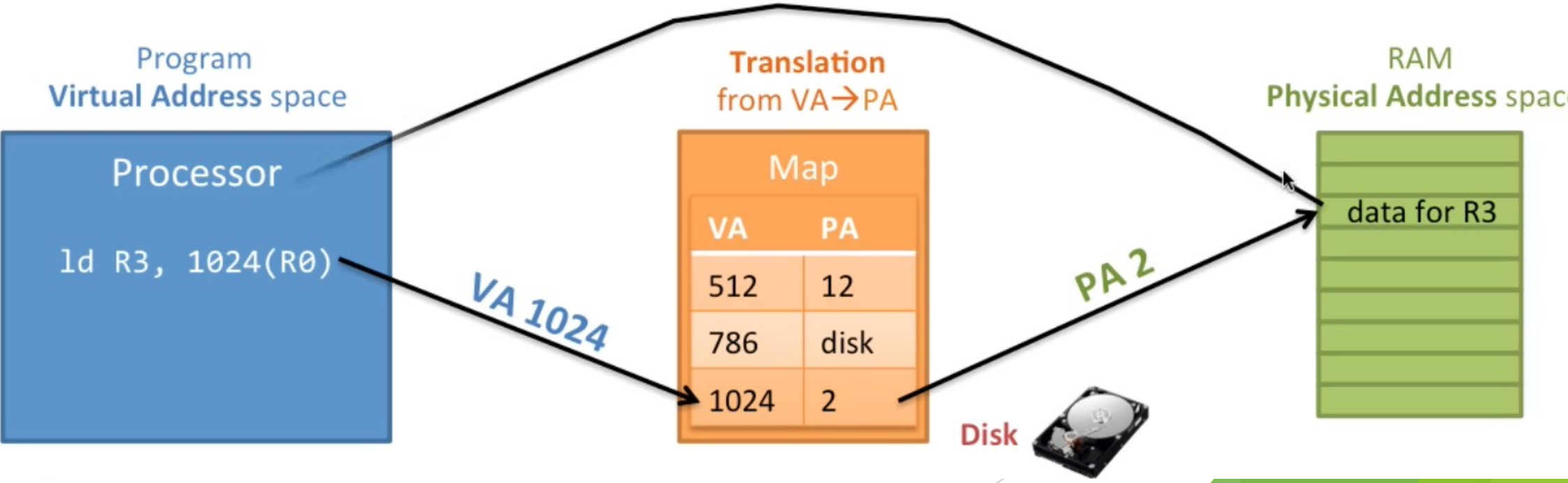
- **Virtual Addresses (VA)**
  - What the program uses
  - In MIPS, this is the full 32-bit address space: 0 to 2^{32}-1
- **Physical Addresses (PA)**
  - What the hardware uses to talk to the RAM
  - Address space determined by how much RAM is installed

Virtual Memory

Making VM Works: Address Translation (1)

How does a program access memory?

1. Program executes a load specifying a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to program

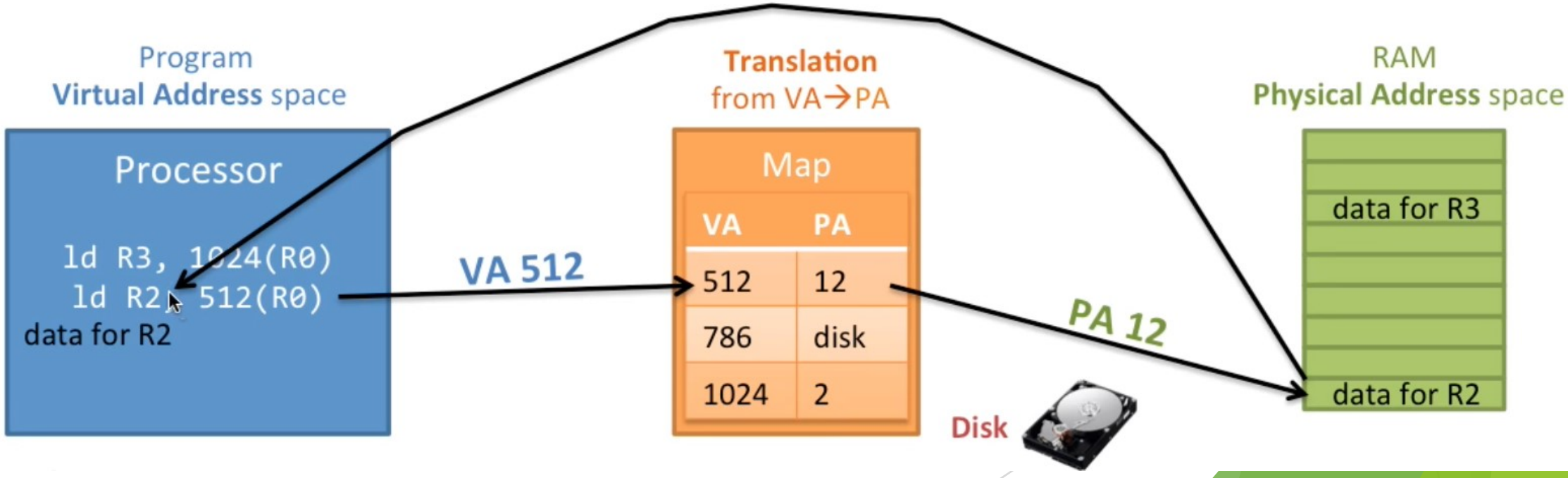


Virtual Memory

Making VM Works: Address Translation (2)

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to t program

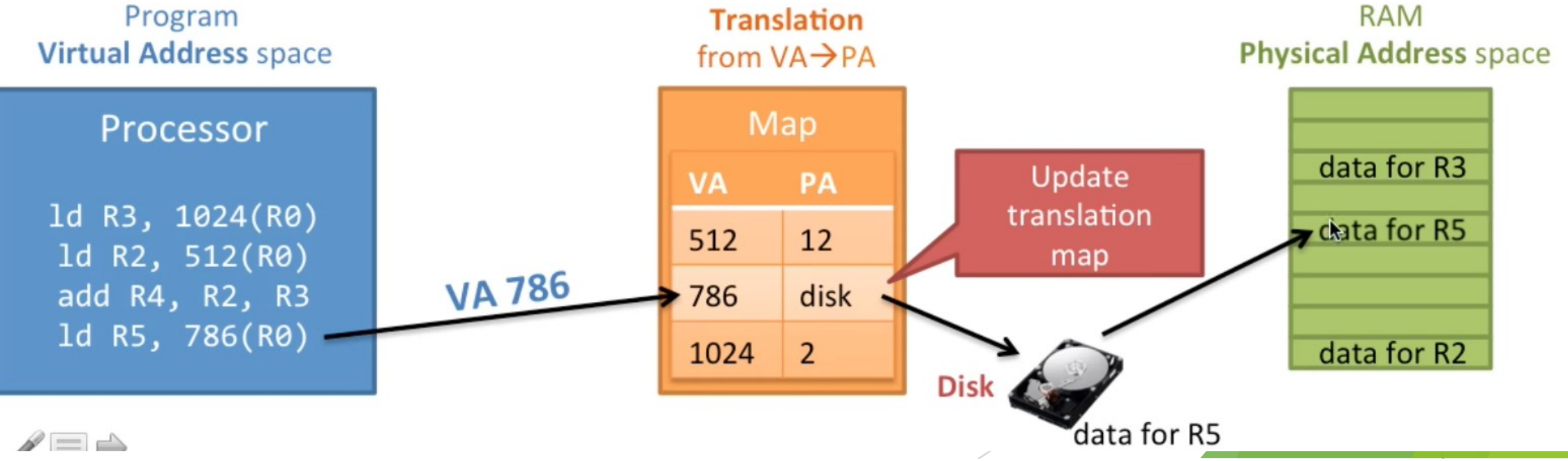


Virtual Memory

Making VM Works: Address Translation (3)

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system **loads it in from disk**)
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data to th program

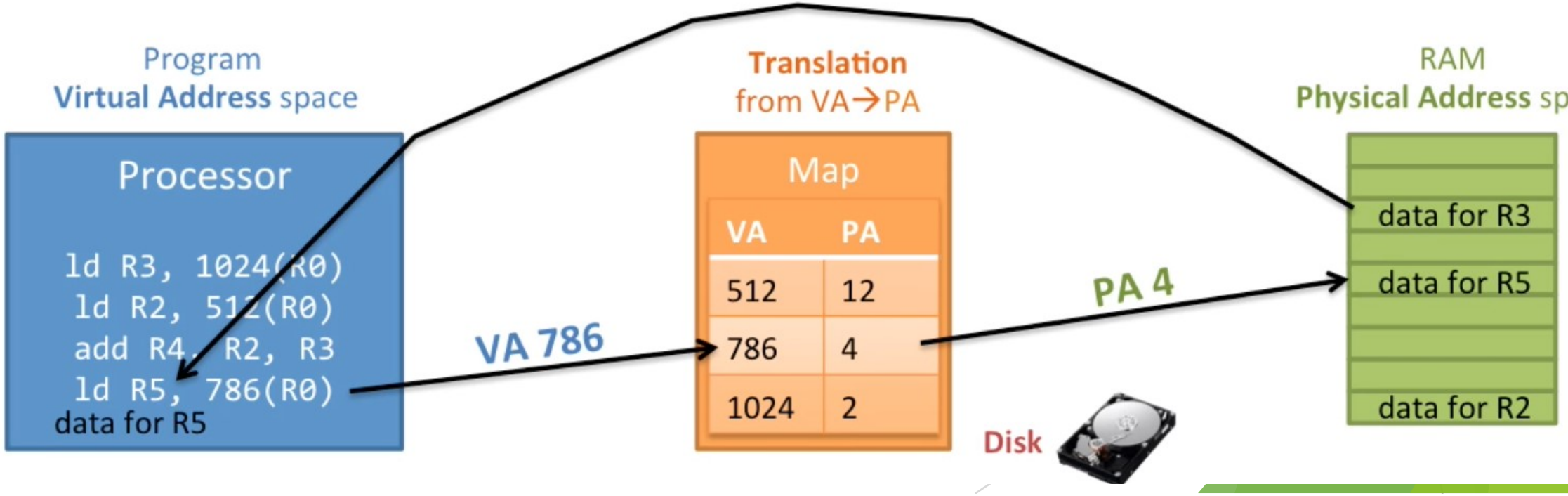


Virtual Memory

Making VM Works: Address Translation (4)

How does a program access memory?

1. Program executes a load with a **virtual address (VA)**
2. Computer **translates** the address to the **physical address (PA)** in memory
3. (If the **physical address (PA)** is not in memory, the operating system loads it in from **disk**
4. The computer then **reads the RAM** using the **physical address (PA)** and returns the data program



Virtual Memory

Page Tables

The map from **Virtual Addresses (VA)** to **Physical Addresses (PA)** is the **Page Tab**

So far we have had one **Page Table Entry (PTE)** for every **Virtual Address**

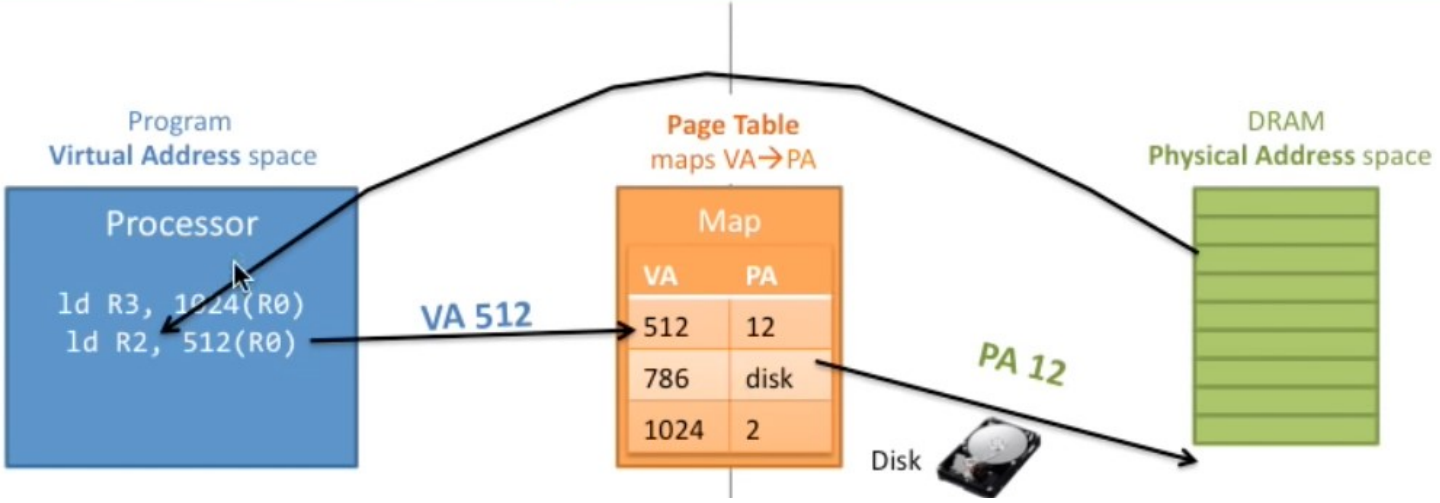
Q: How many entries do we need in our Page Table?

- 1 for every byte = 2^{32} = 4 billion
- 1 for every word = 2^{30} = 1 billion
- 1 for every register = 32

A: 1 for every word = 2^{30} = 1 billion

We have a word-aligned memory so we need to be able to address every word.

Note: each entry needs the PA which is 32 bits, so that's 1GB of memory for the Page Table alone!



Virtual Memory

Page table size

- We need to translate every possible address:
  - Our programs have 32-bit Virtual Address spaces
  - That's 2^{30} words that need Page Table Entries (1 billion entries!)

  (If they don't have a Page Table Entry then we can't access them because we can't fi
  the physical address.)
- How can we make this more manageable?
  - What if we divided memory up into chunks (**pages**) instead of words?

Fine-grain:
Maps each word address
2^{30} words to map

Page Table
translates VA→PA

Map

VA	PA
512	12
786	3
1024	2

4kB Pages:
Each entry now covers 4kB of data.

Coarse-grain:
Maps chunks of address
Fewer mappings

Page Table
translates VA→PA

Map

VA	PA
0-4095	4096-8191

Virtual Memory

Coarse Grain: Pages instead of words

- The **Page Table** manages larger chunks (**pages**) of data:
  - Fewer Page Table Entries needed to cover the whole address space
  - But, less flexibility in how to use the RAM (have to move a page at a time
- Today:
  - Typically 4kB pages (1024 words per page)
  - Sometimes 2MB pages (524,288 words per page)

Q: How many entries do we need in our Page Table with 4kB pages on a 32-bit machine?

- 1 for every word = 2^{30} = 1 billion
- 1 for every 1024 words = 1 million
- 1 for every 4096 words = 262,144

A: 1 for every 1024 words = 1 million
With 4kB pages we have 1024 words per page. That means we need 1 Page Table Entry for every 1024 words. In total we have 1 billion words, so 1 billion ÷ 1024 is 1 million Page Table Entries. This is *much* more manageable.

Coarse-grain: maps chunks of address

Page Table
translates VA→P

Map	
VA	**PA**
0-4095	4096-

Virtual Memory

How do we map virtual address

Page Table
translates VA→PA

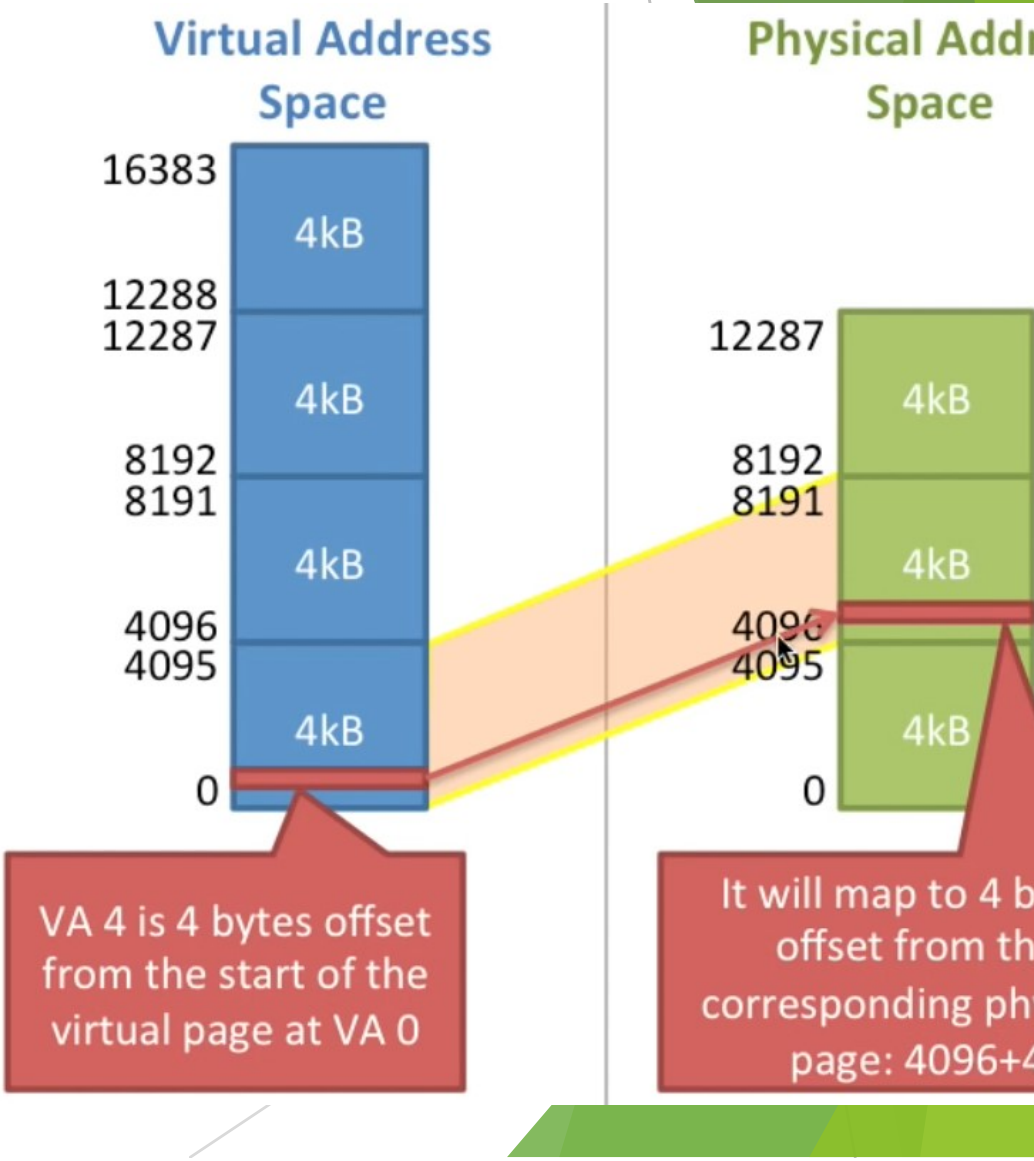
Map	
VA	**PA**
0-4095	4096-8191

Coarse-grain:
maps chunks
of address

Q: What is the Physical Address for Virtual Address 4?

- 4
- 4096
- 4100
- Unknown

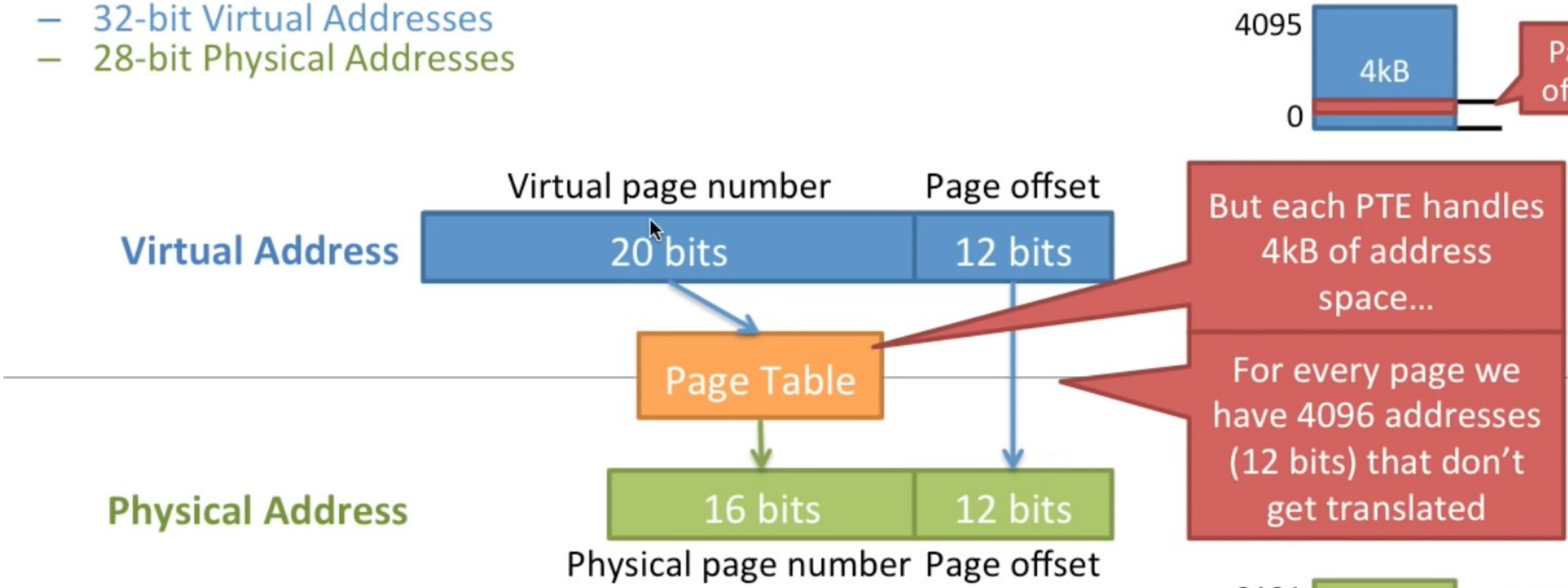
A: 4100
All VA in the range 0-1023 are mapped to the PA in the range 4096-5119. This means that address 4 will be mapped to 4096+4, or 4100.



Virtual Memory
Address Translation

What happens on a 32-bit machine with 256MB of RAM and 4kB pages?

- 32-bit Virtual Addresses
- 28-bit Physical Addresses



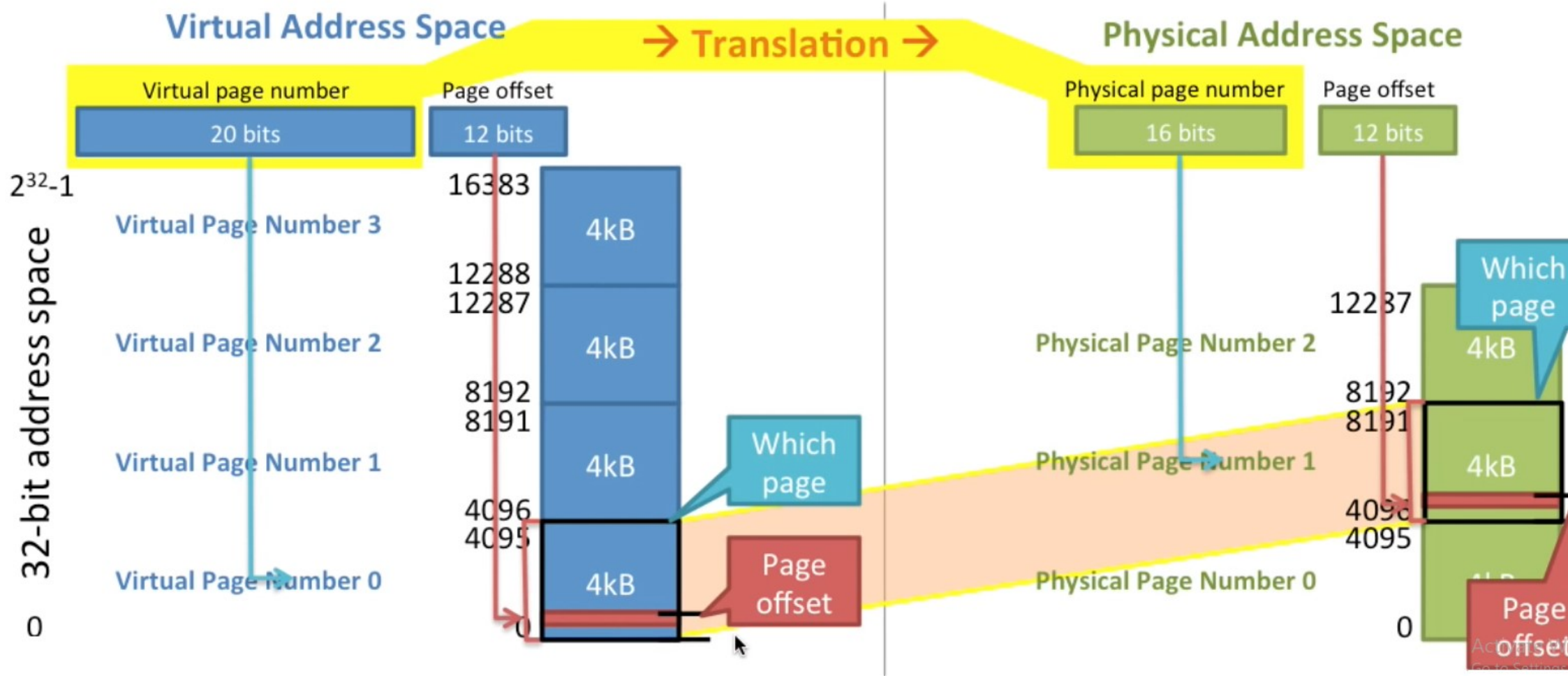
But each PTE handles 4kB of address space…

For every page we have 4096 addresses (12 bits) that don't get translated

Virtual Memory

Pages, offsets and translation

- What happens on a 32-bit machine with 256MB of RAM and 4kB pages?



Virtual Memory

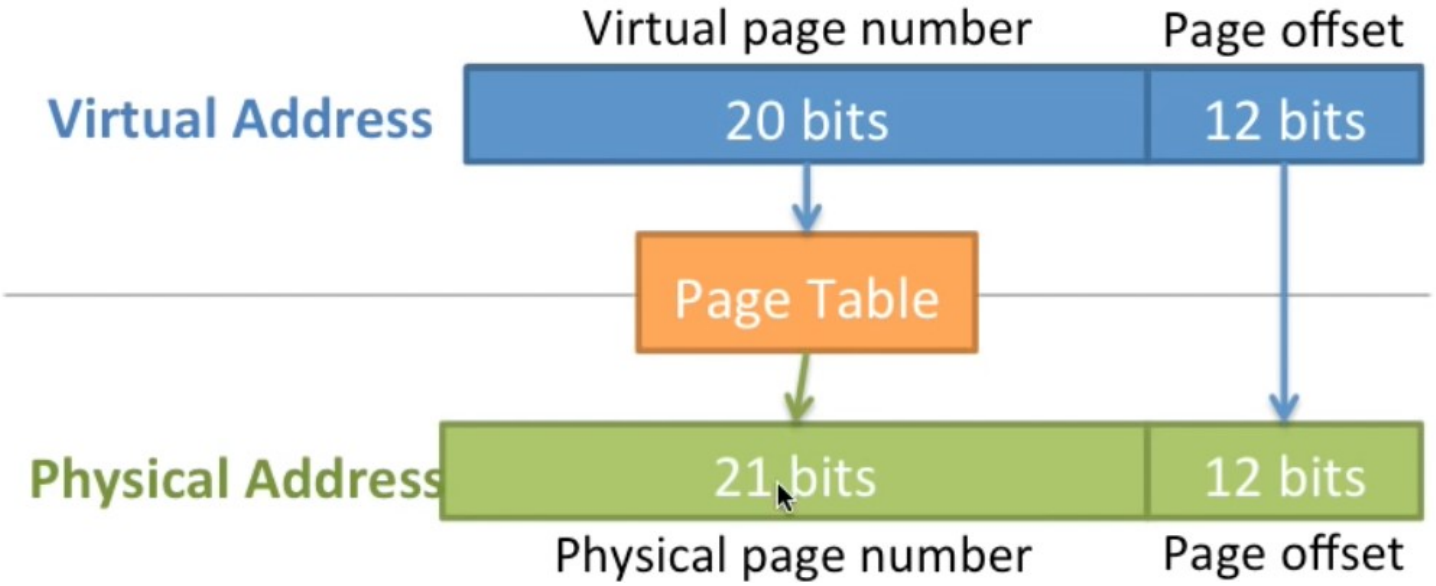
Number of VA and PA bits

Q: What would be true on a 32-bit machine with 8GB of RAM installed?

- The page offset would be larger
- The physical address would be the same size as the virtual address
- You would not need virtual memory
- The physical address would be larger than the virtual address

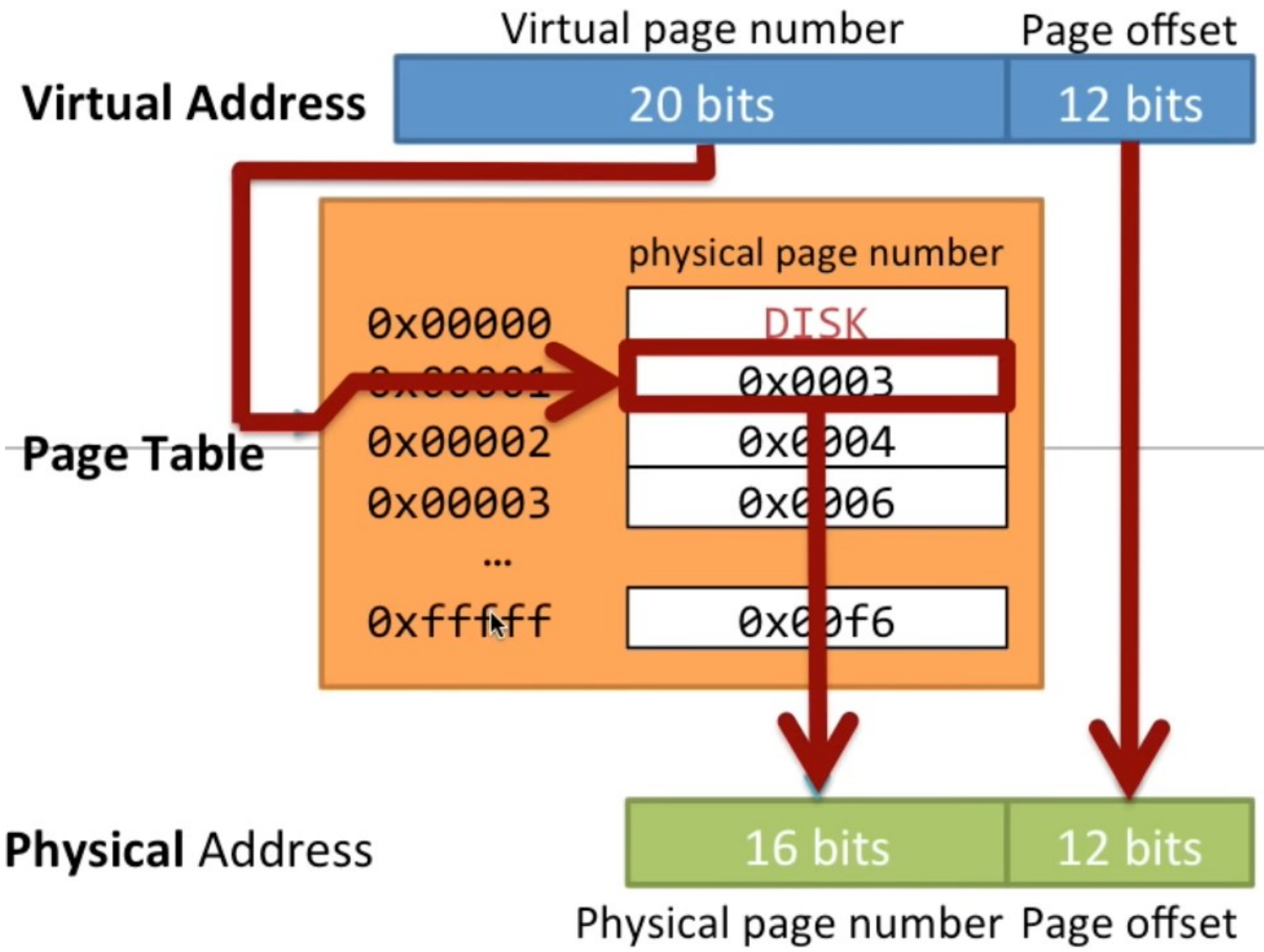
A: The physical address would be larger than the virtual address

To address 8GB of memory you need 33 bits of address. This means the physical address will need to be 33 bits. Of course each program can only address 32 bits because the program address space is only 32 bits. (This is why we have moved to 64 b processors.)



Virtual Memory

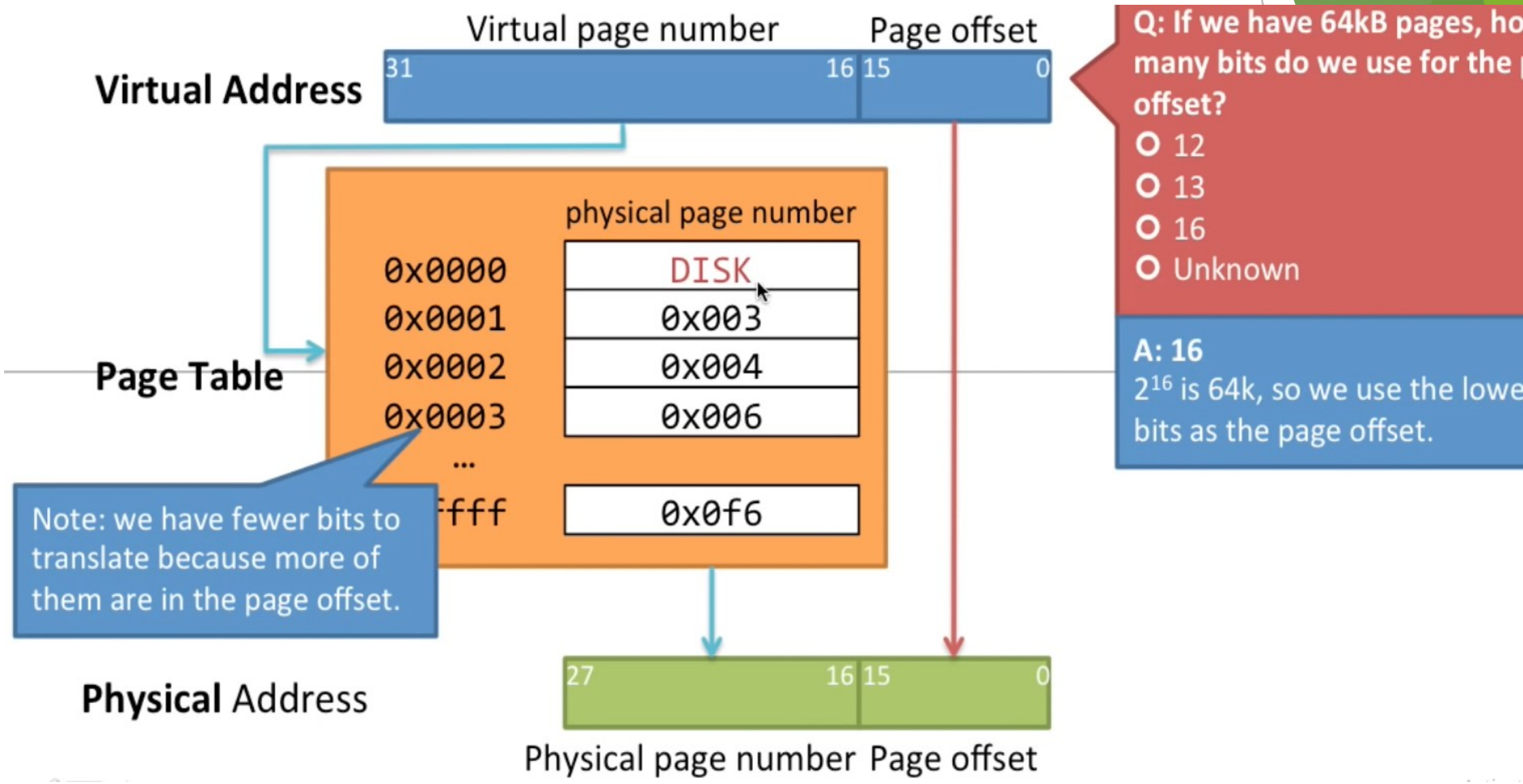
Page table lookup method



- Virtual page number translated to the physical page number
- Page offset untranslated
- Page Table keeps mappin for every virtual page → physical page

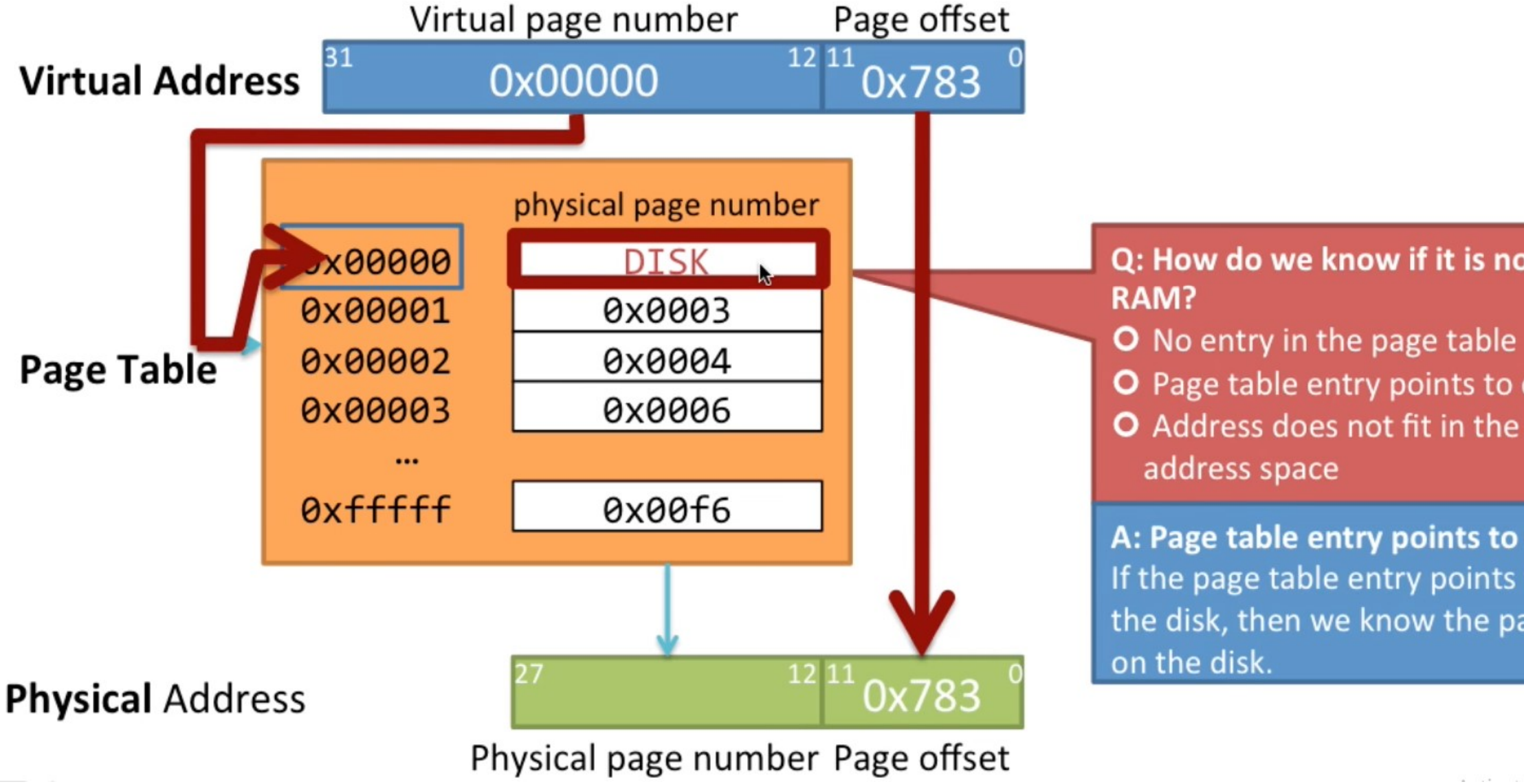
Virtual Memory

Example address translation for 64 KB pages



Virtual Memory

What happens if a page is not in RAM (1)



Virtual Memory

What happens if a page is not in RAM (2)

- Page Table Entry says the page is on disk
- Hardware (CPU) generates a **page fault exception**
- The hardware jumps to the OS page fault handler to clean up
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first
  - The OS then reads the page from disk and puts it in RAM
  - The OS then changes the Page Table to map the new page
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.)

"Dirty" means the da
been changed (writ
If the page has not
written since it was l
from disk, then it do
have to be written

Q: How long does this take?

- No time
- A short time
- A long time
- An amazingly, incredibly, painfully long time

A: An amazingly, incredibly, painfully long time

Disks are *much* slower than RAM, so every time you have a page fault it takes an amazingly, incredibly, painfully long time.

Virtual Memory

How long does a page fault take?

- Page Table Entry says the page is on disk — ~1 cycles
- Hardware (CPU) generates a **page fault exception** — ~100 cycles
- The hardware jumps to the OS page fault handler to clean up — ~10,000 cycles
  - The OS chooses a page to evict from RAM and write to disk
  - If the page is **dirty**, it needs to be written back to disk first — ~40,000,000 cycles
  - The OS then reads the page from disk and puts it in RAM — ~40,000,000 cycles
  - The OS then changes the Page Table to map the new page — ~1,000 cycles
- The OS jumps back to the instruction that caused the page fault.
  - (This time it won't cause a page fault since the page has been loaded.) — ~10,000 cycles

In the time it takes to do handle one page fault
you could execute 80 million cycles on a modern CPU.

Page faults are the SLOWEST possible thing that can happen to a computer (except for human interaction).

Virtual Memory

Virtual Memory and Paging

- Paging is one of those things you don't ever want to use, but it's good to have around:
  - **Very, very slow** when you have to page
  - **Very, very good** if you don't crash because you run out of memory

- If you have enough* RAM then you will never page**

- This is why buying more memory may make your computer fa

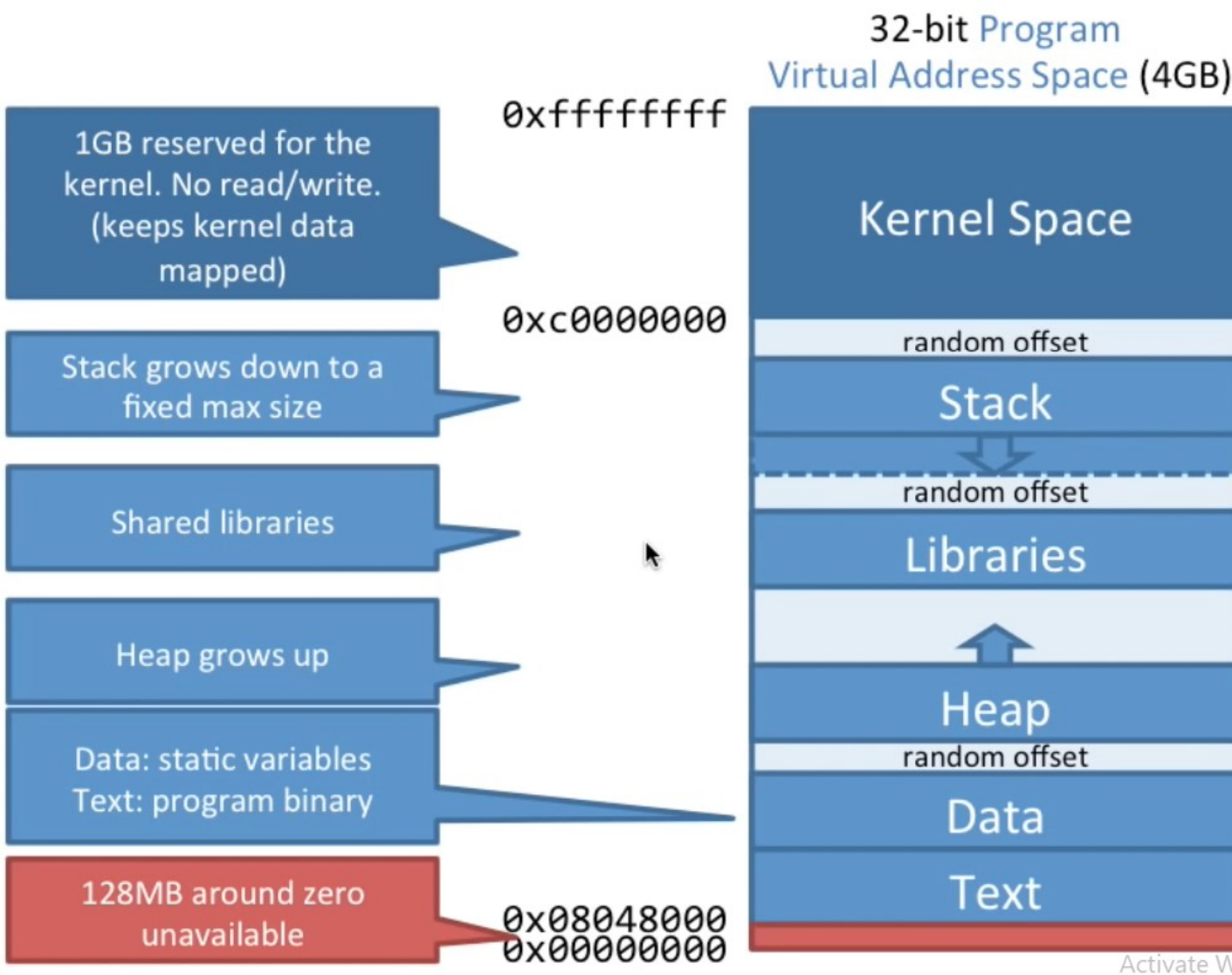
*You can never have enough RAM.
**Some systems do not page:

- iOS: the OS kills your program if you use too much memory
- OS X 10.9: the OS compresses your program first, then pages if it has to

Virtual Memory

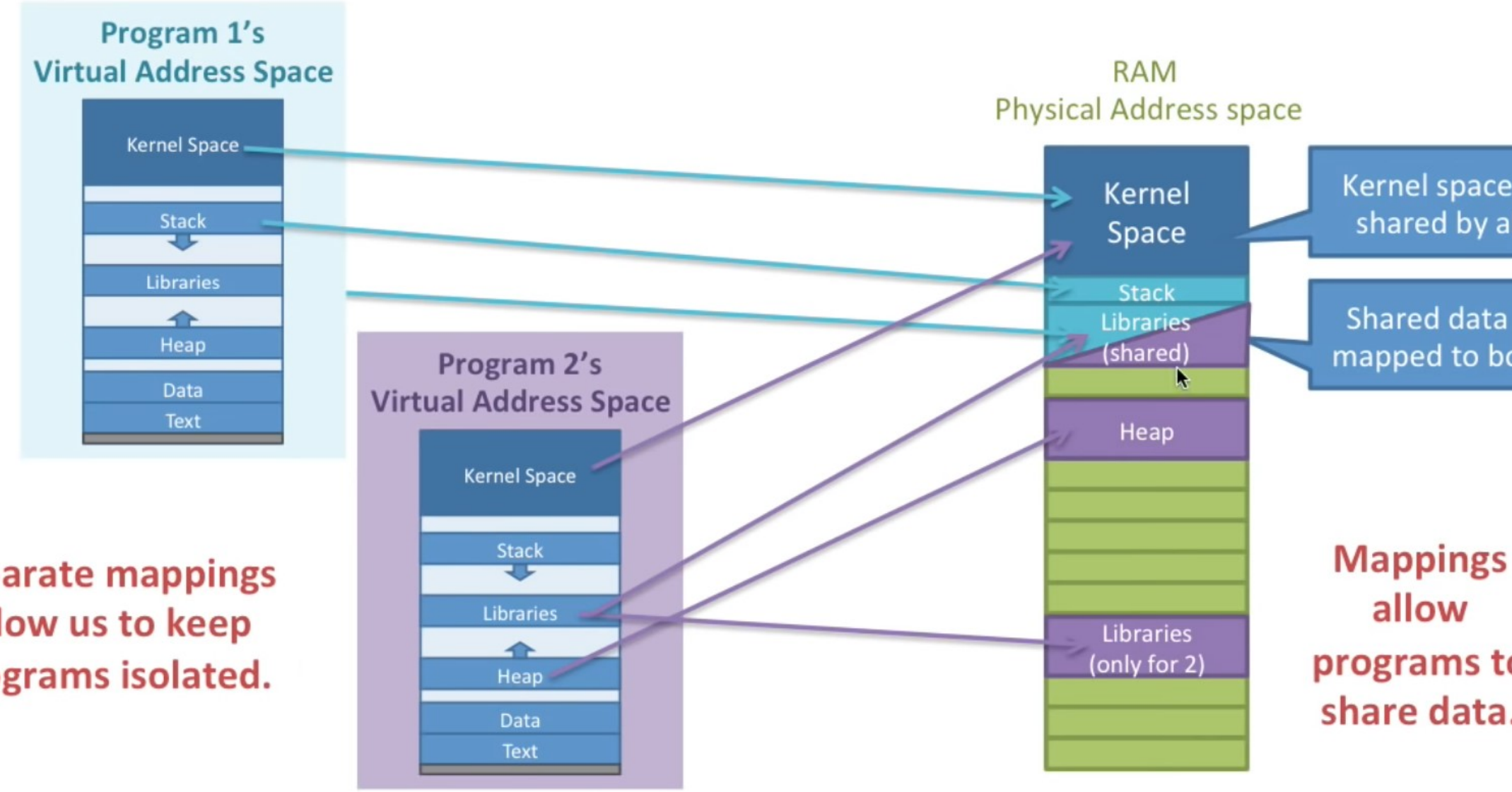
Program address space in Linux (1)

- Each program has it's own 32-bit virtual address space
- Linux defines how that address space is used:
  - 1GB reserved for kernel
  - Program static data at the bottom
  - Heap grows up
  - Stack grows down (and has a fixed maximum size)
- Random offsets to enhance security
  - (You never know exactly where a certain piece of data/code will be.)



Virtual Memory

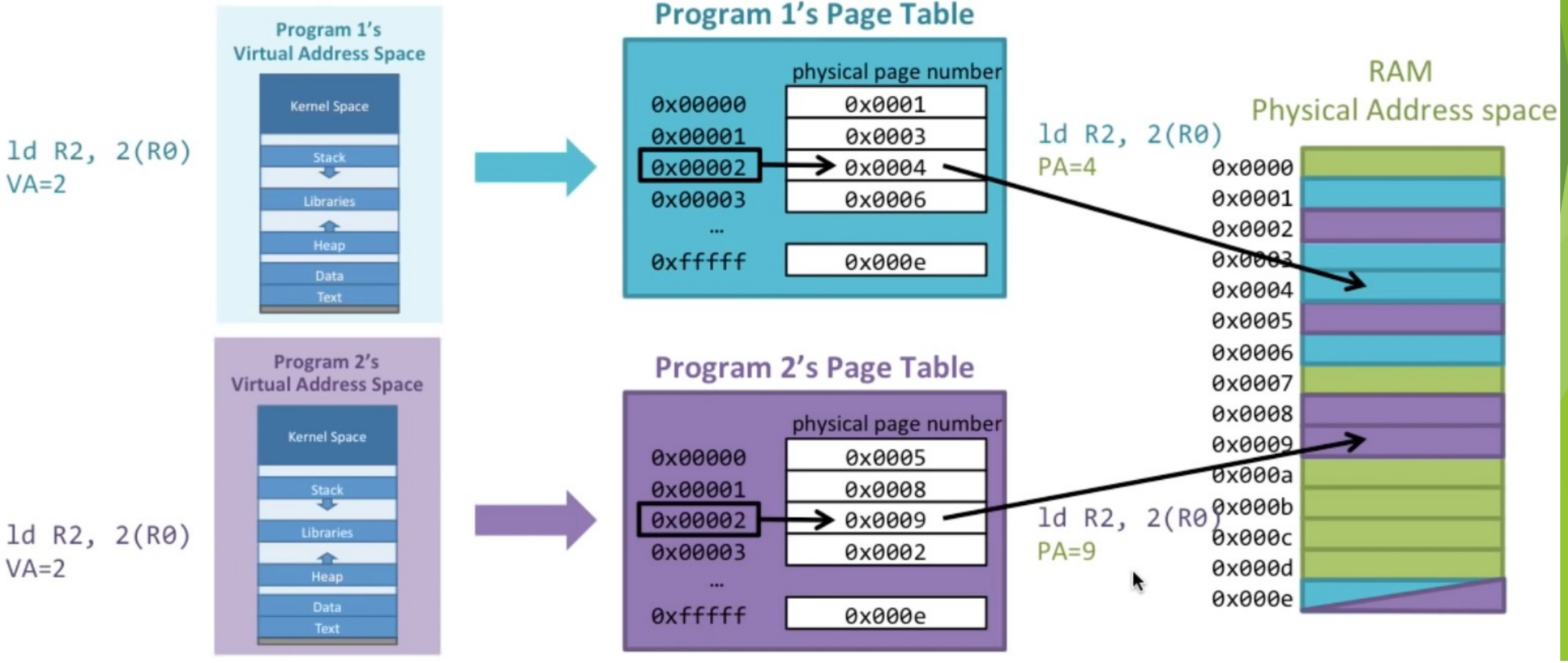
Program address space in Linux (2)



Virtual Memory

How separate mapping is provided

- Each process needs its own Page Table
- OS makes sure they only map to the same physical address when we want sharing



Virtual Memory

How much does VM cost for every memory access

Q: What do we have to do for each memory access with virtual memory?

A:

1. Access the page table in RAM
2. Translate the address
3. Access the data in RAM

This is a lot of work for every memory access…and remember we have an average of 1.33 accesses for each instruction!

Virtual Memory
VM in practice

- VM is great:
  - Unlimited programs/memory, protection, flexibility, etc.

- But it comes at a high cost:
  - ***Every*** memory operation has to look up in the page table
  - Need to access **1) the page table** and **2) the memory address** (2x memory acce

  (Remember, 1.33 memory accesses per instruction. This is going to hurt.)

- How can we make a page table look up ***really really*** fast?
  - Software would be far too slow
    (e.g., an extra 5 instructions for every memory access would kill performance)

- Perhaps a hardware page table **cache**?

Virtual Memory

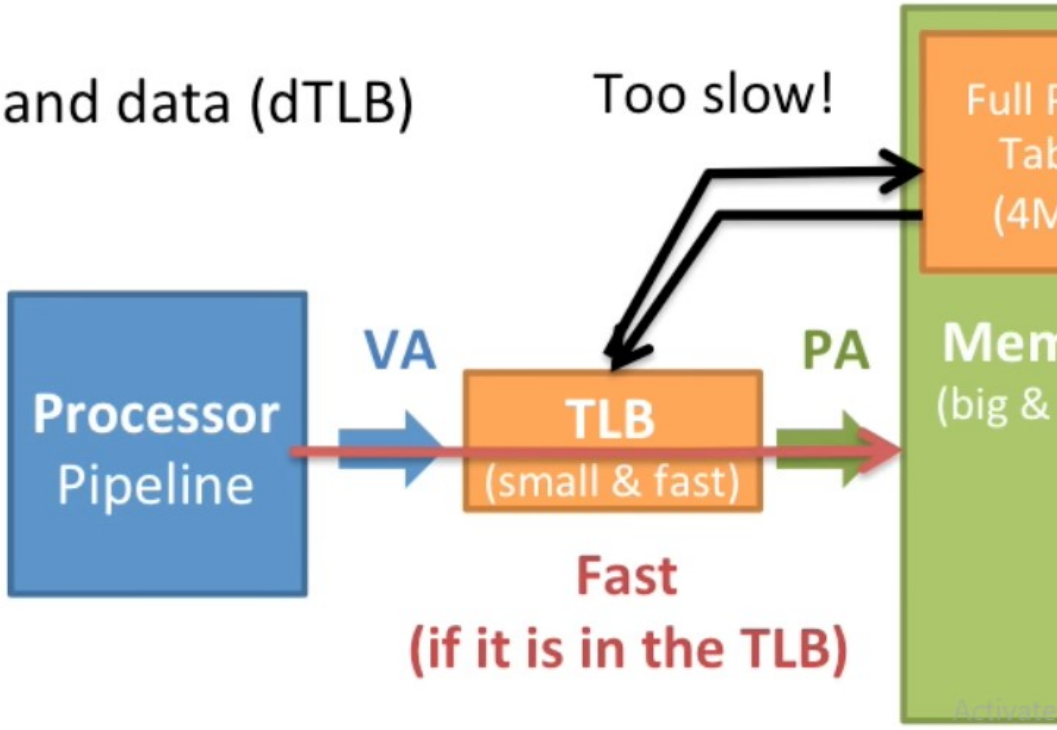
Making VM fast using TLB

- To make VM fast we add a special **Page Table cache:** the **Translation Lookaside Buffer (TLB)**
  - Fast: less than 1 cycle (have to do it for every memory access)
  - Very similar to a cache

- To be fast, TLBs must be small:
  - Separate TLBs for instructions (iTLB) and data (dTLB)
  - 64 entries, 4-way (4kB pages)
  - 32 entries, 4-way (2MB pages)

  (Page Table is 1M entries)

Lots of locality!
Miss rates are typically only a few percent.



Virtual Memory

What can happen during memory access

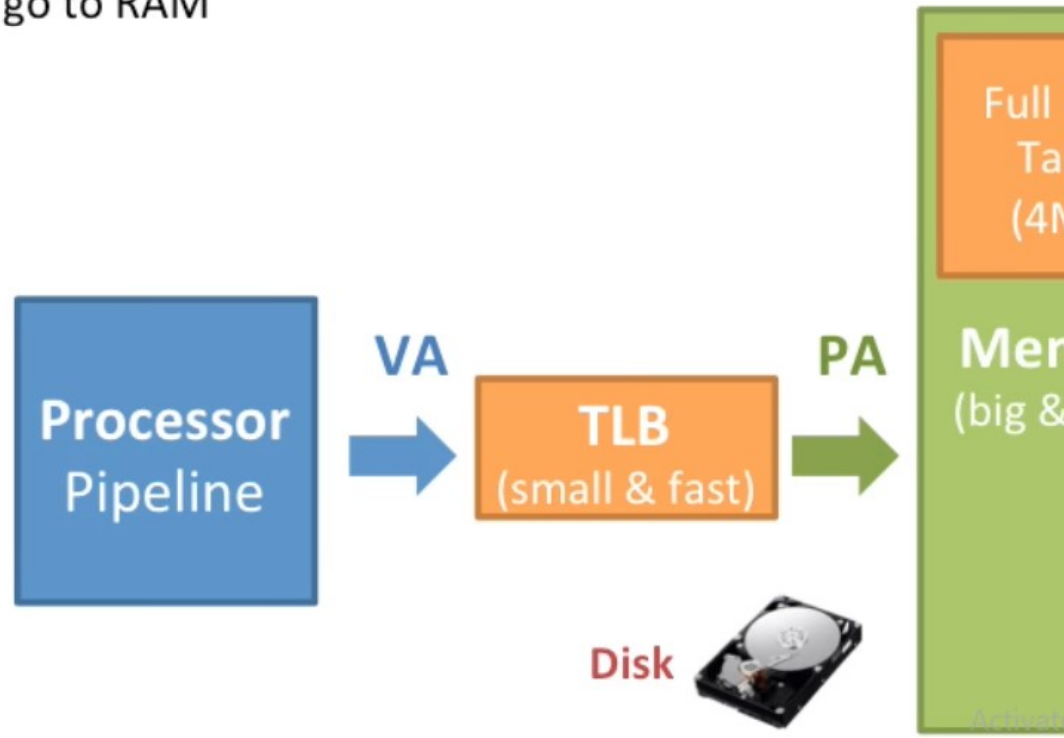
Good: Page in RAM

- **PTE in the TLB**
  - **Excellent**
  - **<1** cycle to translate, then go to RAM (or cache)
- **PTE not in the TLB**
  - **Poor**
  - **20-1000** cycles to load PTE from RAM, then go to RAM

With 1.33 memory accesses per instruction we can't afford 20-1000 cycles very often.

Bad: Page not in RAM

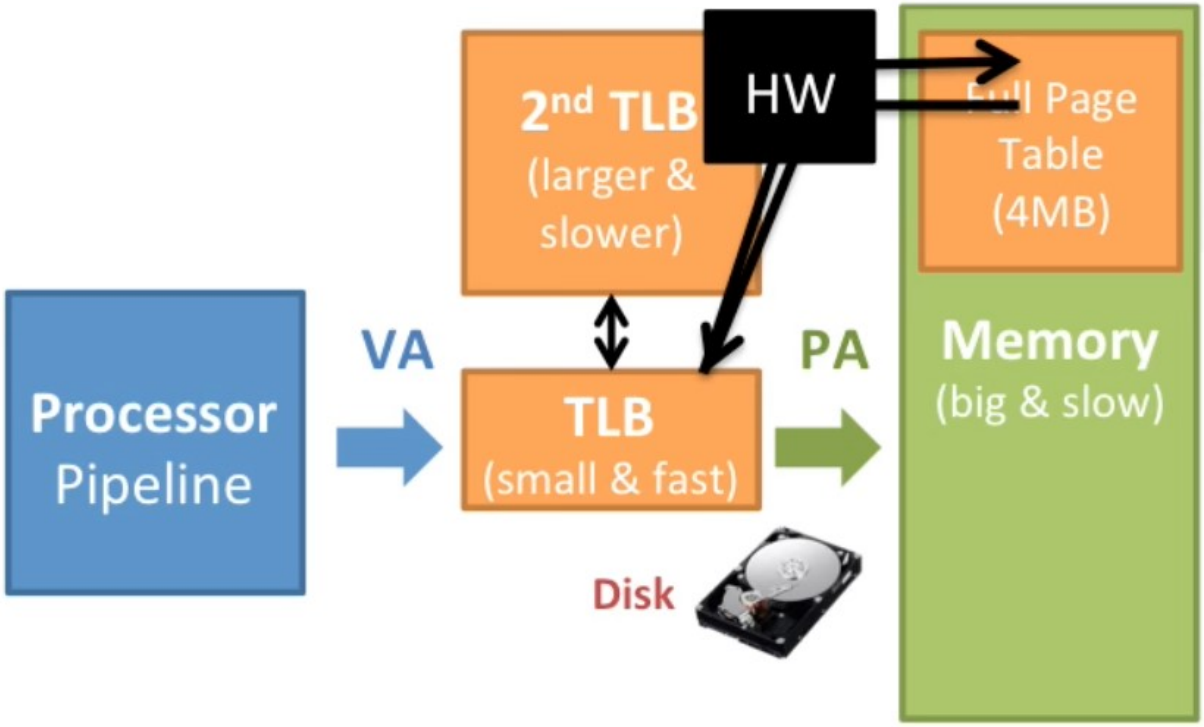
- **PTE in the TLB** (unlikely)
  - **Horrible**
  - 1 cycle to know it's on disk
  - **~80M** cycles to get it from disk
- **PTE not in the TLB**
  - **(ever so slightly more) horrible**
  - 20-1000 cycles to know it's on disk
  - **~80M** cycles to get it from disk



Virtual Memory
Making TLBs bigger

Q: TLBs are small. How can we make them effectively bigger without slowing them down?

- Just make them hold more PTEs!
- Make pages larger
- Add a second TLB that is larger, but a bit slower
- Have hardware to fill the TLB automatically if there is a miss. (E.g., instead of having the OS do it in software.)



A:

1. Make pages larger

This increases the reach of the TLB because you need few pages to cover more data.

2. Add a second TLB that is larger, but a bit slower

Sure. Most processors have a level 2 TLB that is about 8x larger than the level 1 TLB, but also about twice as slow.

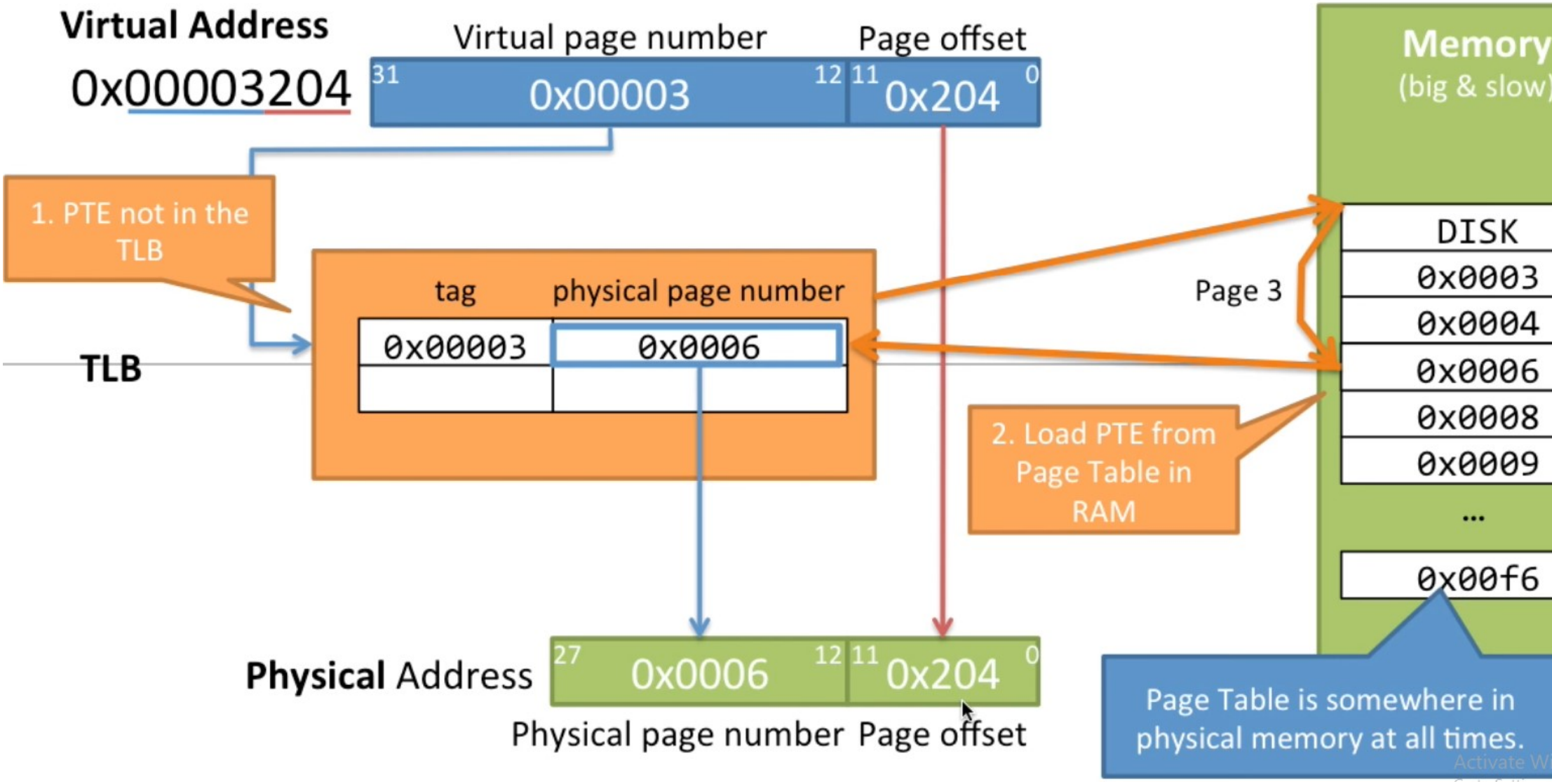
3. Have hardware to fill the TLB automatically

This is called a "hardware page table walk". Basically the hardware assumes the page table is in a special form in memory, and it can go get data from it on a TLB miss without having to go to the OS.

64 4kB pages = 256kB of data
32 2MB pages = 64MB of data

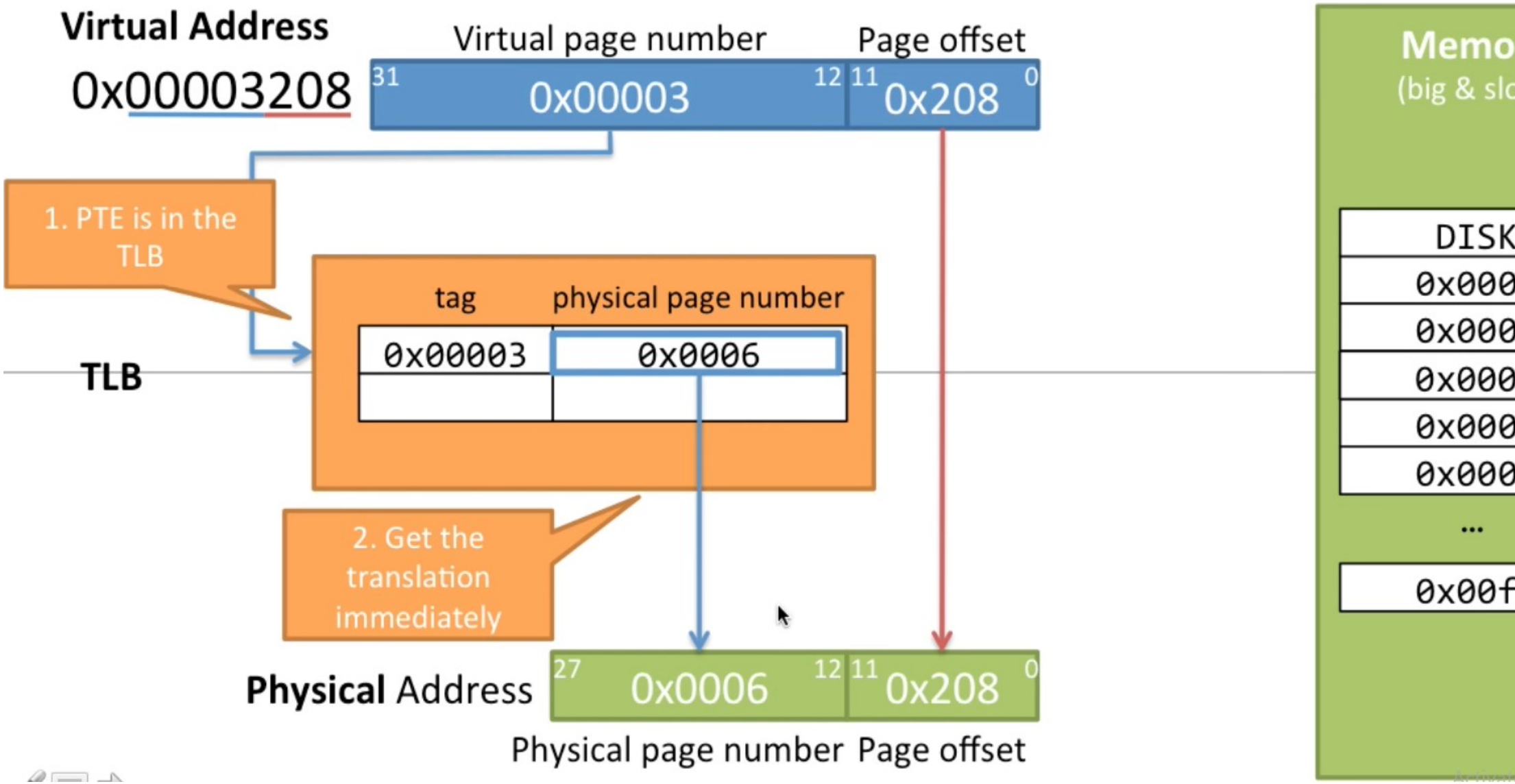
Virtual Memory

Example of address translation: TLB is empty



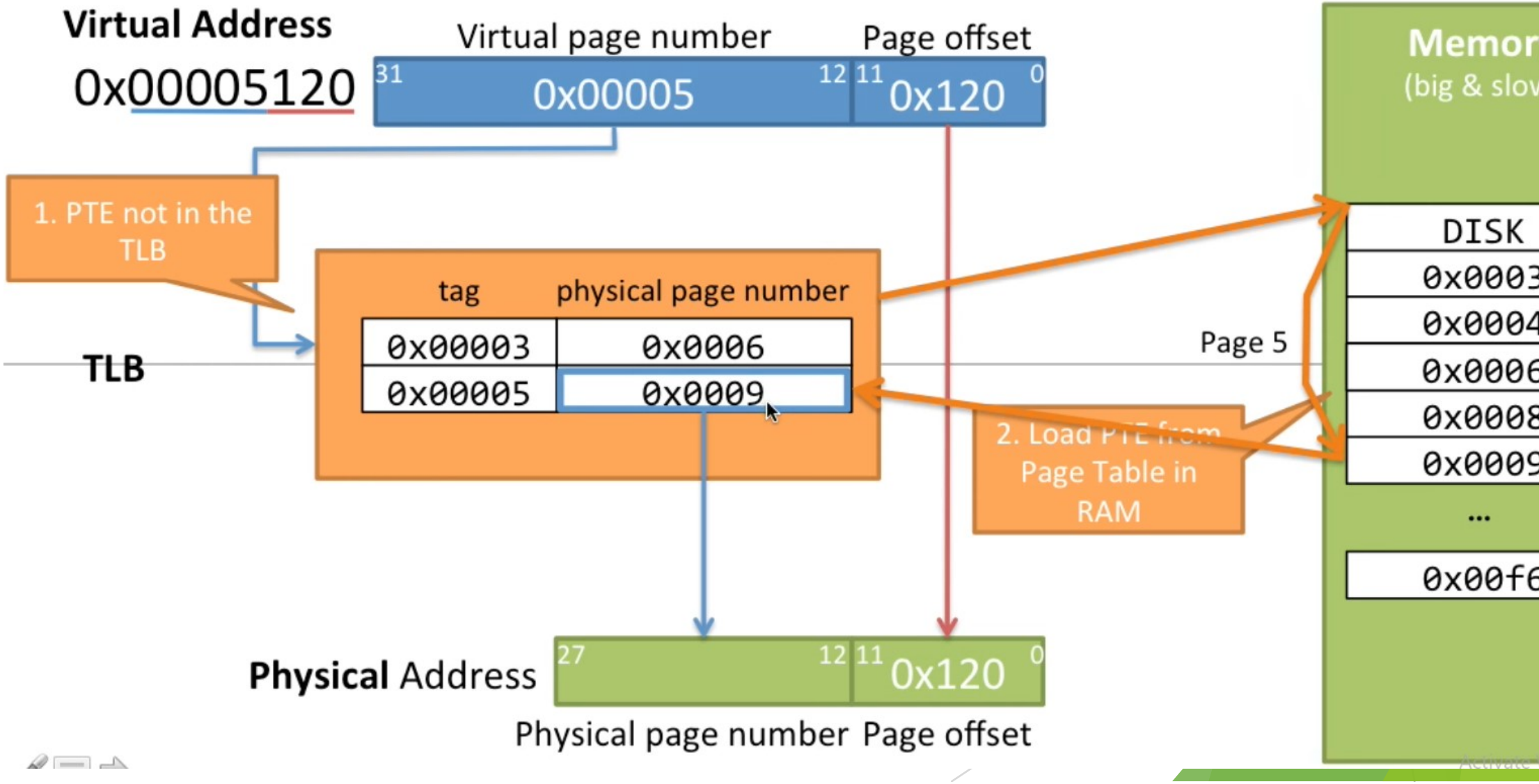
Virtual Memory

Example of address translation: TLB hit



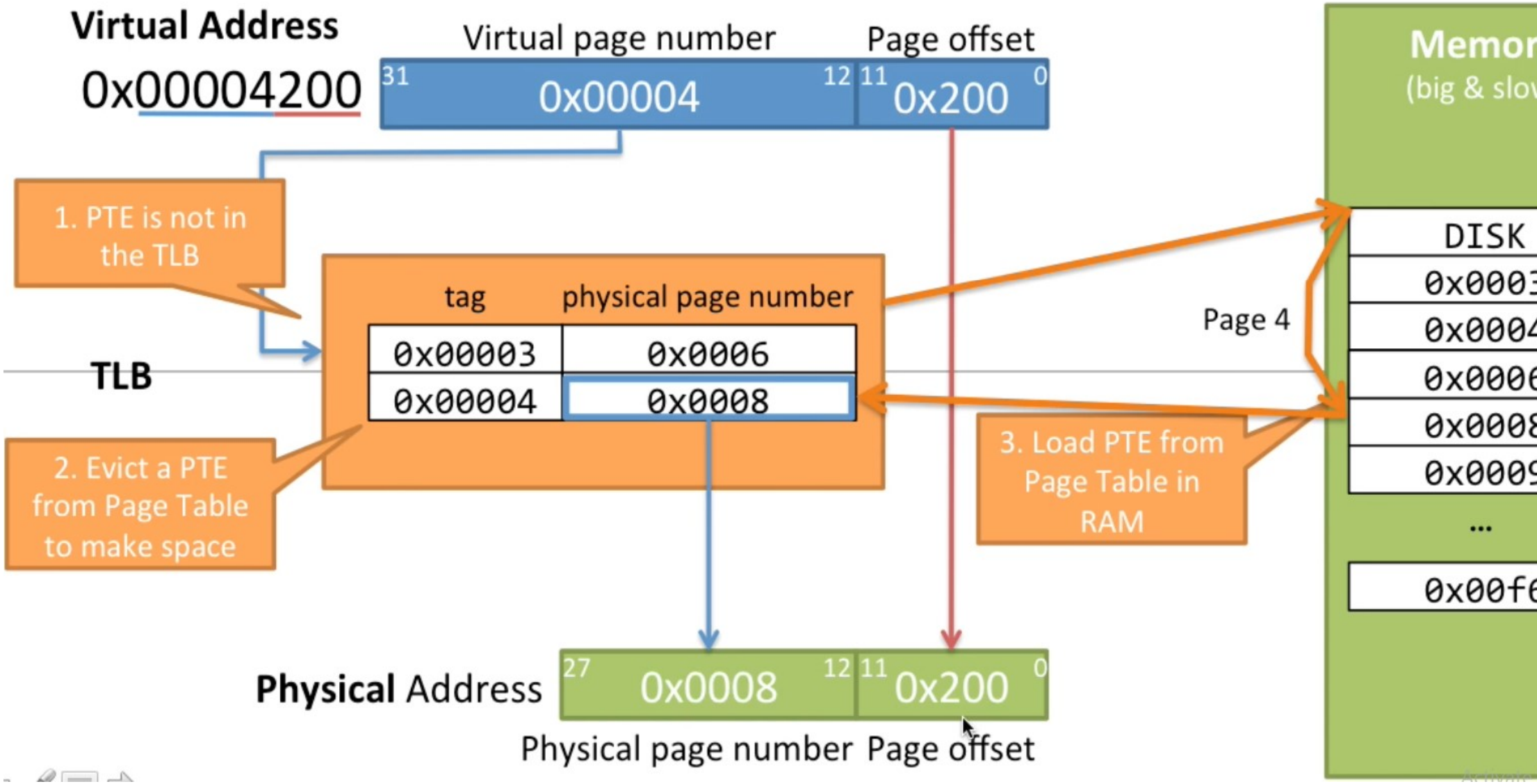
Virtual Memory

Example of address translation: TLB miss



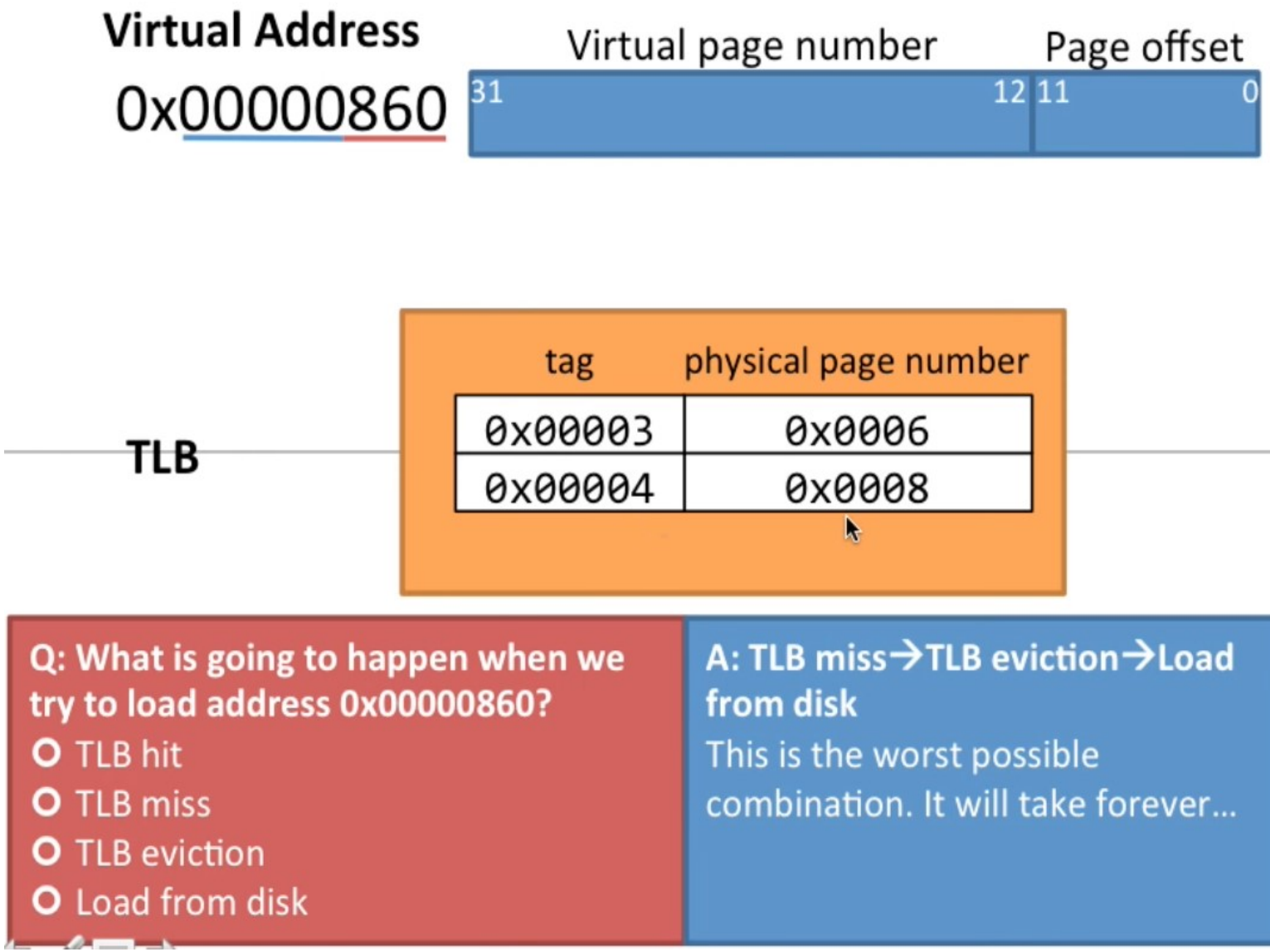
Virtual Memory

Example of address translation: TLB miss + evictio



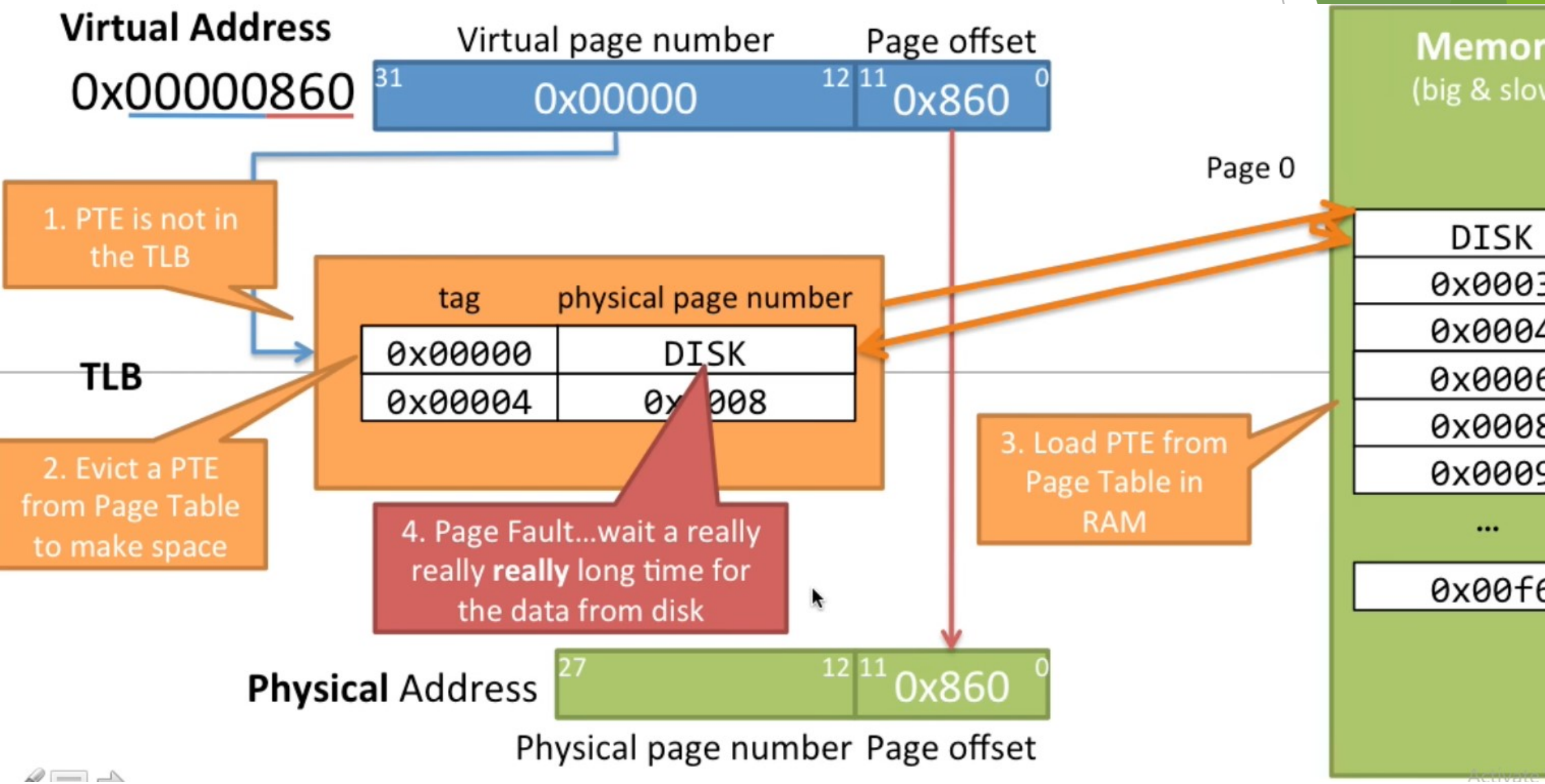
Virtual Memory

Example of address translation: disk access (1)



Virtual Memory

Example of address translation: disk access (2)



Virtual Memory

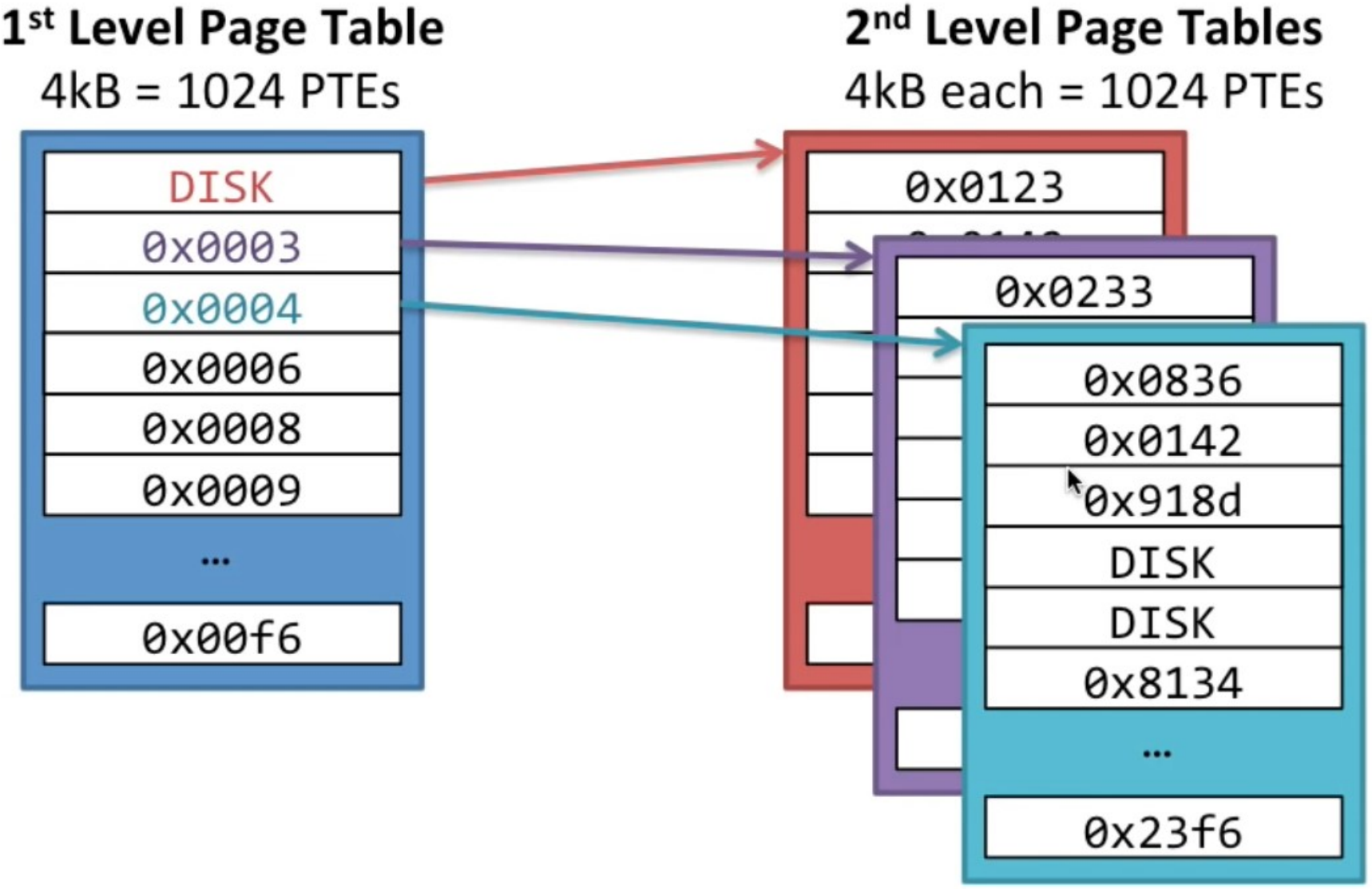
Page table size

- For 32-bit machine with 4kB pages we need:
  - 1M Page Table Entries (32 bits – 12 bits for page offset = 20 bits, 2^{20}
  - Each PTE is about 4 bytes (20 bits for physical page + permission bits
  - 4MB total
- Not bad…

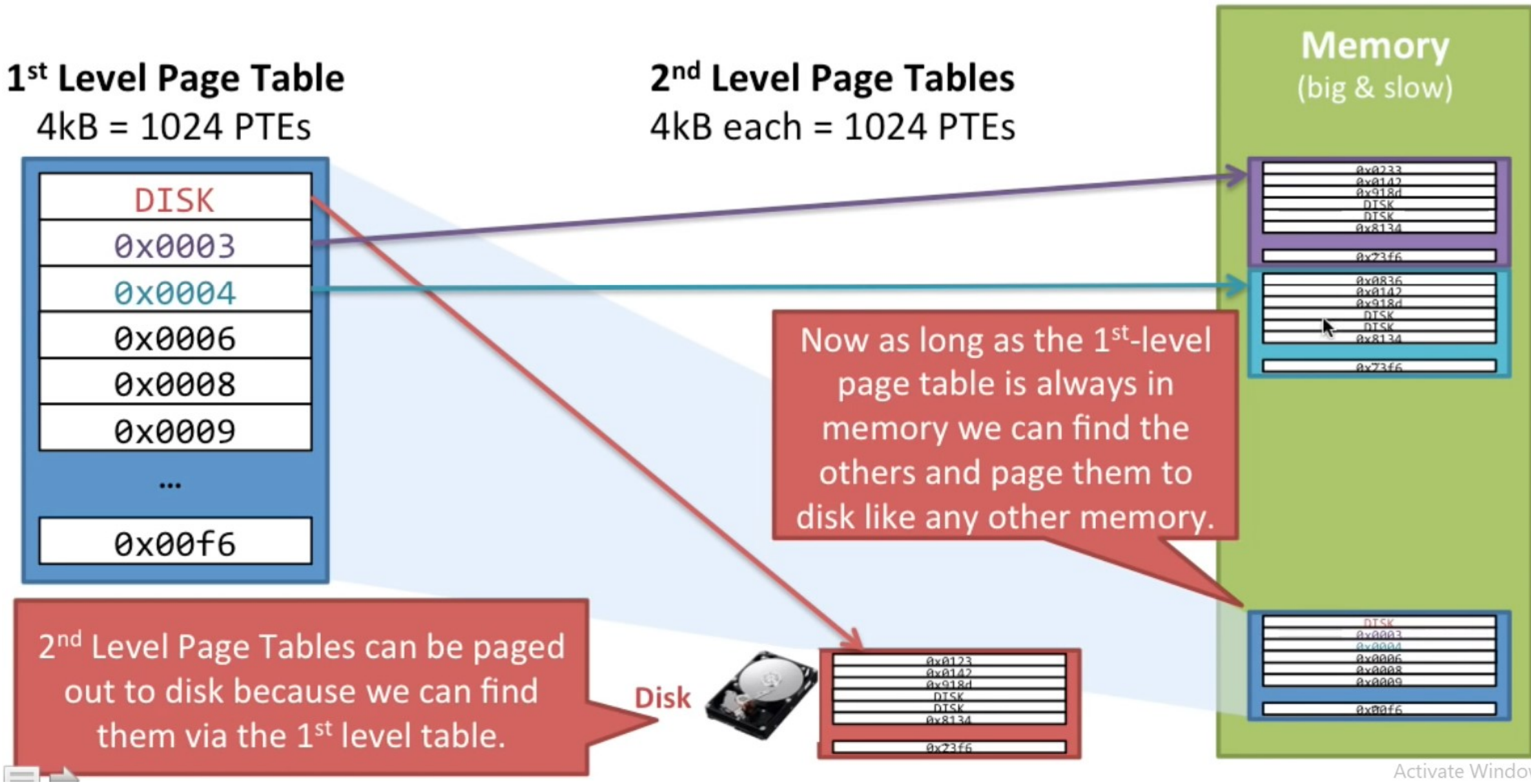
  …except **each program needs its own page table**…
  - If we have 100 programs running, we need 400MB of Page Tables!
- And here's the tough part:
  - We **can't swap the page tables out to disk**
  - If the page table is not in RAM, we have no way to access it to find it
- How can we fix this?
  - Just add more indirection…

Virtual Memory

Multi-level page tables (1)



Virtual Memory
Multi-level page tables (2)



Virtual Memory

Multi-level page table size

Q: With multi-level page tables, what is the smallest amount of page table data we need to keep in memory for each 32-bit application?

- 4MB+4kB (always need the whole page table plus the 1st-level page)
- 4MB (always need the whole page table in RAM)
- 4kB+4kB (need the 1st-level page table and one 2nd-level page table)
- 4kB (just need the 1st-level page table)

A: 4kB+4kB

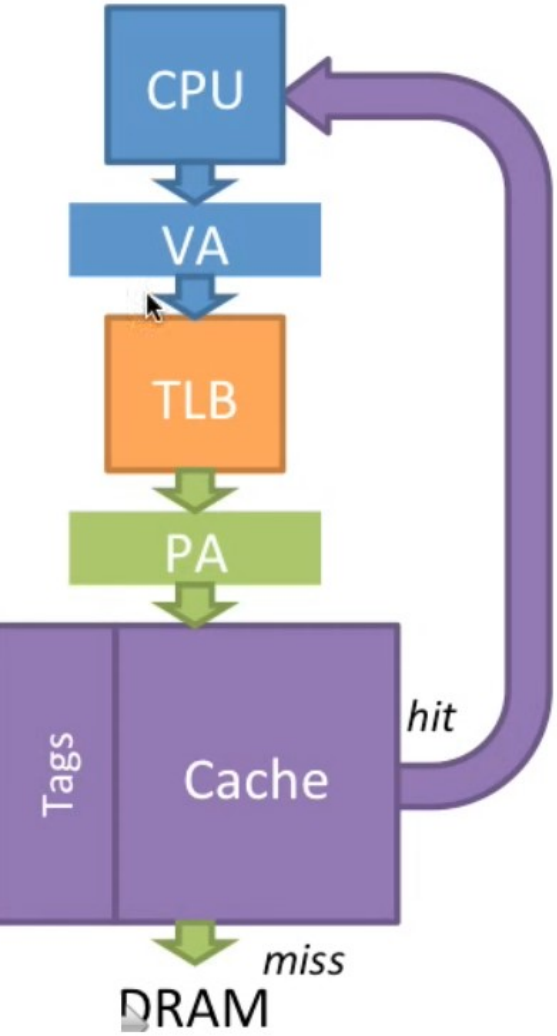
We always need the 1st-level page table so we find the 2nd-level ones.

But, the 1st-level page table only helps us find page tables. It isn't enough by itself to transla program addresses. So we need at least one 2 level page table to actually translate memory addresses:

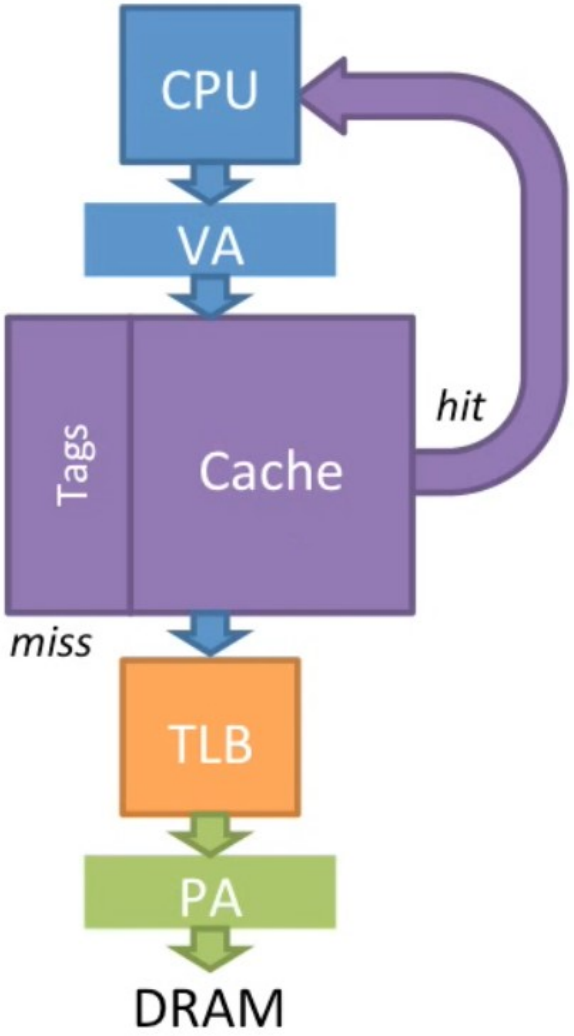
One first-level page table: 4kB
One second-level page table: 4kB
4kB+4kB per application (much better than 4M per application!)

Virtual Memory
TLBs and Caches

Physical Cache



Virtual Cache



Physical Cache

Slow: Must do a TLB lookup *before* accessing the cache

Virtual Cache

Fast: TLB lookups *only* when we miss in the cache

Q: Can you have two programs share a virtual cache?

- Yes. The TLB provides protection.
- No. Each needs its own TLB.
- No. The cache is virtual so there is no way to provide protection

A: No.

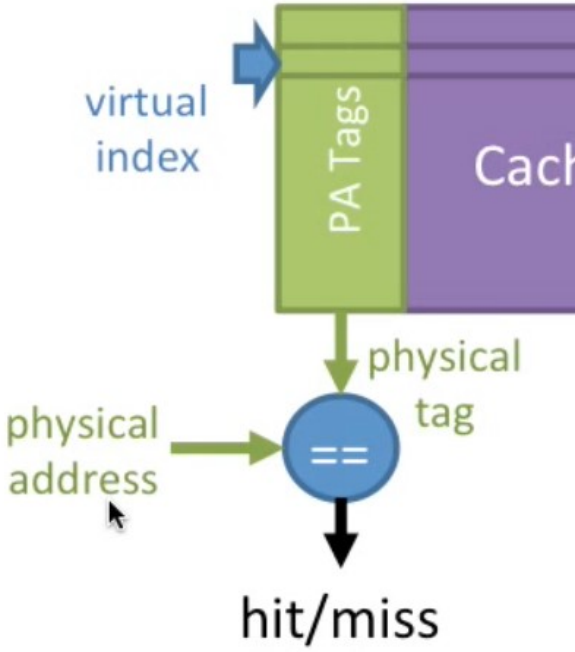
A virtual cache stores data by the virtual program address. There is no translation, so VM-based protection can't keep applications apart!

(You can have a separate bit for the process ID or you have to flush the caches when you switch programs.)

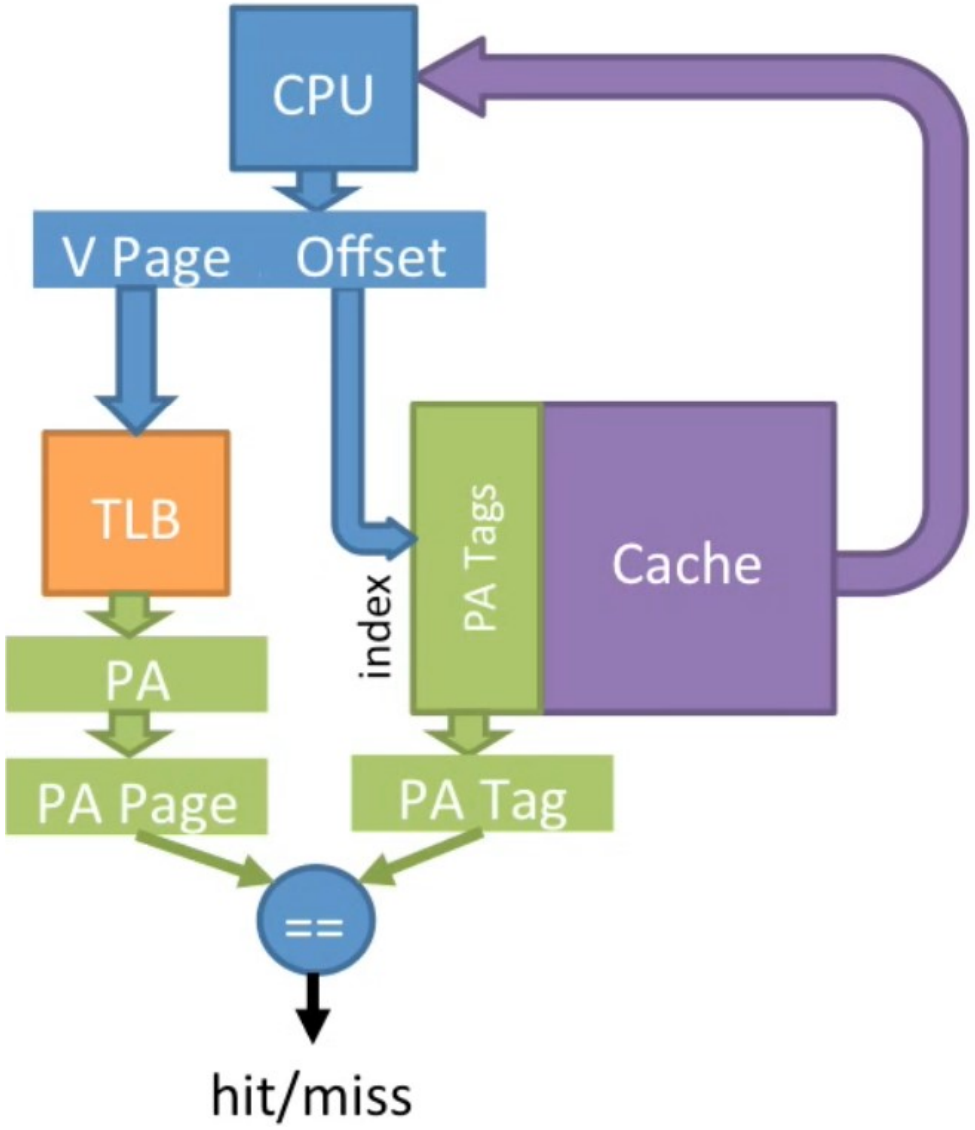
Virtual Memory
VIPT Caches (1)

- Can we access the cache at the same time as the TLB but still have the protection of VM?

- Idea:
  - Look up in the cache with a **virtual address**
  - Verify that the data is right with a **physical tag**
- VIPT: Virtually Indexed, Physically Tagged
  - Data in the cache is indexed by the virtual address
  - But tagged by the physical address
  - We only get a hit if the tag matches the physical address, but we can start looking with the virtual address



Virtual Memory
VIPT Caches (2)



TLB translation and **Cache lookup** at the **same time**

- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as th cache

Safe: Cache hit only on PA match

- **But**, can only use non-translated bits to index cache (limit on how large the cache can be)
- Most processors use VIPT for L1 caches today

Virtual Memory
VIPT cache size

Q: With 4kB pages, how many bytes can a direct-mapped (1-way) VIPT cache store?

- 4kB
- 4096kB
- Unlimited
- Need to know the cache line size

A: 4kB

We can only use the page offset bits (12 for 4kB pages) to index into the cache. S index can only address 12 bits of address 4kB of data.

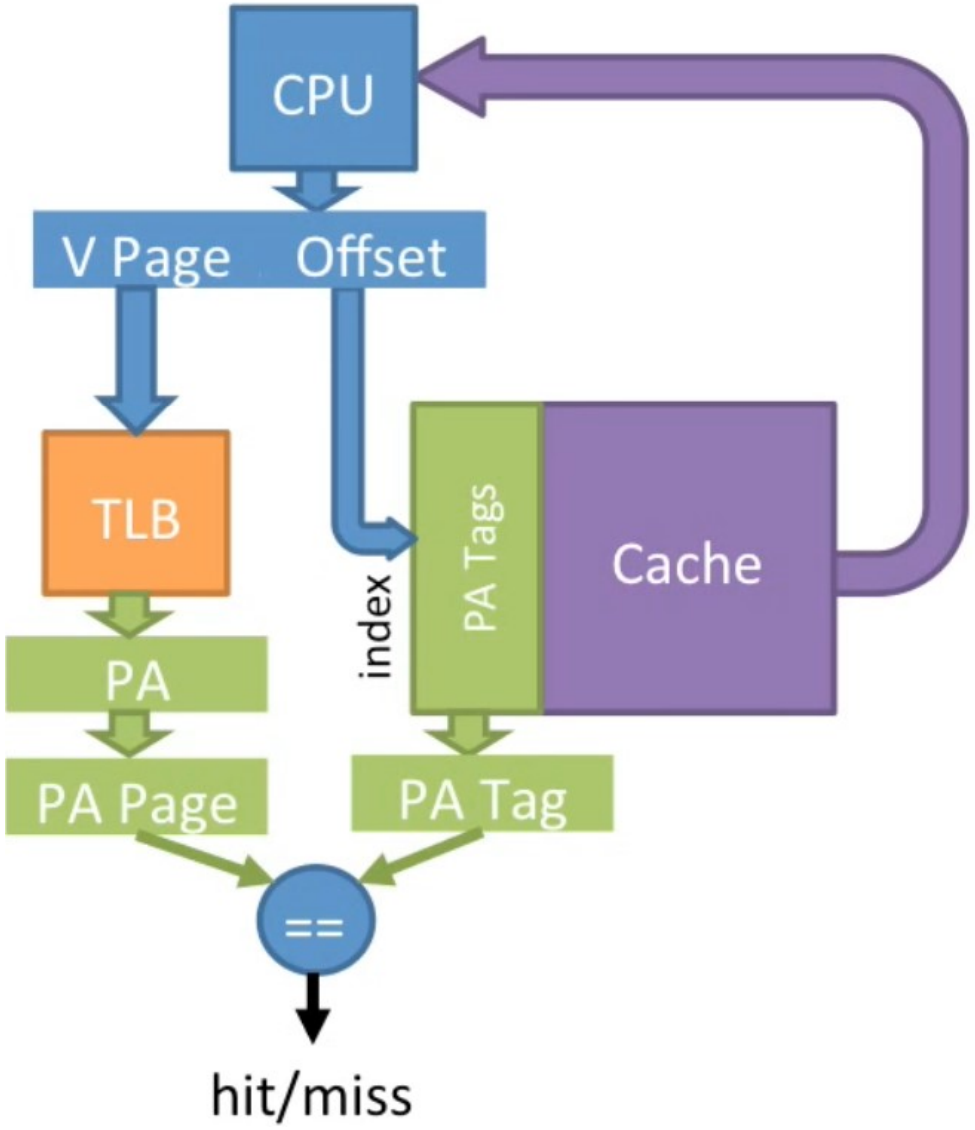
Note: if we increase associativity, we car make it larger! E.g., 8-way cache would g us 32kB of cache size. (Each way uses the same index.)

Virtual Memory

Virtual Memory Summary

- VM adds a level of **indirection** between the **virtual program addresses (VA)** and the **physical RAM addresses (PA)**
- This allow us to do lots of cool things:
  - Map memory to disk ("unlimited" memory)
  - Keep programs from accessing each other's memory (security)
  - Fill holes in the RAM address space (efficiency)
- But, **we have to translate every single memory access** from a **VA** to a **PA**
  - Page Tables for each program keep track of all translations
  - Use larger pages (4kB) to reduce the number of Page Table Entries (PTEs) ne
  - Fast translation via a hardware translation lookaside buffer (TLB)
- Need to combine the TLB and the cache for good performance
  - Physical caches: require translation first (slow)
  - Virtual caches: use virtual addresses (no translation: fast, but protection)
  - Virtually-Indexed, Physically-Tagged (VIPT) caches: use VA for index and PA f

Virtual Memory
VIPT Caches (2)



TLB translation and **Cache lookup** at the **same time**

- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as th cache

Safe: Cache hit only on PA match

- **But**, can only use non-translated bits to index cache (limit on how large the cache can be)
- Most processors use VIPT for L1 caches today

Advanced Cache Optimizations

1. Simple/Small L1 Cache

- Small L1 Cache reduces hit time and power consumption
- 1st level cache should match the clock cycle of CPU, otherwise overall execution time will be increased
- Fast clock cycle and Power limitation encourages small L1 cache
- Cache addressing is a three step process
  - Address tag memory with index portion of address
  - Compare the found tag with address tag
  - Choose cache set
- Low associativity is faster, Overlap tag check with data transmission, Consumes less power

Advanced Cache Optimizations

1. Simple/Small L1 Cache (Cont.)

- Higher associativity increases hit time to some extent but
- Small L1 with higher associativity is preferable because
  - many processors take at least two clock cycles to access the cache
    - thus the impact of a longer hit time is not critical
  - to keep the TLB out of the critical path, L1 caches should be virtually indexed.
    - This limits the size of the cache to the page (size * associativity)
  - with the introduction of multithreading, conflict misses can increase, making higher associativity more attractive
  - possibility of eliminating address aliases

Advanced Cache Optimizations

2. Way Prediction

- Idea: predict “the way”
  - – which block within set will be accessed next
- Index multiplexor (mux) starts working early
- Implemented as extra bits kept in cache for each block
- Prediction accuracy (simulated)
  - – more effective of instruction caches
  - – > 90% for two-way associative
  - – > 80% for four-way associative
- On misprediction
  - – Try the other block
  - – Change the prediction bits
  - – Incur penalty (commonly 1 cycle for slow CPUs)
- Examples: 1st use MIPS R10000 in 1990s, ARM Cortex-A8

Advanced Cache Optimizations

3. Pipelined Cache Access

- Give fast clock cycle time and Improves bandwidth
- Effective latency of a 1st level cache hit
  - – Pentium (1993) 1 cycle
  - – Pentium Pro (1995), Pentium III (1999) 2 cycles
  - – Pentium 4 (2000), Intel Core i7 (2010) 4 cycles
- Interaction with branch prediction
  - – Increased penalty for mispredicted branches
- Load instructions are longer
  - – Waiting for cache pipeline to finish
- Pipeline cache cycles make high degrees of associativity easier to implemen

Advanced Cache Optimizations

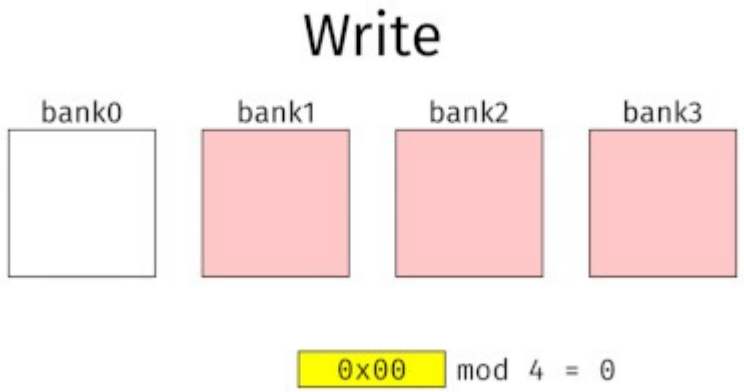
4. Nonblocking Caches to Increase Bandwidth

- For processor with out-of-order execution, if one instruction stalls on a cache miss, should the following instruction stall if its data is in cache?
  - – NO, but you have to design a nonblocking / lockup-free cache
  - – Call it “hit under miss”
- What about “hit under multiple miss” / “miss under miss”?
  - – Next level cache has to be able to handle multiple misses
- Most high-performance processors (e.g., Intel Core i7) support both
- while lower end processors, such as the ARM A8, provide only limited nonblocking support in L2.
- Simulation shows that allowing **hit under 1-miss** reduces the miss penalty by 9% and 12.5% for integer and FP benchmarks, respectively
- Allowing a **hit under 2-misses** improves these results to 10% and 16%

Advanced Cache Optimizations

5. Multibanked Caches

- Main memory has been organized in banks for increased bandwidth
  - Caches can do this too
  - Each cache block is evenly spread across banks called Sequential interleaving
- https://en.wikipedia.org/wiki/Interleaved_memory
- Modern use
  - – ARM Cortex-A8
    - 1-4 banks in L2 cache
  - – Intel Core i7
    - 4 banks in L1 (2 memory accesses/cycle)
    - 8 banks in L2
- Reduced power usage



Advanced Cache Optimizations

7. Merging Write Buffer

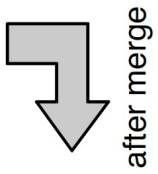
- Write buffer basics
  - - Write buffer sits between cache and memory
  - - Write buffer stores both: data and its address
  - - Write buffer allows for the store instruction to finish immediately
    - Unless the write buffer is full
  - - Especially useful for write-through caches
  - Write-back caches will benefit for when block is replaced
  - Reduces miss penalty

Advanced Cache Optimizations

7. Merging Write Buffer (Cont.)

- Merging write buffer: a buffer that merges write requests
- When storing to a block that is already pending in the write buffer, only update the write buffer
- Another way to look at it: the write buffer with merging is equivalent to a larger buffer but without merging
- Merging buffer reduces stalls due to the buffer being full
- Besides, multiword writes are usually faster than writes performed one word at a time.
- Should not be used for I/O addresses (special memory locations)
- Example: The Intel Core i7, among many others, uses write merging

Addr	valid	Data	valid		valid		valid	
100	1	M[100]	0		0		0	
108	1	M[108]	0		0		0	
116	1	M[116]	0		0		0	
124	1	M[124]	0		0		0	



Addr	valid	Data	valid		valid		valid	
100	1	M[100]	1	M[108]	1	M[116]	1	M[124]

Important Formulas

$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$