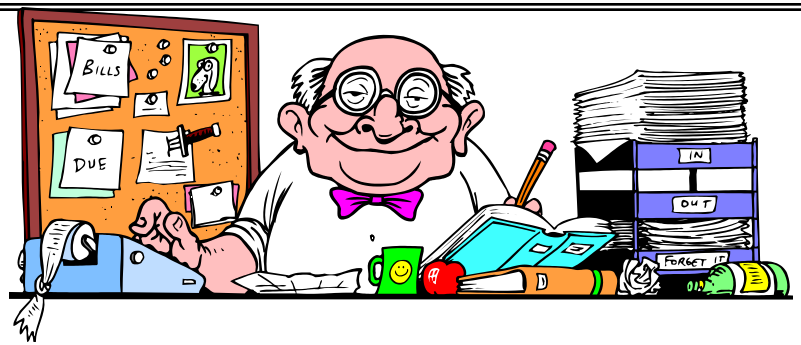
Caching

Edited slide from http://cs162.eecs.Berkeley.edu

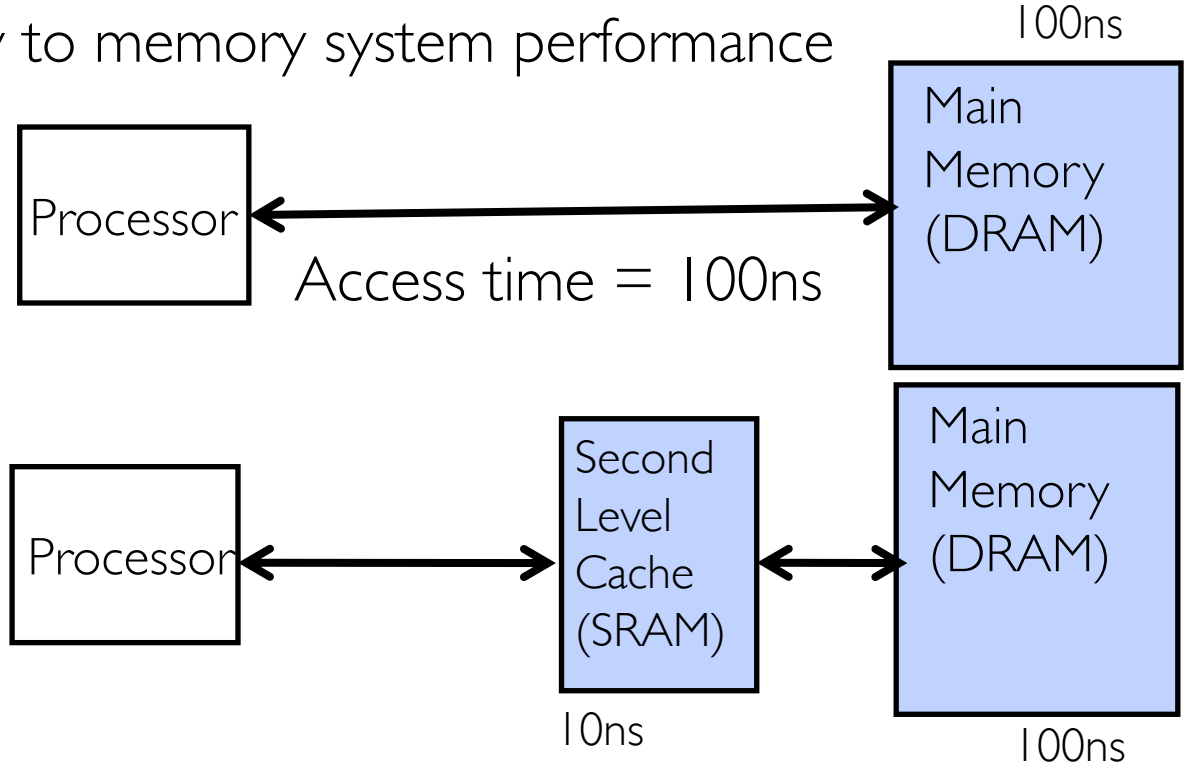
Caching Concept



- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Recall: In Machine Structures (eg. 61C) …

- Caching is the key to memory system performance



Average Access time=(Hit Rate x HitTime) + (Miss Rate x MissTime)

HitRate + MissRate = 1

HitRate = 90% => Avg. Access Time=(0.9 x 10) + (0.1 x 100)=19ns

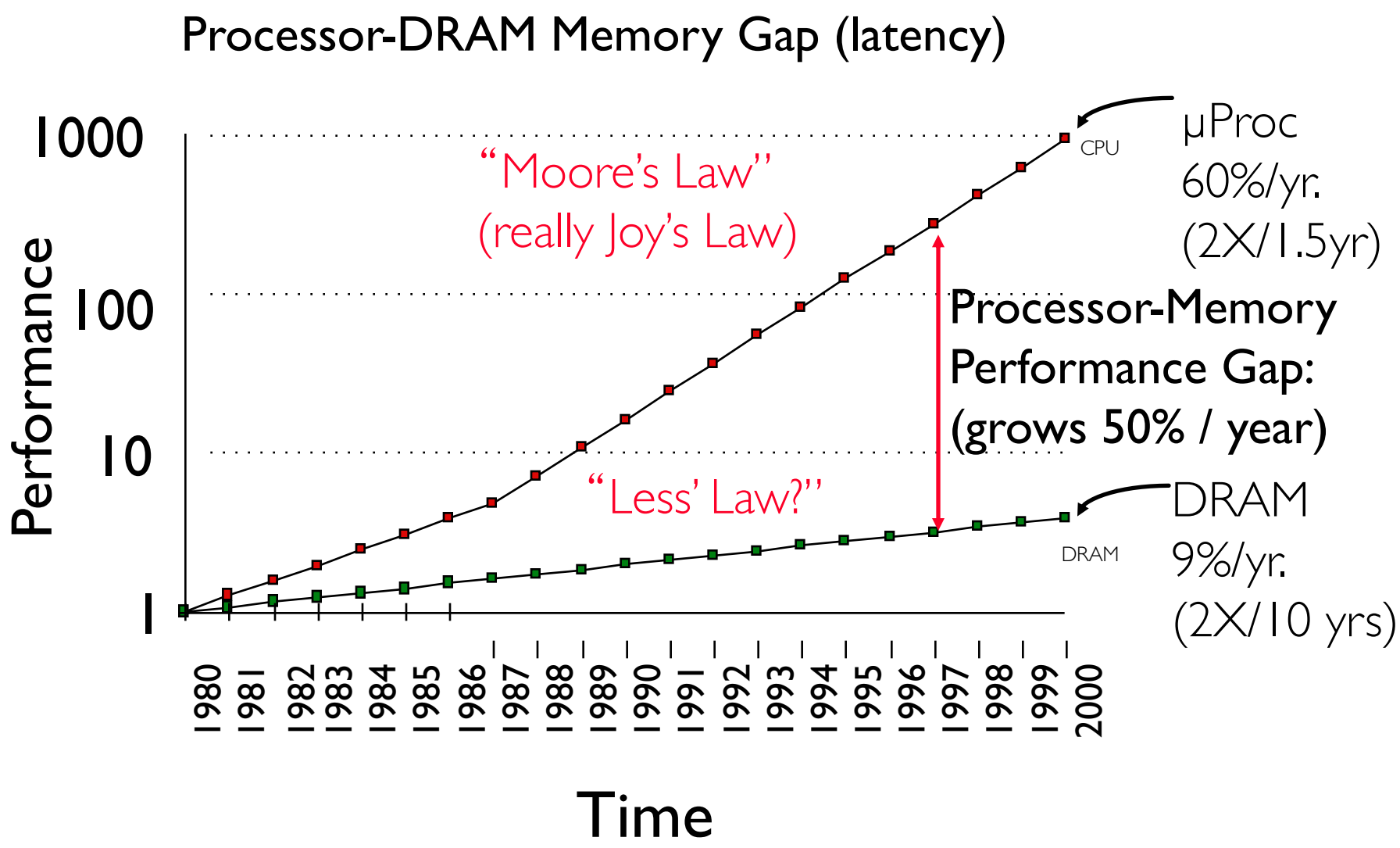
HitRate = 99% => Avg. Access Time=(0.99 x 10) + (0.01 x 100)=10.9 ns

The role of a cache is to reduce the average memory access time.
Here we have a processor; if it directly needs to go to memory to access something, it takes 100ns.
Remember, processors are superfast, they compute in small cycles, and hence 100ns is really lonag time for them.

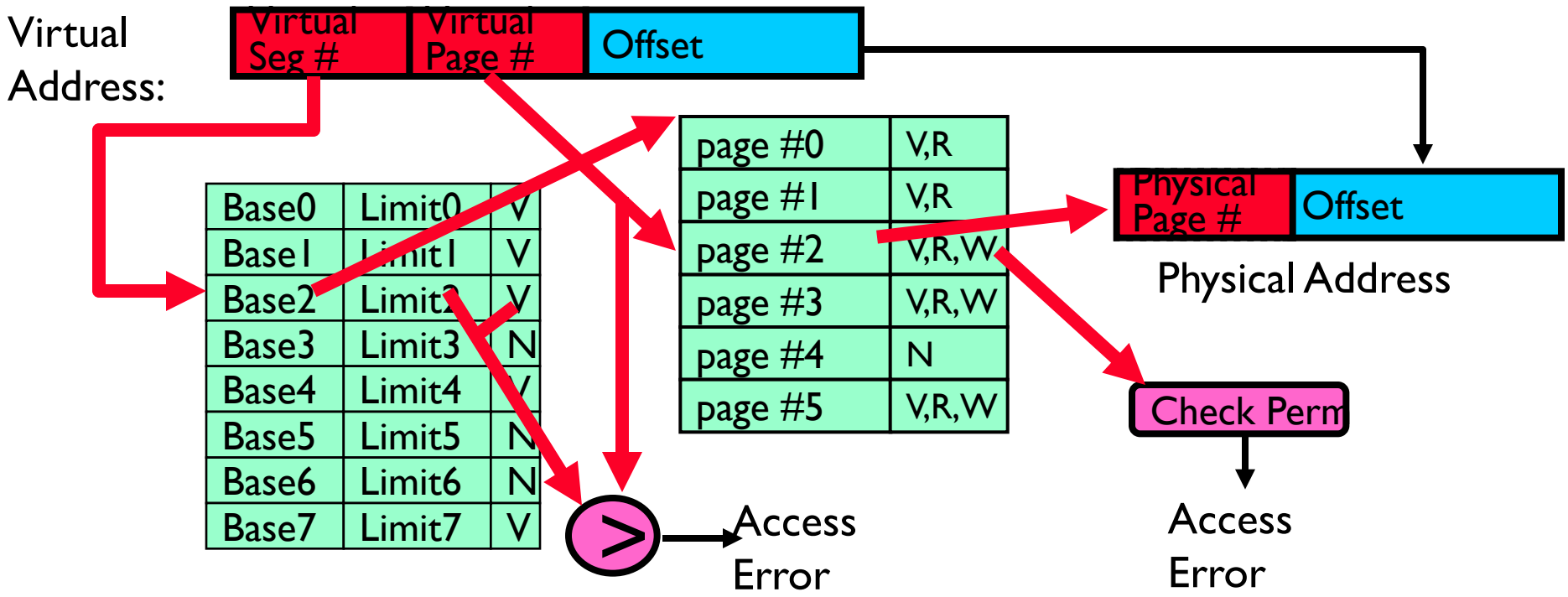
If we add a second level cache, we can instead access the same data in 10ns, i.e.,10 times faster than directly going to memory.
To quantify the gain due to a cache, we can compute the average access time: and its simply the hit-rate (times we find it in the cache).
...

So, we get an illusion that we can access the main memory at such a fast speed, due to caching.

Why Bother with Caching?

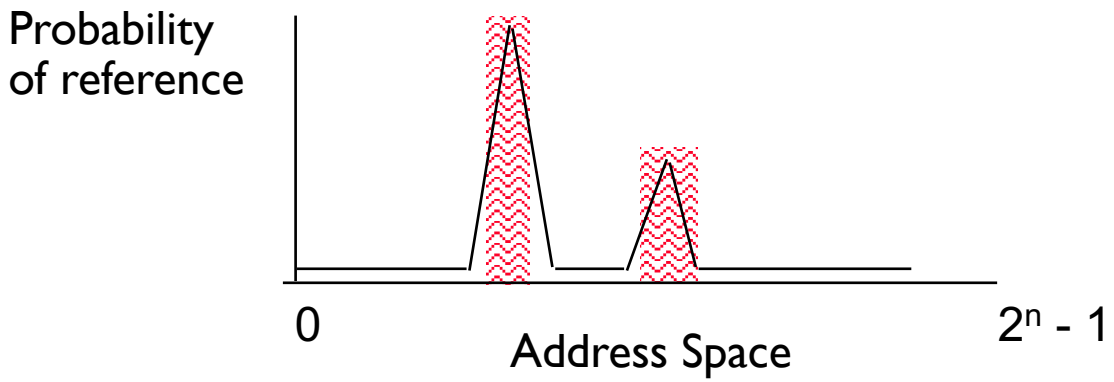


Another Major Reason to Deal with Caching

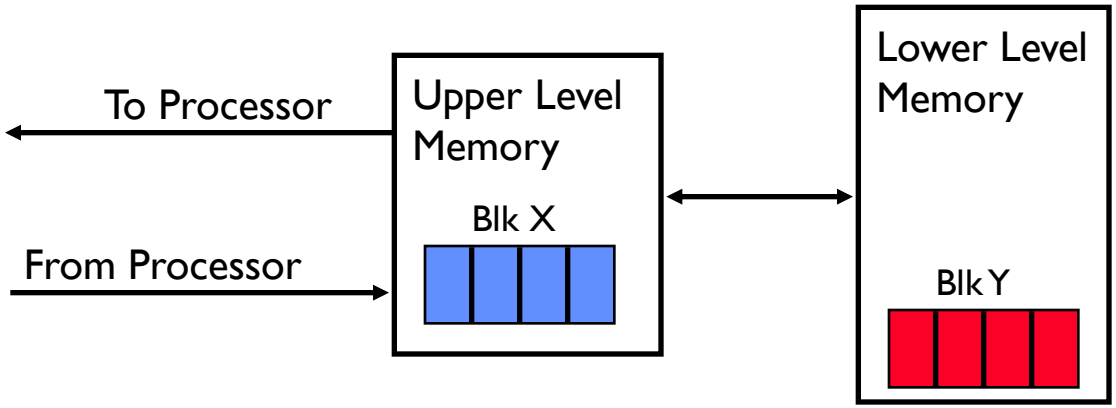


- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

Why Does Caching Help? Locality!



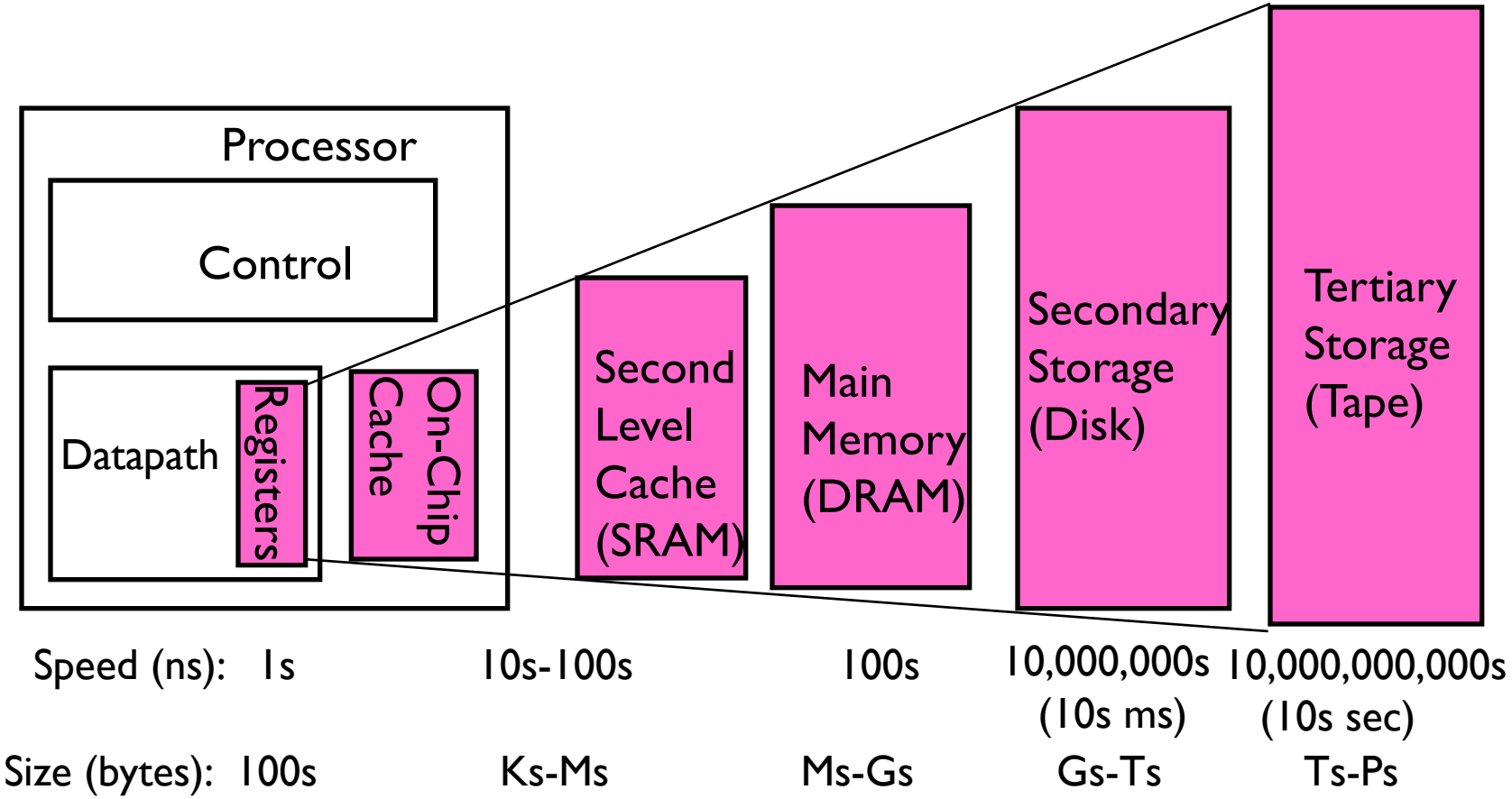
- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



How does the memory hierarchy work? Well it is rather simple, at least in principle.
In order to take advantage of the temporal locality, that is the locality in time, the memory hierarchy will keep those more recently accessed data items closer to the processor because chances are (points to the principle), the processor will access them again soon.
In order to take advantage of the spatial locality, not ONLY do we move the item that has just been accessed to the upper level, but we ALSO move the data items that are adjacent to it.

Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

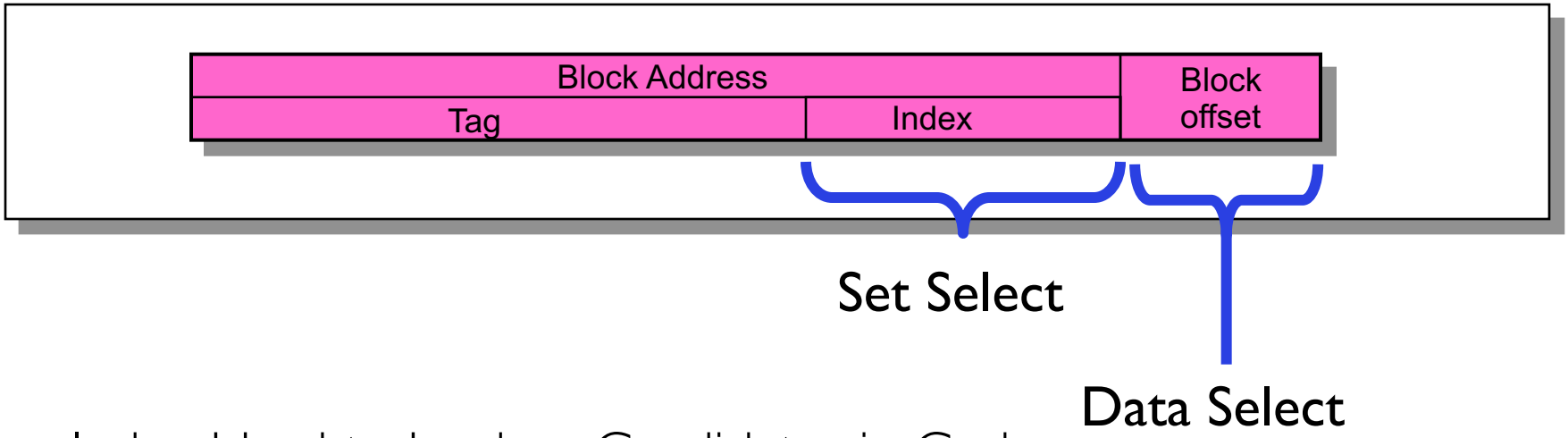


A Summary on Sources of Cache Misses

- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation): other process (e.g., I/O) updates memory

(Capacity miss) That is the cache misses are due to the fact that the cache is simply not large enough to contain all the blocks that are accessed by the program.
The solution to reduce the Capacity miss rate is simple: increase the cache size.
Here is a summary of other types of cache miss we talked about.
First is the Compulsory misses. These are the misses that we cannot avoid. They are caused when we first start the program.
Then we talked about the conflict misses. They are the misses that caused by multiple memory locations being mapped to the same cache location.
There are two solutions to reduce conflict misses. The first one is, once again, increase the cache size. The second one is to increase the associativity.
For example, say using a 2-way set associative cache instead of directed mapped cache.
But keep in mind that cache miss rate is only one part of the equation. You also have to worry about cache access time and miss penalty. Do NOT optimize miss rate alone.
Finally, there is another source of cache miss we will not cover today. Those are referred to as invalidation misses caused by another process, such as IO , update the main memory so you have to flush the cache to avoid inconsistency between memory and cache.

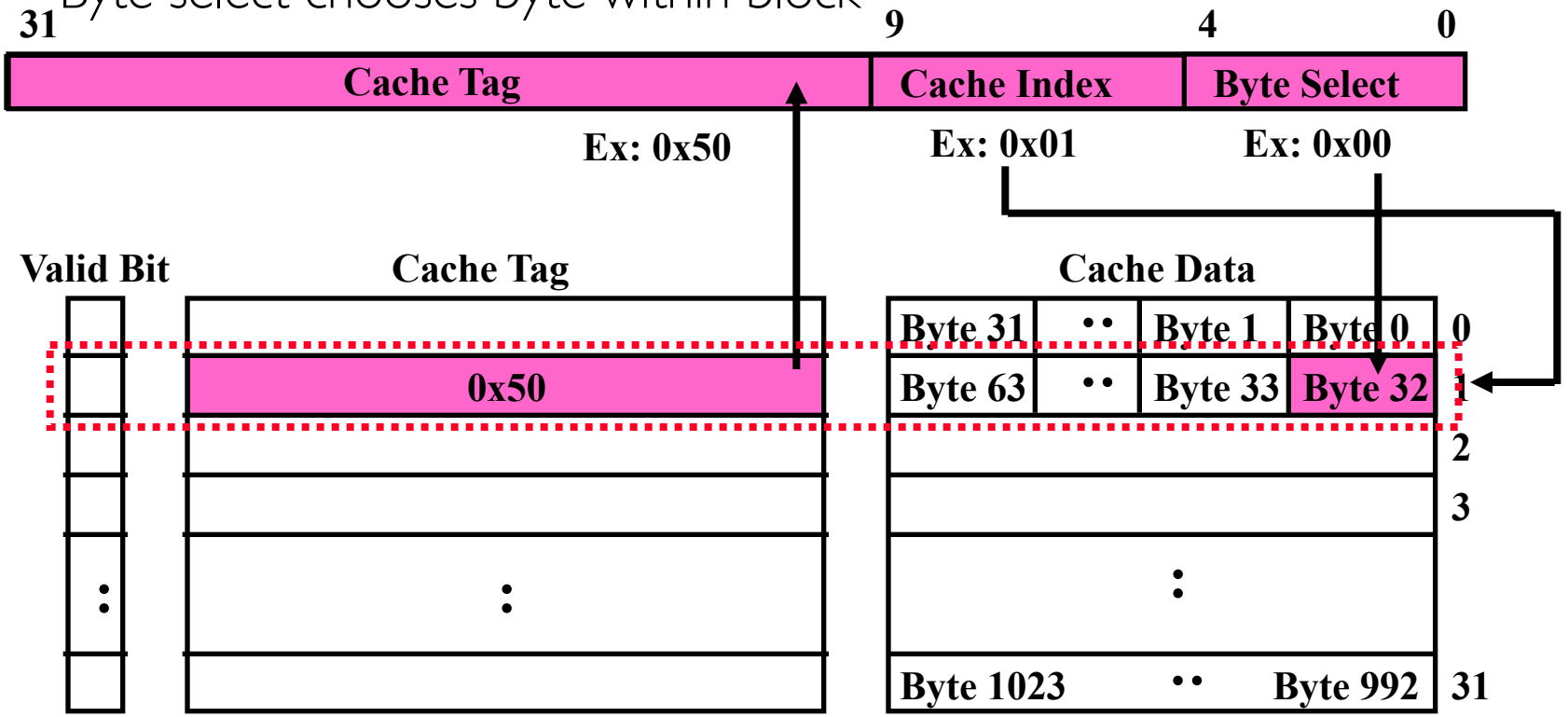
How is a Block found in a Cache?



- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field

Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block



Let's use a specific example with realistic numbers: assume we have a 1 KB direct mapped cache with block size equals to 32 bytes.

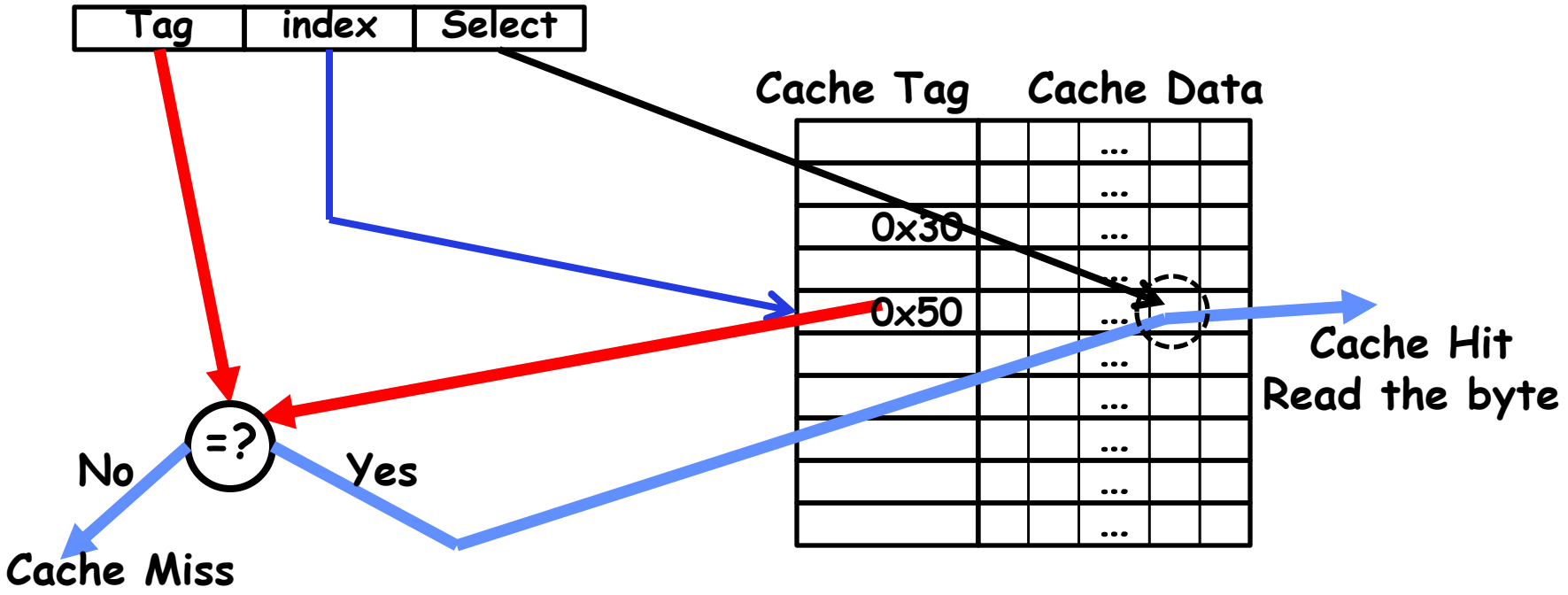
In other words, each block associated with the cache tag will have 32 bytes in it (Row 1).

With Block Size equals to 32 bytes, the 5 least significant bits of the address will be used as byte select within the cache block.

Since the cache size is 1K byte, the upper 32 minus 10 bits, or 22 bits of the address will be stored as cache tag.

The rest of the address bits in the middle, that is bit 5 through 9, will be used as Cache Index to select the proper cache entry.

Direct Mapped Cache: Example

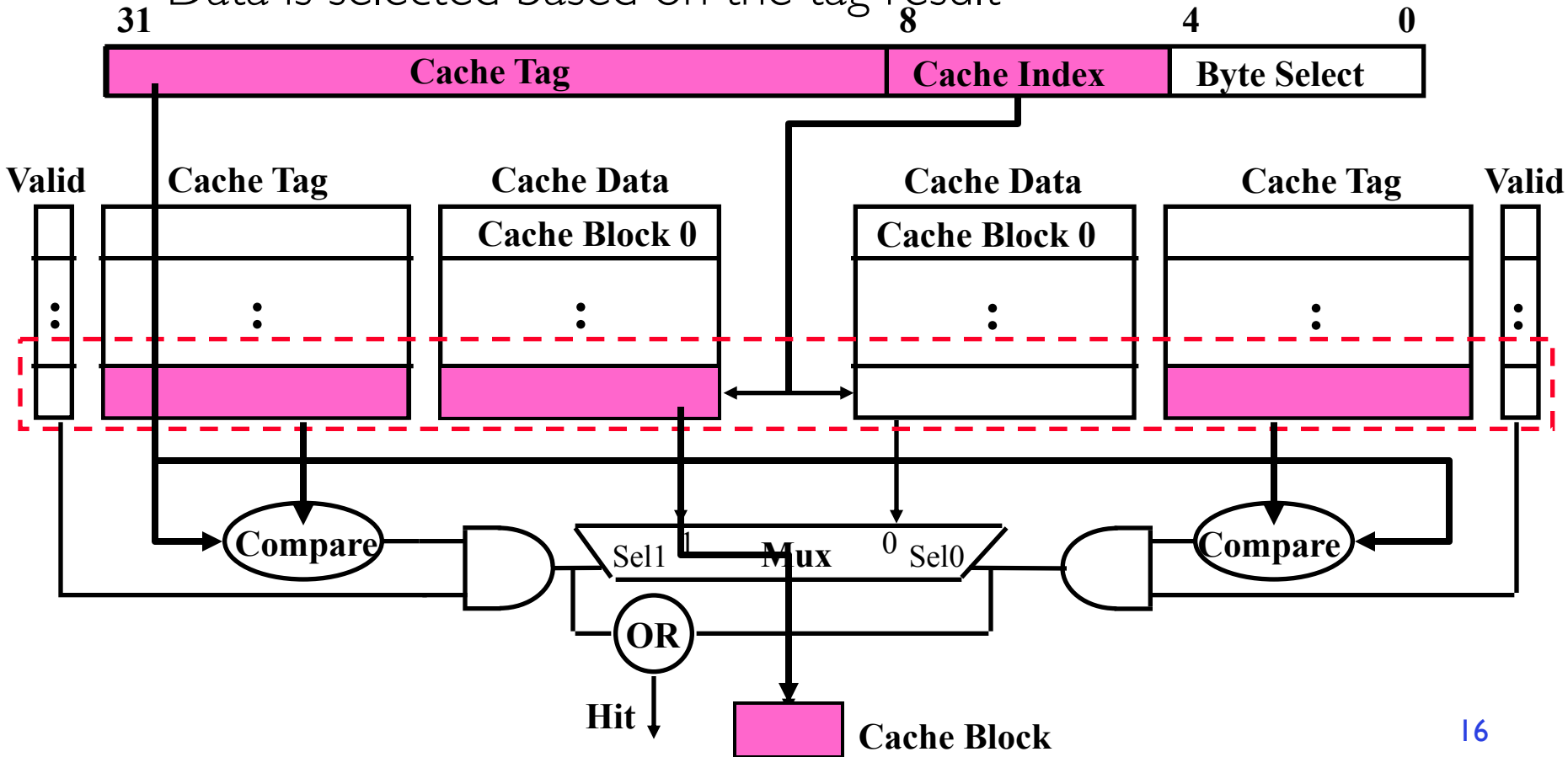


Reference stream

<0x50,0x04,0x00>
<0x30,0x02,0x00>
<0x50,0x04,0x03>
<0x30,0x02,0x02>
...
<0x60,0x04,0x00>
<0x30,0x02,0x02>
<0x60,0x04,0x01>
<0x50,0x04,0x05>

Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



This is called a 2-way set associative cache because there are two cache entries for each cache index. Essentially, you have two direct mapped cache works in parallel.

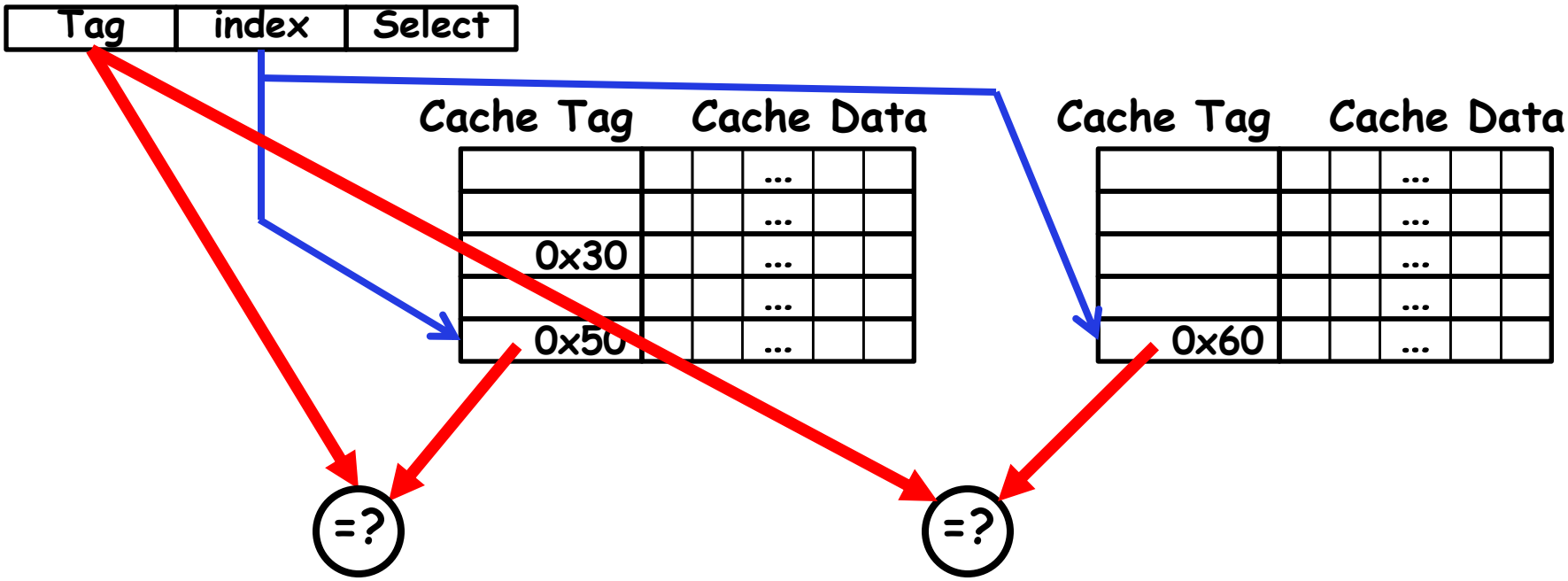
This is how it works: the cache index selects a set from the cache. The two tags in the set are compared in parallel with the upper bits of the memory address.

If neither tag matches the incoming address tag, we have a cache miss.

Otherwise, we have a cache hit and we will select the data on the side where the tag matches occur.

This is simple enough. What is its disadvantages?

Set Associative Cache: Example(2-Way)

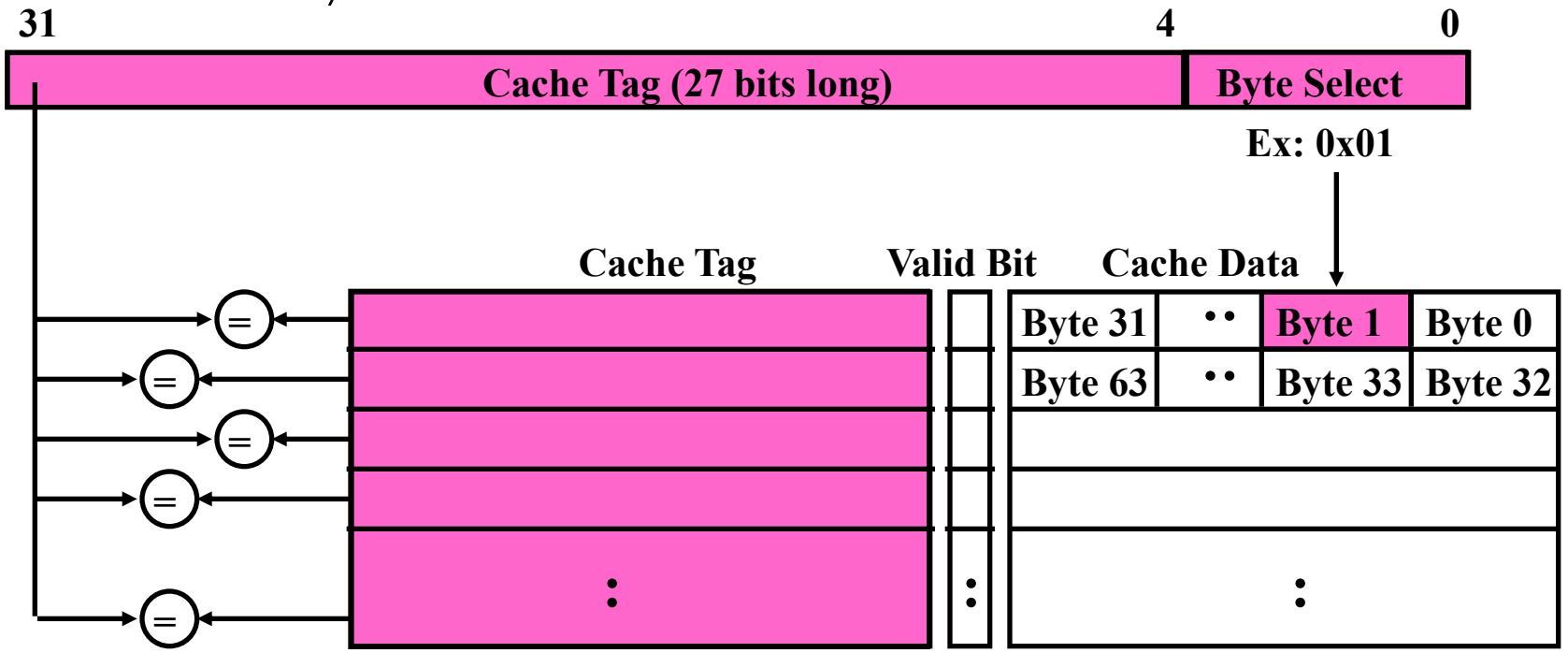


Reference stream

<0x50,0x04,0x00>
<0x30,0x02,0x00>
<0x50,0x04,0x03>
<0x30,0x02,0x02>
...
<0x60,0x04,0x00>
<0x30,0x02,0x02>
<0x60,0x04,0x01>
<0x50,0x04,0x05>

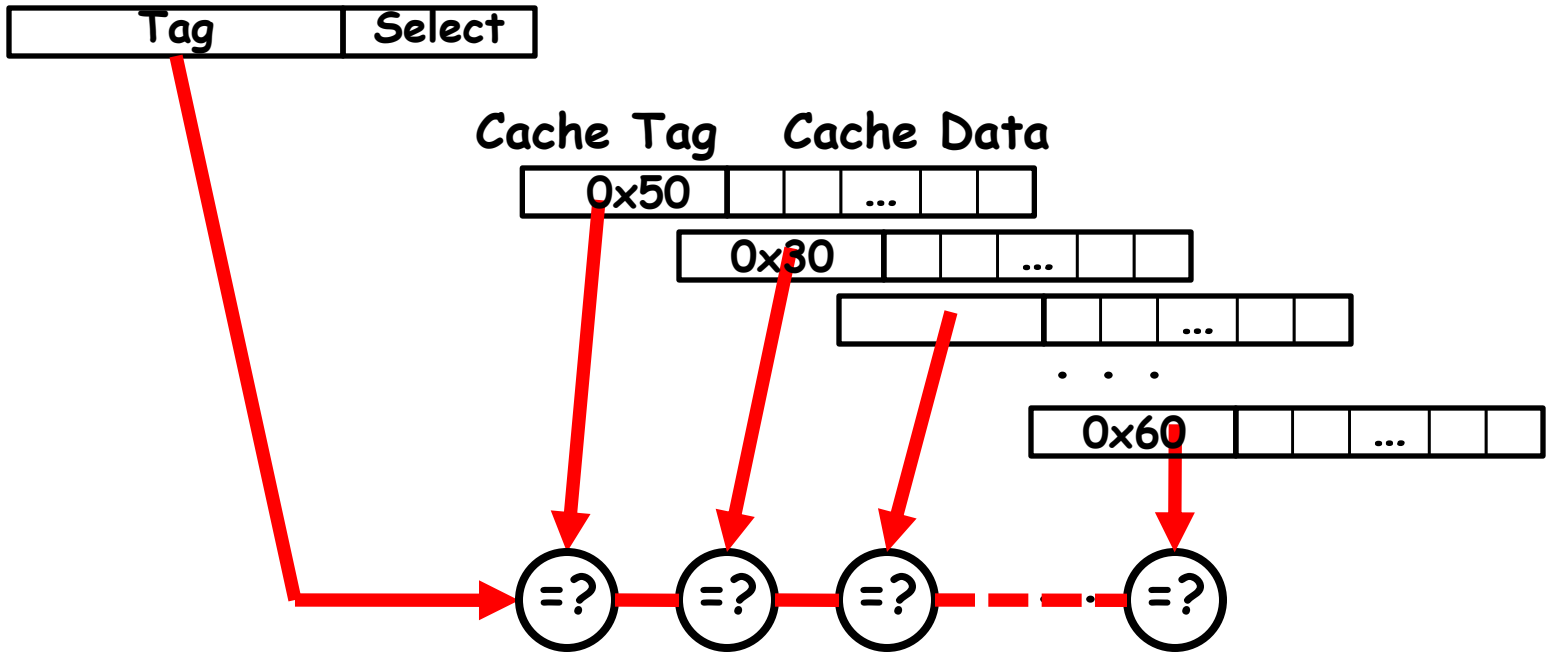
Review: Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



While the direct mapped cache is on the simple end of the cache design spectrum, the fully associative cache is on the most complex end.
It is the N-way set associative cache carried to the extreme where N in this case is set to the number of cache entries in the cache.
In other words, we don't even bother to use any address bits as the cache index.
We just store all the upper bits of the address (except Byte select) that is associated with the cache block as the cache tag and have one comparator for every entry.
The address is sent to all entries at once and compared in parallel and only the one that matches are sent to the output. This is called an associative lookup.
Needless to say, it is very hardware intensive. Usually, fully associative cache is limited to 64 or less entries.
Since we are not doing any mapping with the cache index, we will never push any other item out of the cache because multiple memory locations map to the same cache location.
Therefore, by definition, conflict miss is zero for a fully associative cache.
This, however, does not mean the overall miss rate will be zero.
Assume we have 64 entries here. The first 64 items we accessed can fit in.
But when we try to bring in the 65th item, we will need to throw one of them out to make room for the new item. This bring us to the third type of cache misses: Capacity Miss.

Fully Associative Cache: Example



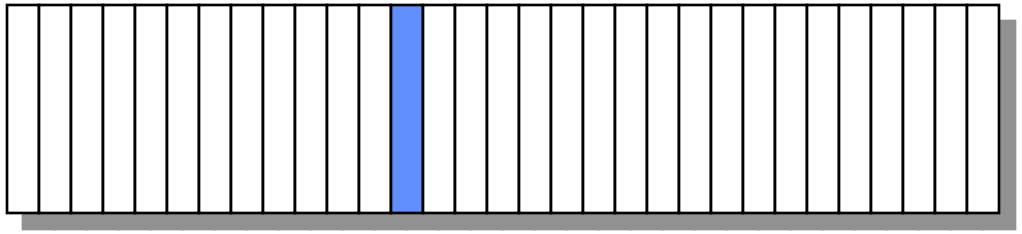
Reference stream

<0x50,0x04,0x00>
<0x30,0x02,0x00>
<0x50,0x04,0x03>
<0x30,0x02,0x02>
...
<0x60,0x04,0x00>
<0x30,0x02,0x02>
<0x60,0x04,0x01>
<0x50,0x04,0x05>

Where does a Block Get Placed in a Cache?

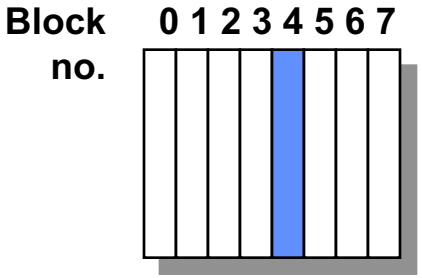
- Example: Block 12 placed in 8 block cache

32-Block Address Space:



Block no. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

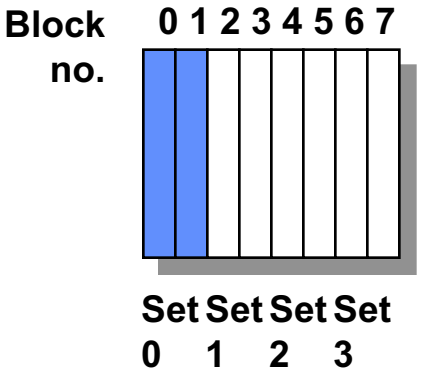
Direct mapped:
block 12 can go only into block 4 (12 mod 8)



12=1100

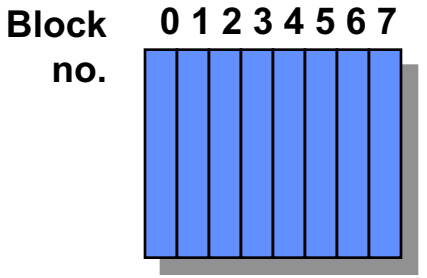
Tag:1, Index: 100

Set associative:
block 12 can go anywhere in set 0 (12 mod 4)



Tag:11, Index: 00

Fully associative:
block 12 can go anywhere



Tag:1100, Index:NA

Review: Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

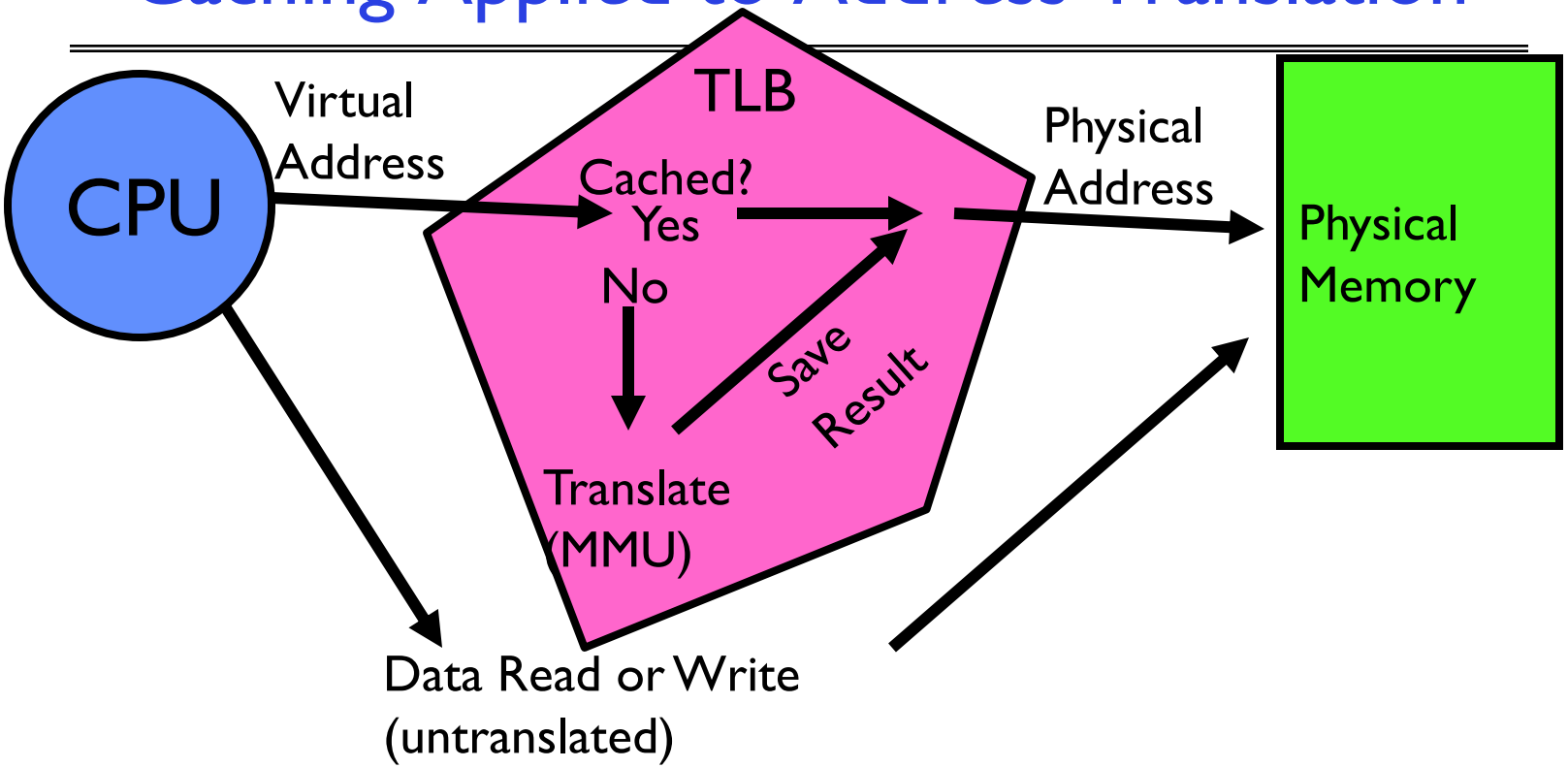
- Miss rates for a workload:

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Review: What happens on a write?

- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: Processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM
      processor not held up on writes
    - » CON: More complex
      Read miss may require writeback of dirty data

Caching Applied to Address Translation



- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

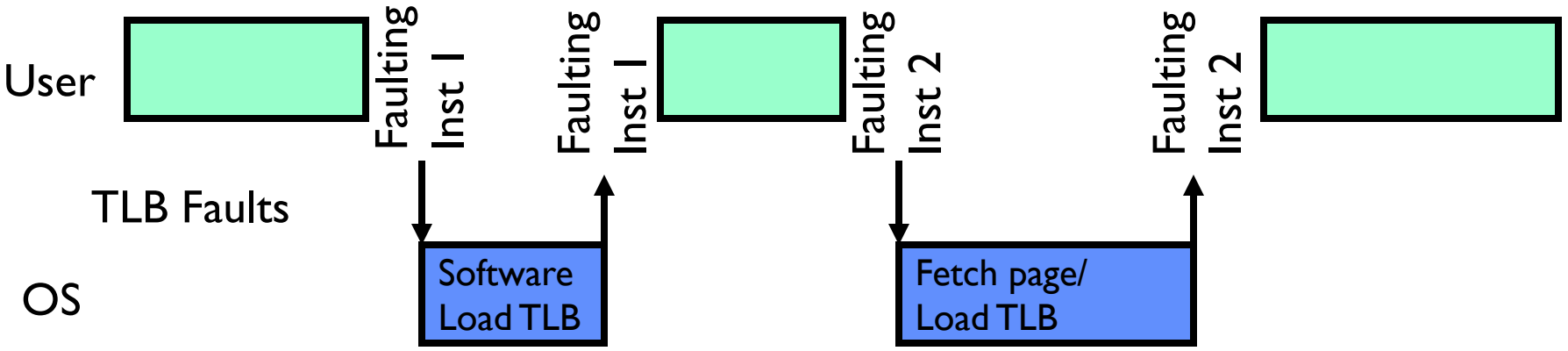
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - » shared segments
    - » user-level portions of an operating system

What happens on a Context Switch?

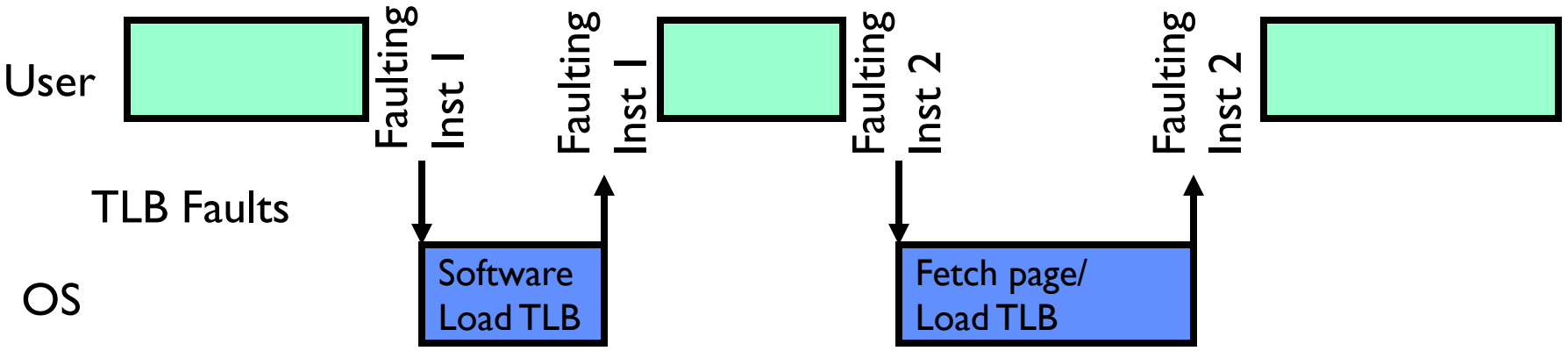
- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

Transparent Exceptions: TLB/Page fault (1/2)



- How to transparently restart faulting instructions?
  - (Consider load or store that gets TLB or Page fault)
  - Could we just skip faulting instruction?
    - » No: need to perform load or store after reconnecting physical page

Transparent Exceptions: TLB/Page fault (2/2)



- Hardware must help out by saving:
  - Faulting instruction and partial state
    - » Need to know which instruction caused fault
    - » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    - » Save/restore registers, stack, etc
- What if an instruction has side-effects?

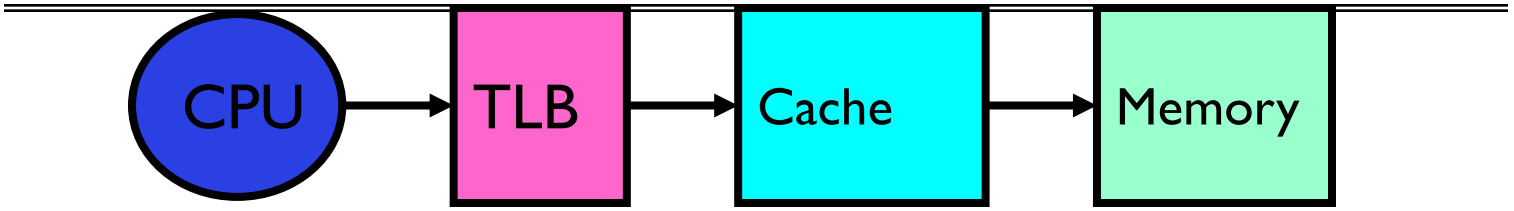
Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    - » Unwind side-effects (easy to restart)
    - » Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`
    - » What if page fault occurs when write to stack pointer?
    - » Did `sp` get incremented before or after the page fault?
  - Example 2: `strcpy (r1), (r2)`
    - » Source and destination overlap: can't unwind in principle!
    - » IBM S/370 and VAX solution: execute twice – once read-only
- What about "RISC" processors?
  - For instance delayed branches?
    - » Example:
      ```
      bne somewhere
      ld r1,(sp)
      ```
    - » Precise exception state consists of two PCs: PC and nPC (next PC)
  - Delayed exceptions:
    - » Example:
      ```
      div r1, r2, r3
      ld r1, (sp)
      ```
    - » What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

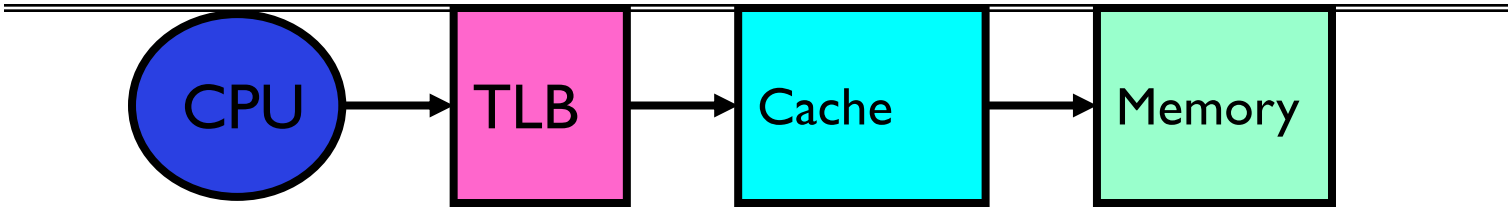
- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead to forsaking precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Recall: TLB Organization

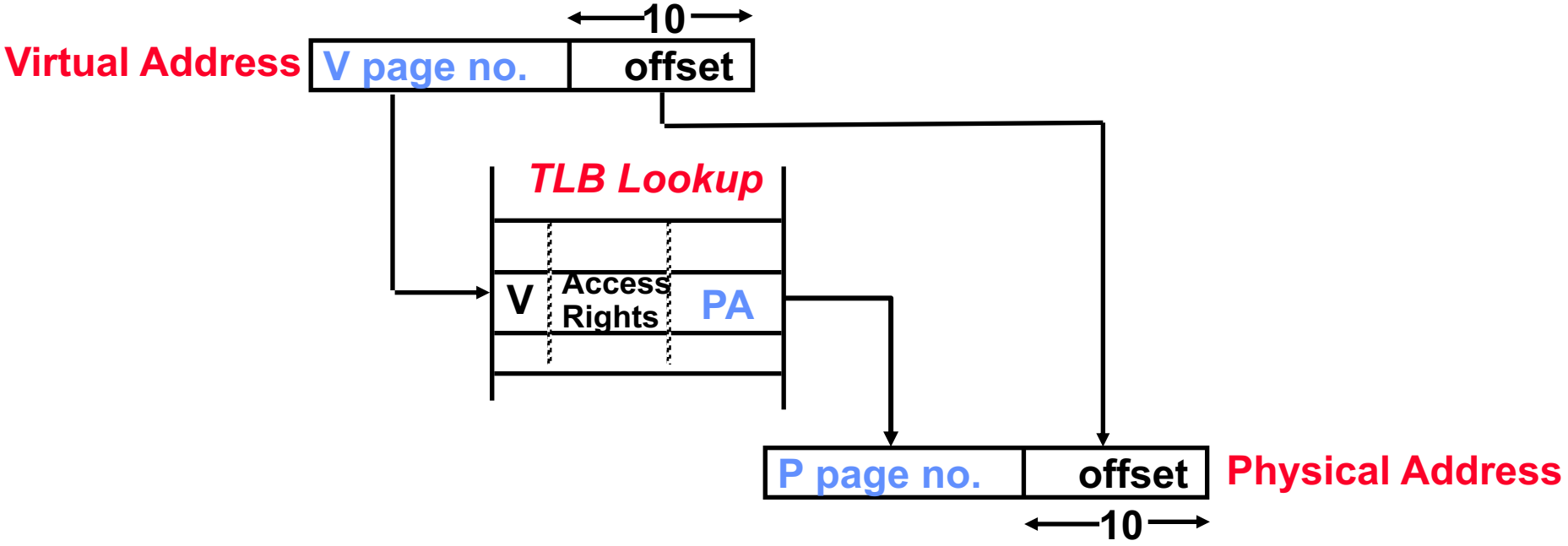


- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

Reducing translation time further



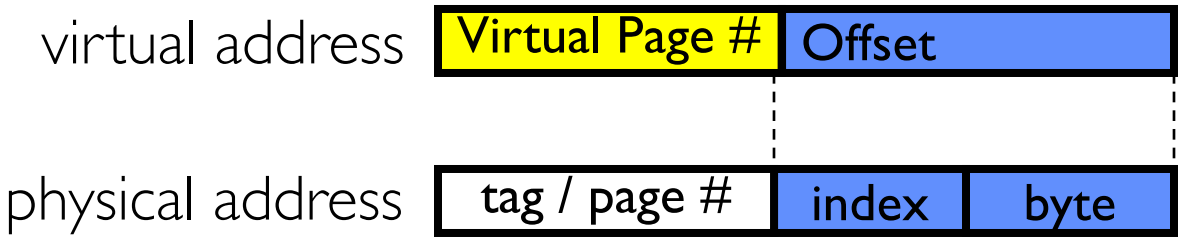
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

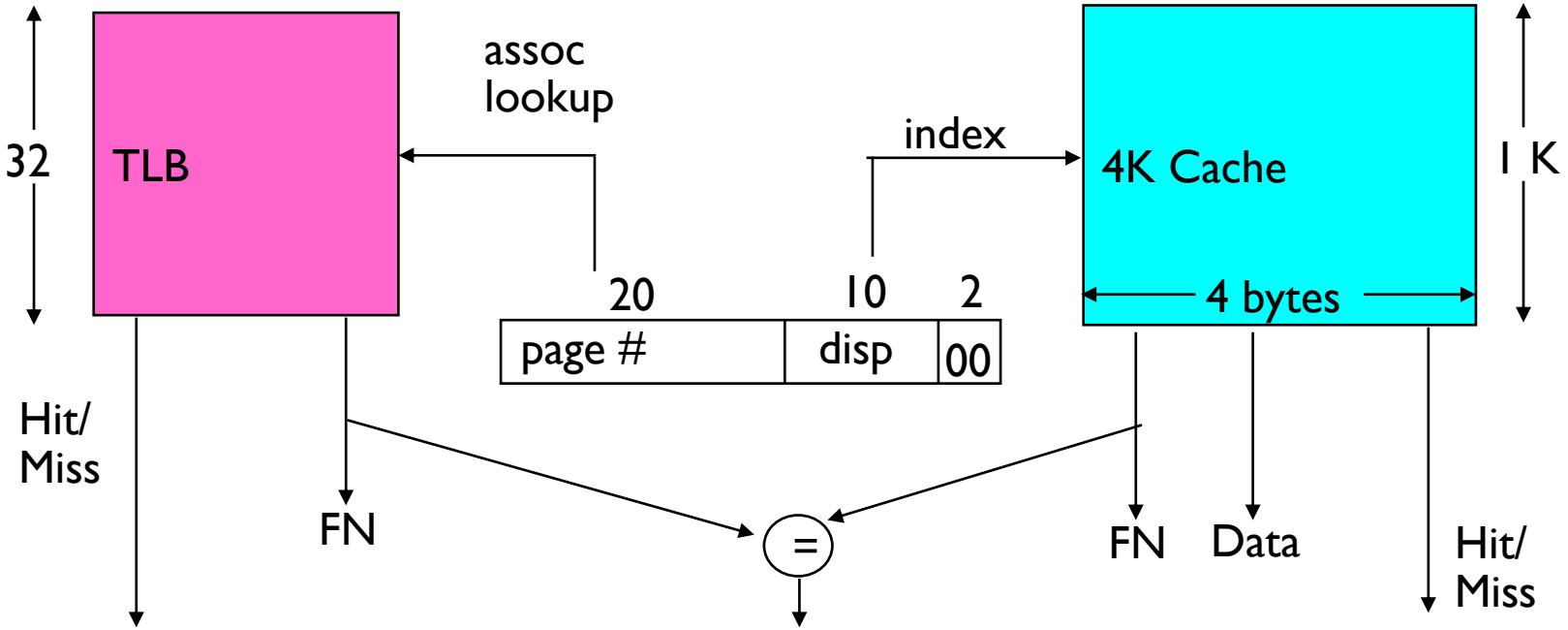
Overlapping TLB & Cache Access (1/2)

- Main idea:
  - Offset in virtual address exactly covers the "cache index" and "byte select"
  - Thus can select the cached byte(s) in parallel to perform address translation



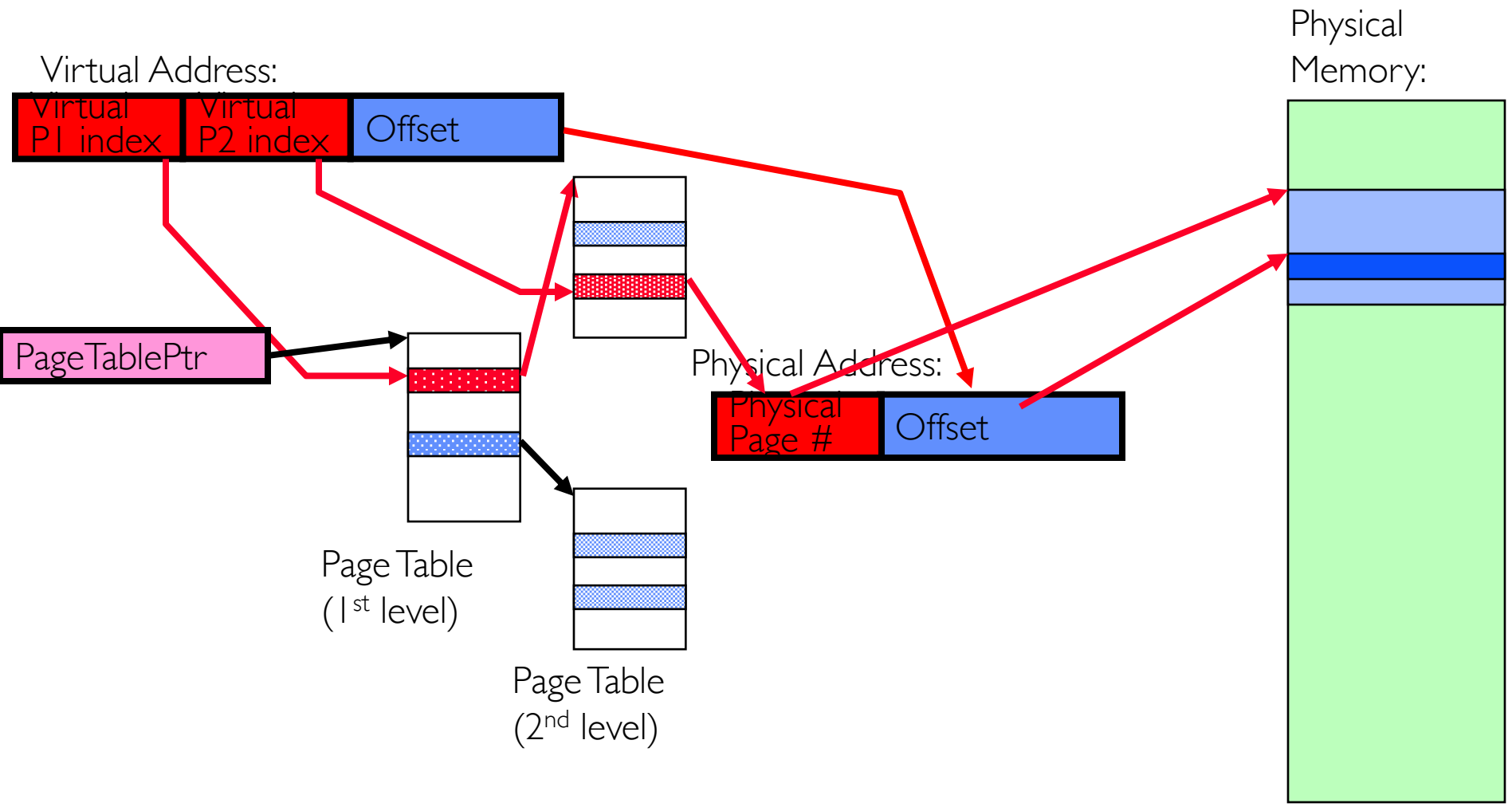
Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

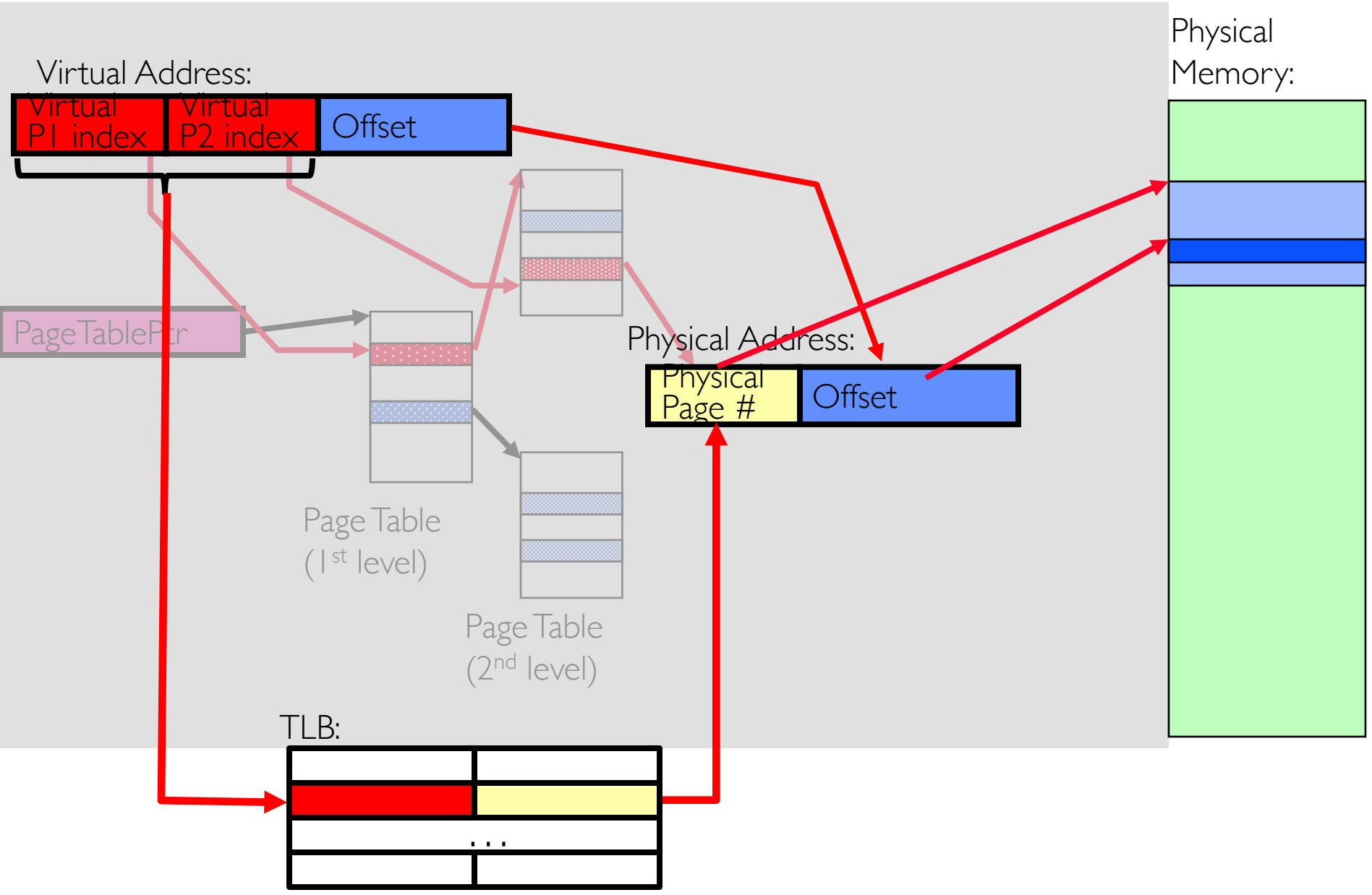


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

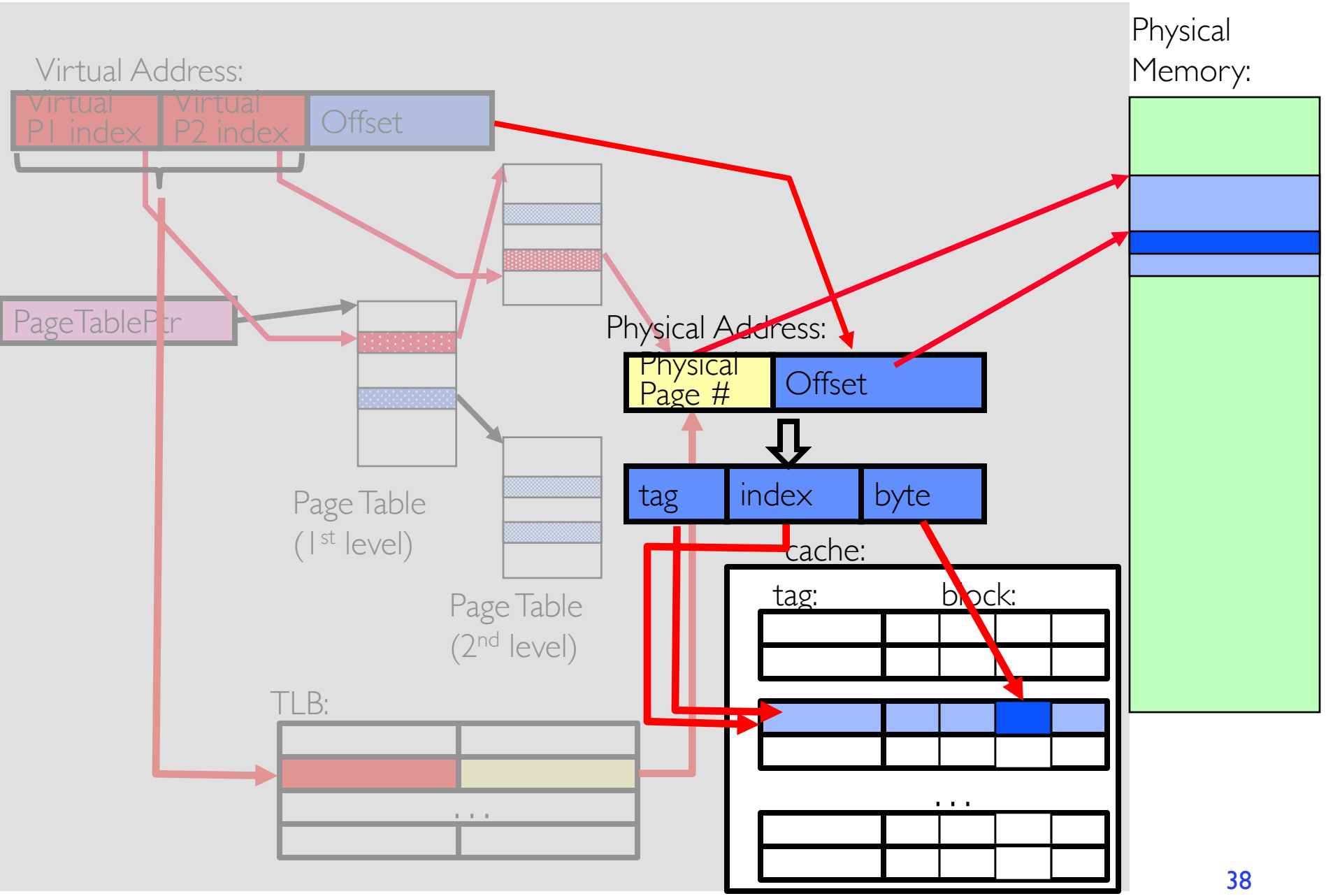
Putting Everything Together: Address Translation



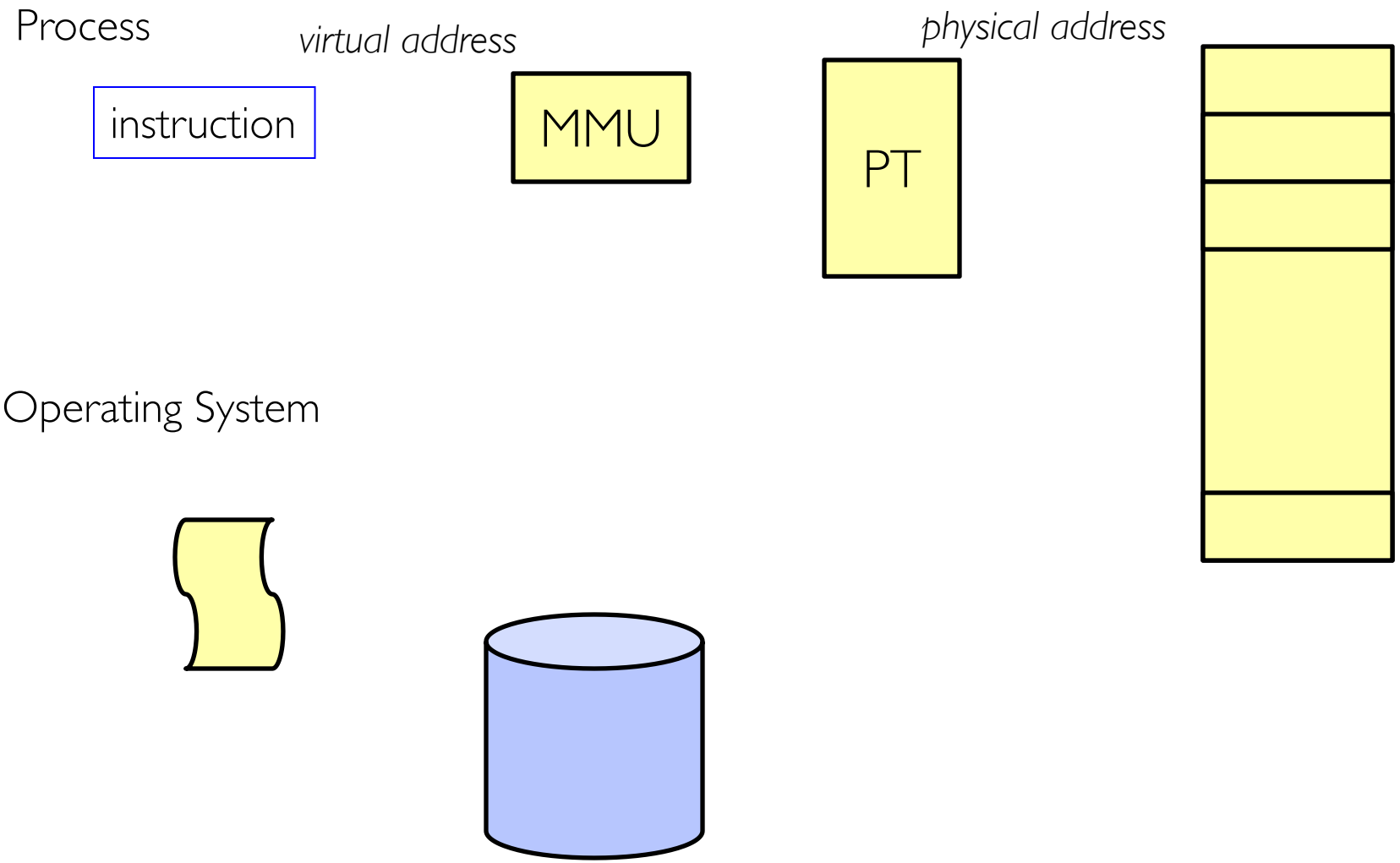
Putting Everything Together: TLB



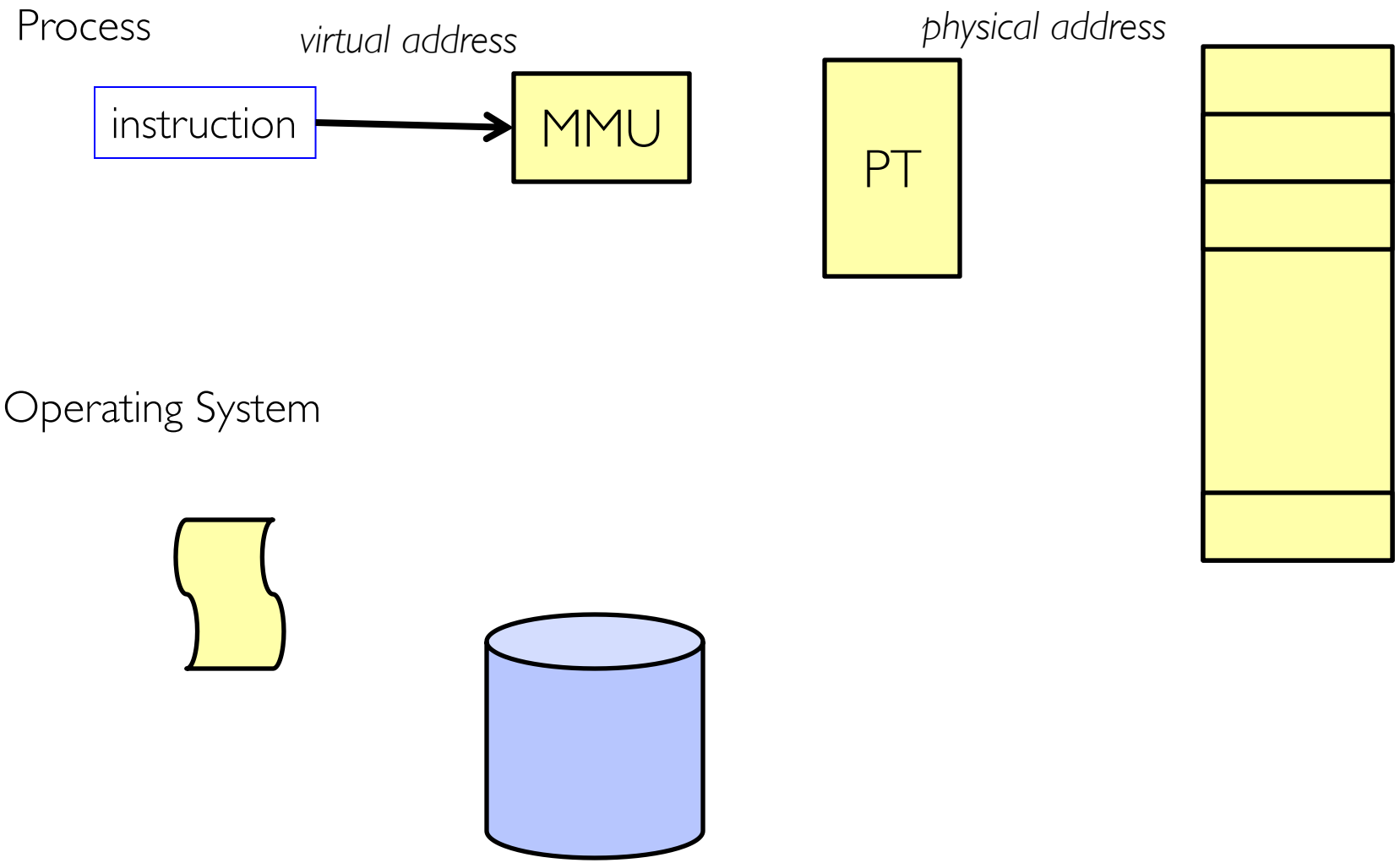
Putting Everything Together: Cache



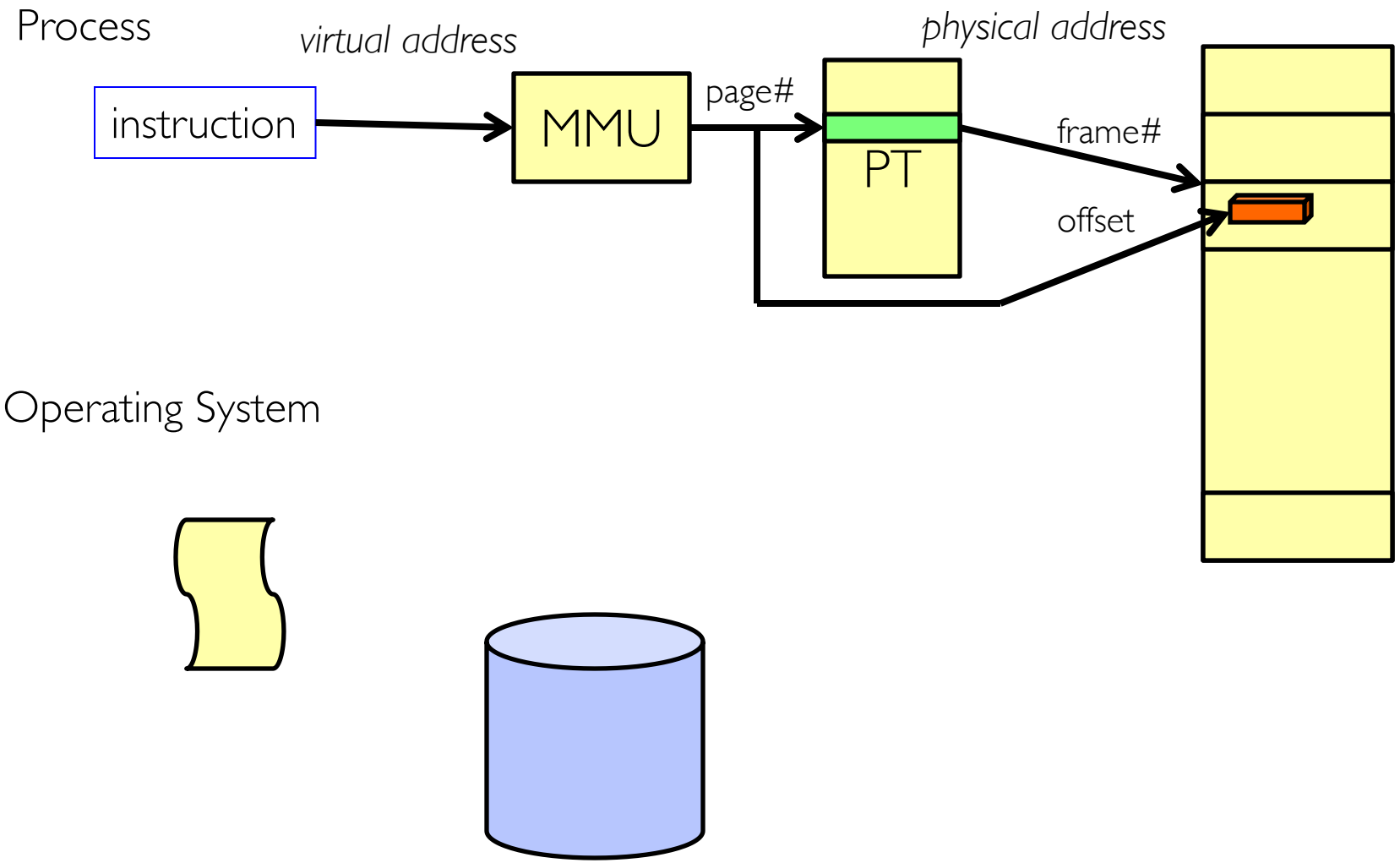
Next Up: What happens when …



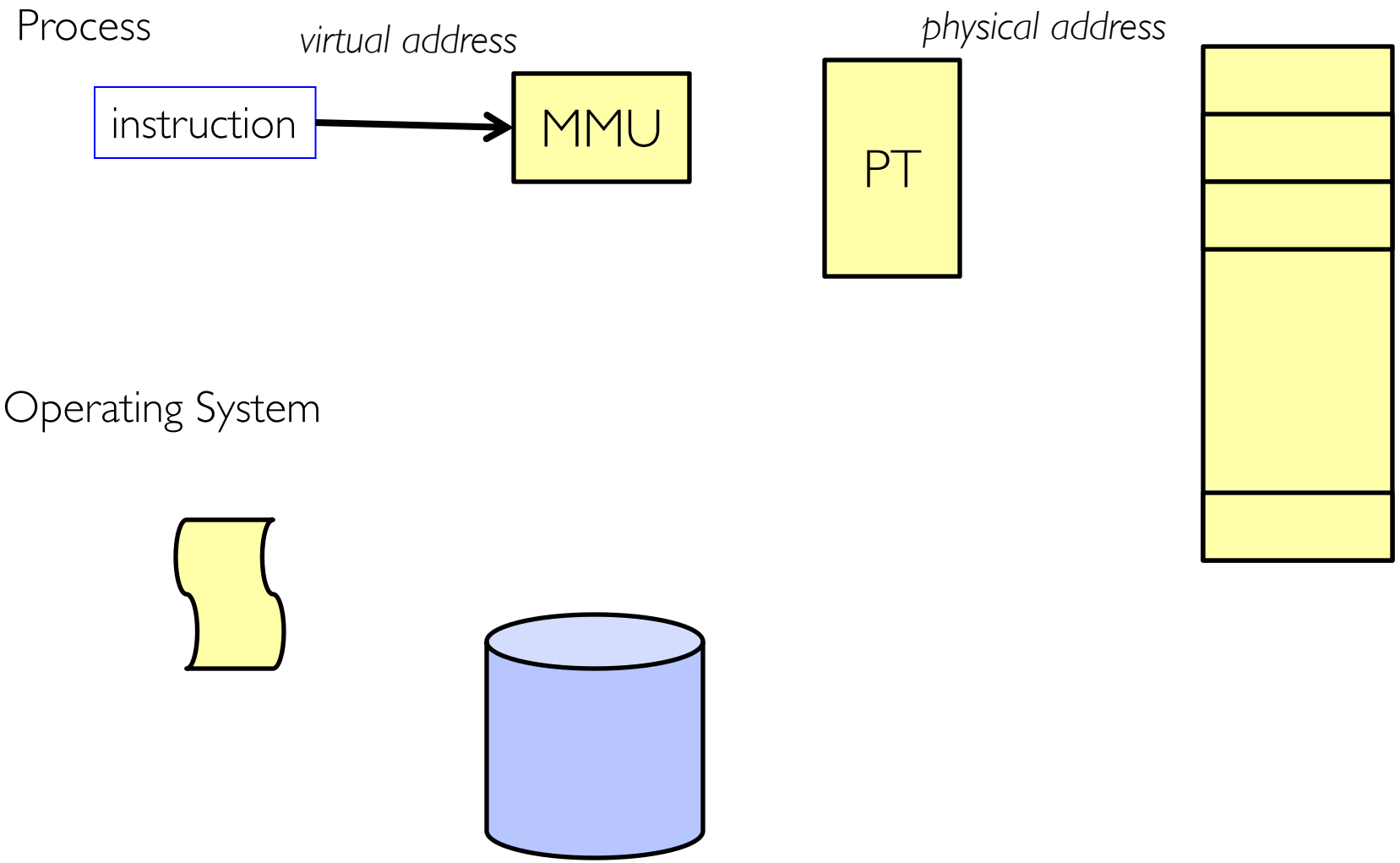
Next Up: What happens when …



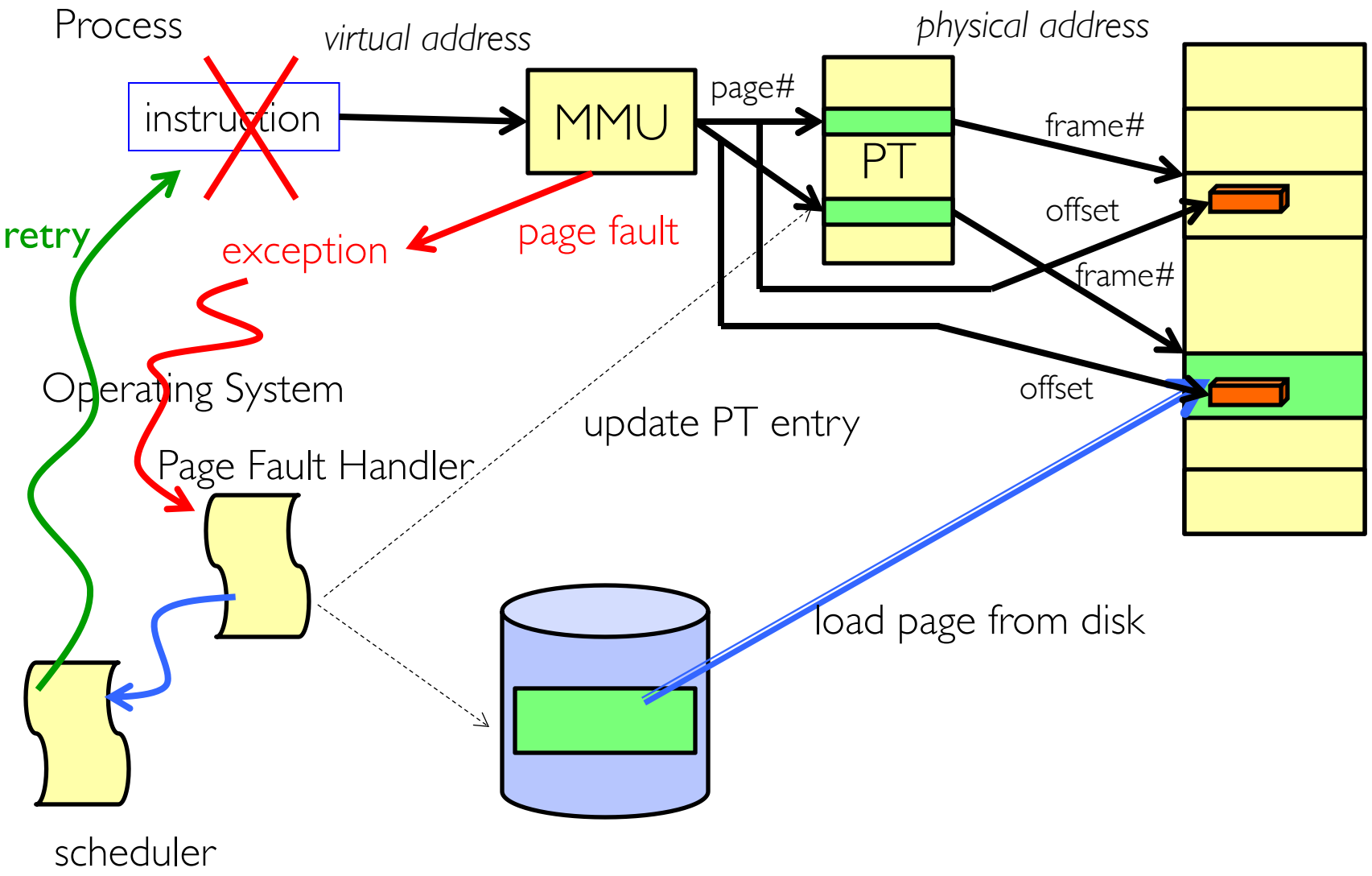
Next Up: What happens when …



Next Up: What happens when …



Next Up: What happens when …



Summary (1/2)

- The Principle of Locality:
  - Program likely to 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 (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices

- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

Summary (2/2)

- A cache of translations called a "Translation Lookaside Buffer" (TLB)
  - Relatively small number of PTEs and optional process IDs (< 512)
  - Fully Associative (Since conflict misses expensive)
  - On TLB miss, page table must be traversed and if located PTE is invalid, cause Page Fault
  - On change in page table, TLB entries must be invalidated
  - TLB is logically in front of cache (need to overlap with cache access)
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Can manage caches in hardware or software or both
  - Goal is highest hit rate, even if it means more complex cache management