EE 457 Unit 7d

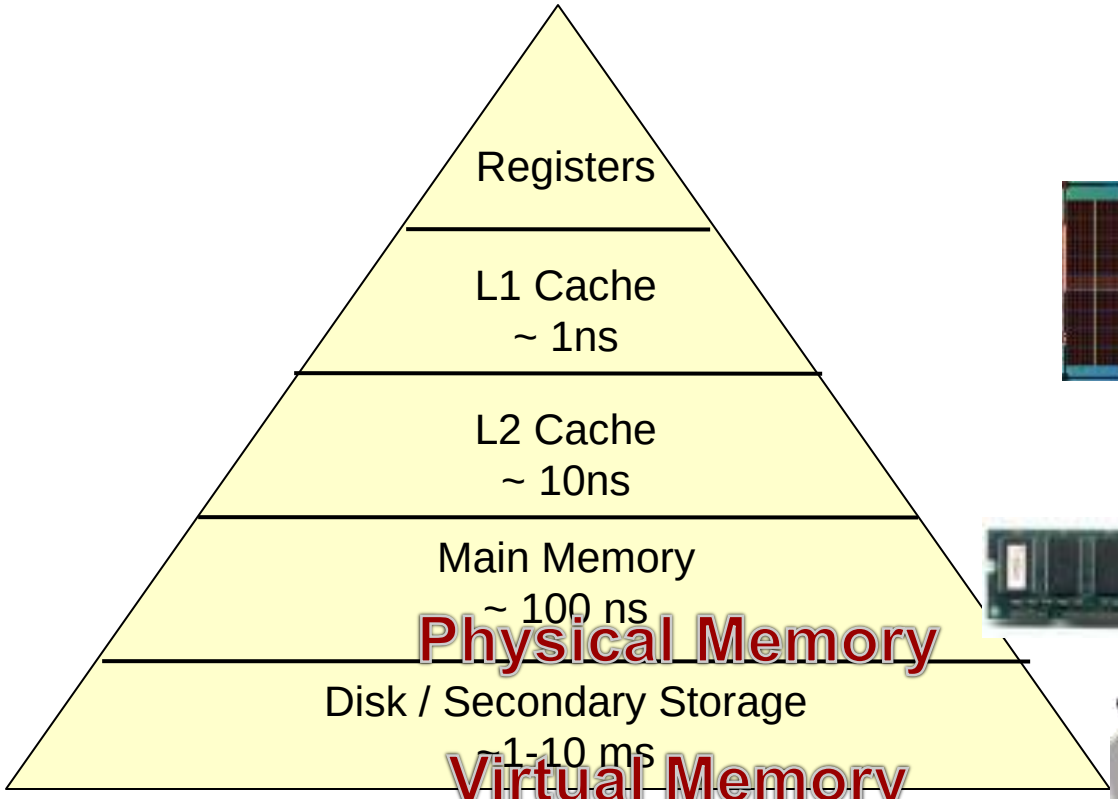
Virtual Memory

Virtual Memory Concept

- A mechanism for hiding the details of how much physical memory exists and how it's being shared
- Allows the OS to
  - Efficiently **share the physical memory** between several running programs/processes and provide **protection** against each other
  - Remove the need of the programmer to know how much memory is physically present and/or give the illusion of **more or less physical memory** than is present
  - Interpose itself whenever the HW performs a memory access (useful for a host of OS features, abstractions, and optimizations, etc.)
- Have software use a set of virtual "fictitious" addresses that the OS can specify how to translate to physical "real" addresses
  - Often includes use of MM as a cache for multiple programs and their data as they run using secondary storage (disk) as the home location

Memory Hierarchy & Caching

- Lower levels act as a cache for upper levels

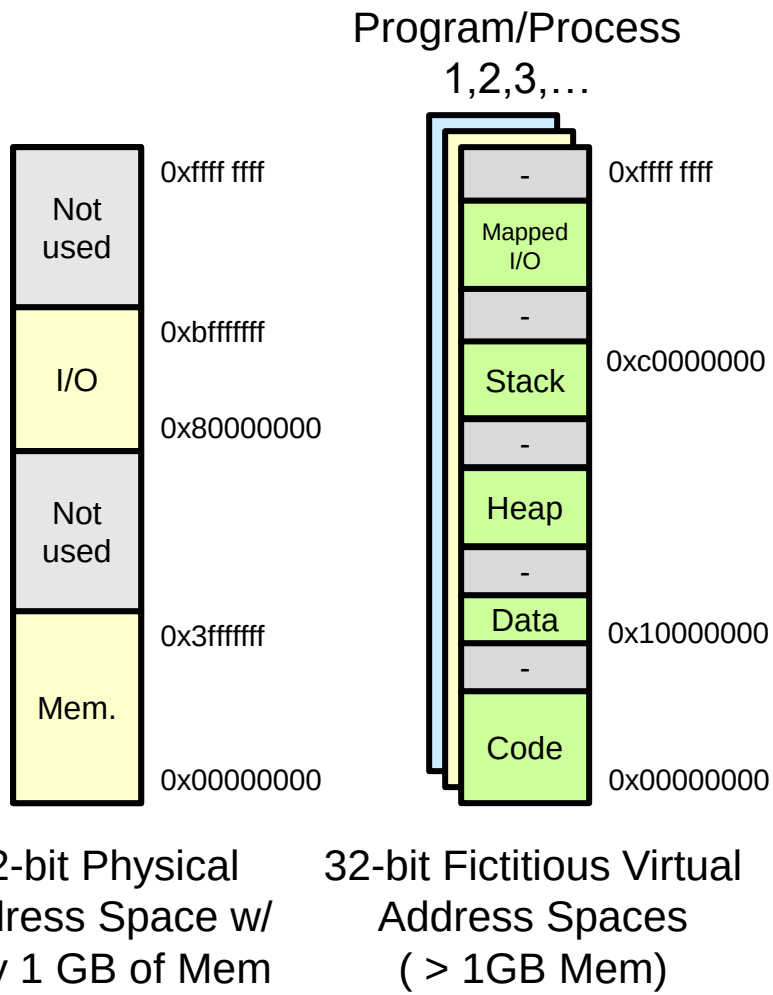


L1/L2 is a "cache" for main memory

Virtual memory provides each process its own address space in secondary storage and uses main memory as a cache

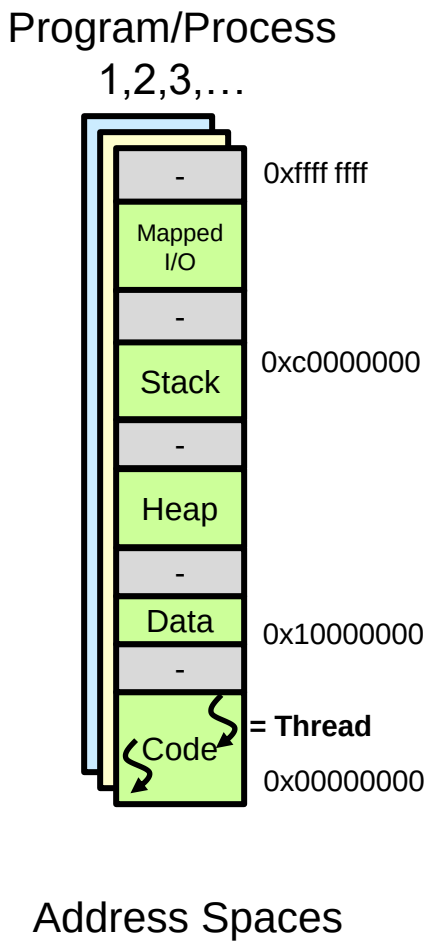
Address Spaces

- Physical address spaces corresponds to the actual system address range (based on the width of the address bus) of the processor and how much main memory is physically present
- Each process/program runs in its own private "virtual" address space
  - Virtual address space can be larger (or smaller) than physical memory
  - Virtual address spaces are protected from each other



Processes

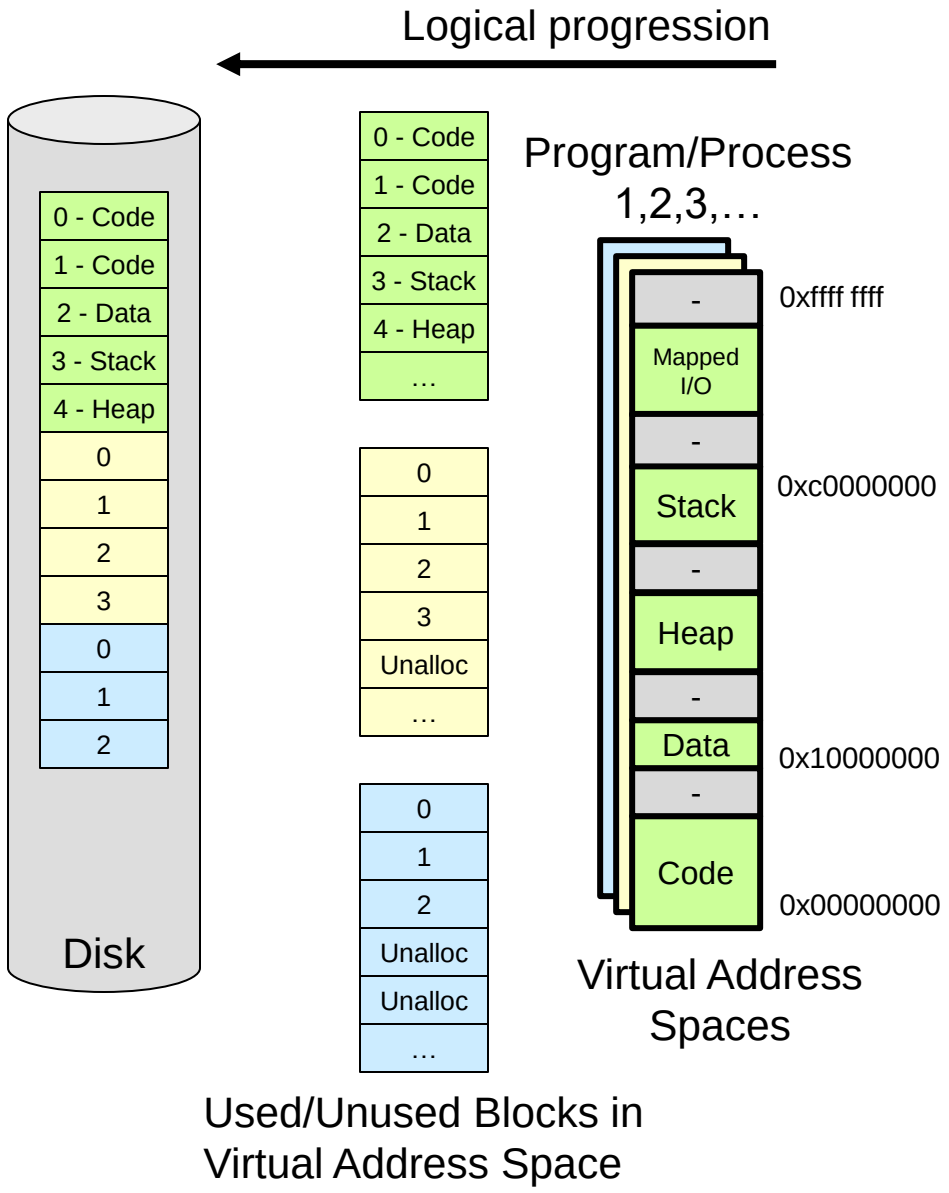
- **Process**
  - **(def 1.) Address Space + Threads**
    - (Virtual) Address Space = Protected view of memory
    - 1 or more threads
  - **(def 2.) : Running instance of a program that has limited rights**
    - Memory is protected: Address translation (VM) ensures no access to any other processes' memory
    - I/O is protected: Processes execute in user-mode (not kernel mode) which generally means direct I/O access is disallowed instead requiring **system calls** into the kernel
- OS Kernel is not considered a "process"
  - Has access to all resources and much of its code is invoked under the execution of a user process thread



Address Spaces

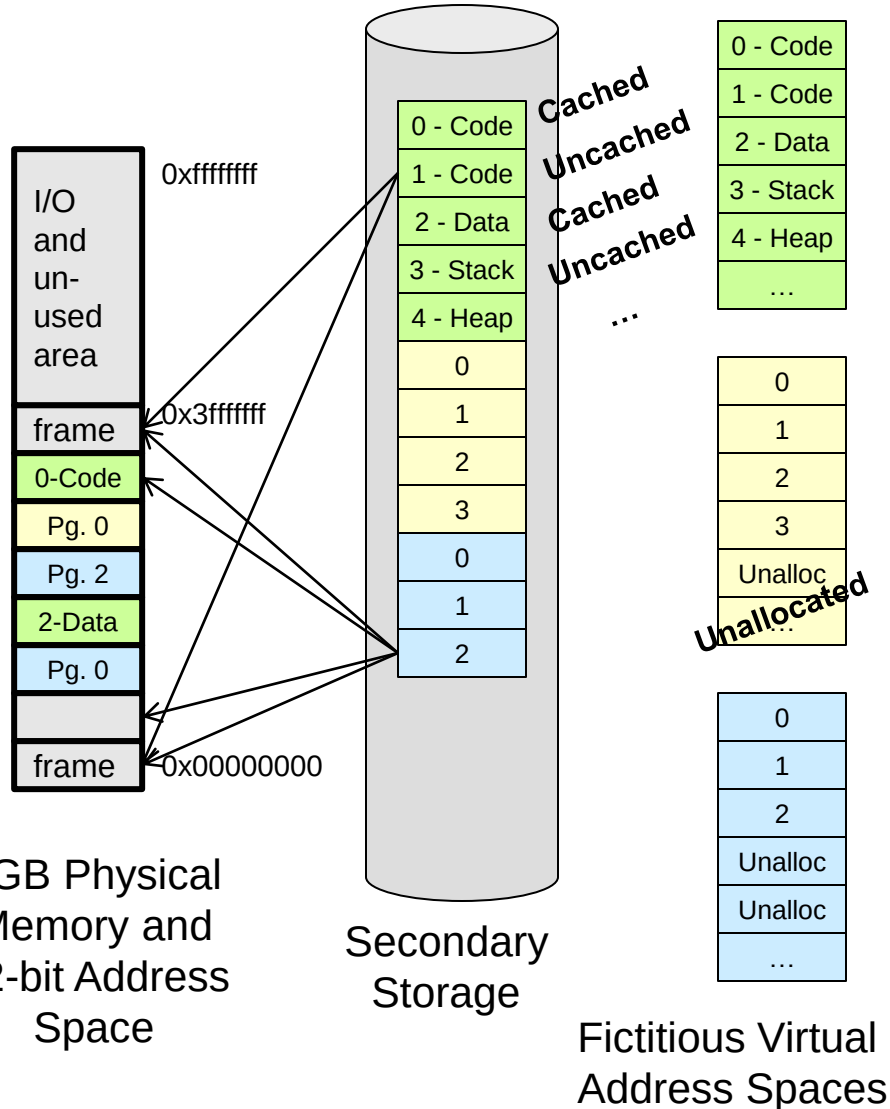
Virtual Address Spaces (VAS)

- Virtual address spaces (VASs) are broken into blocks called "pages"
- Depending on the program, much of the virtual address space will be unused
- Pages can be allocated "on demand" (i.e. when the stack grows, etc.)
- All allocated pages can be stored in secondary storage (hard drive)



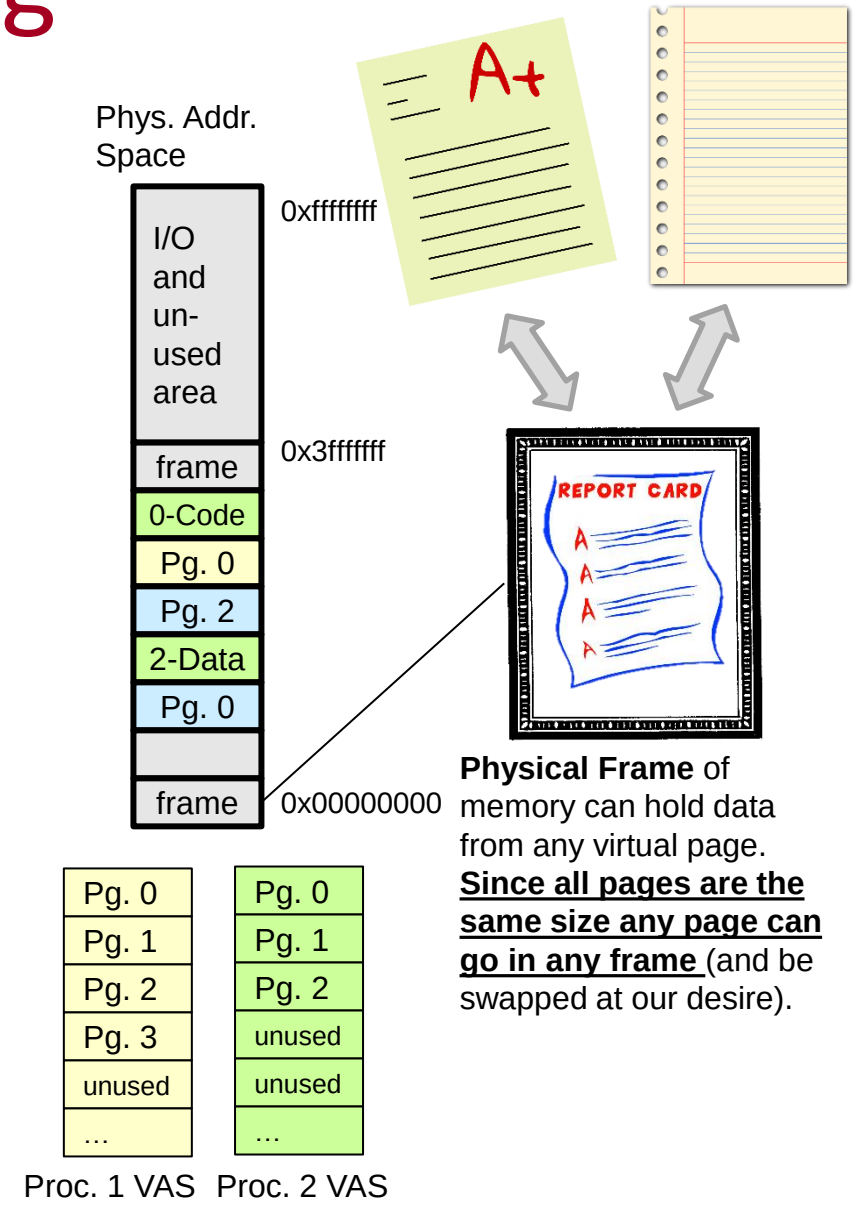
Physical Address Space (PAS)

- Physical memory is broken into page-size blocks called "frames"
- Multiple programs can be running and their pages can share the physical memory
- Physical memory acts as a cache for pages with secondary storage acting as the backing store (next lower level in the hierarchy)
- A page can be:
  - **Unallocated** (not needed yet…stack/heap)
  - Allocated and residing in secondary storage (**Uncached**)
  - Allocated and residing in main memory (**Cached**)



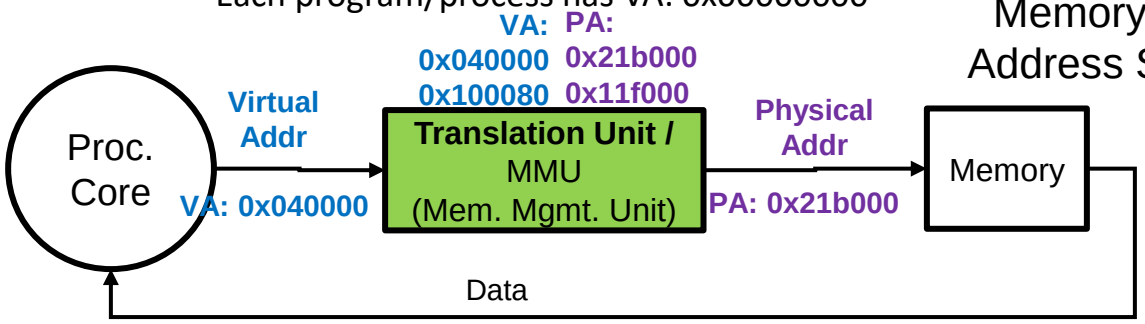
Paging

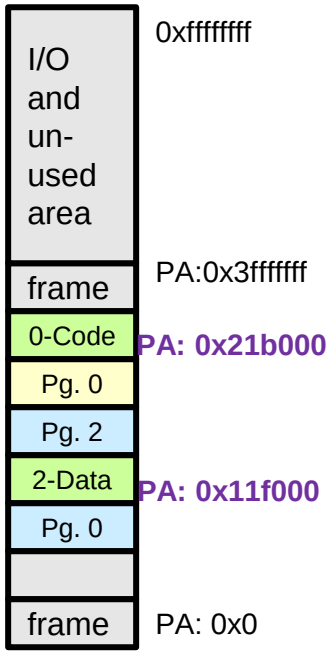
- Virtual address space is divided into equal size "pages" (often around 4KB)
- Physical memory is broken into page frames (which can hold any page of virtual memory and then be swapped for another page)
- Virtual address spaces can be contiguous while physical layout is not



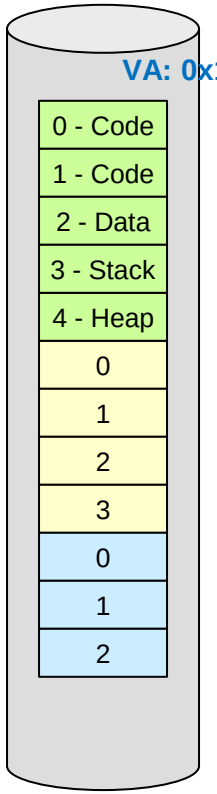
Virtual vs. Physical Addresses

- **Key**: Programs are written using **virtual addresses**
- HW & the OS will **translate** the virtual addresses used by the program to the **physical address** where that page resides
- If an attempt is made to access a page that is not in physical memory, HW generates a "page fault exception" and the OS is invoked to bring in the page to physical memory (possibly evicting another page)
- Notice: Virtual addresses are not unique
  - Each program/process has VA: 0x00000000

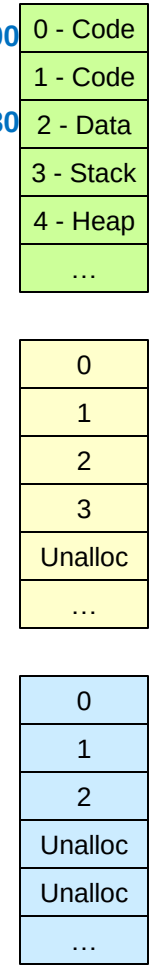




Physical Memory and Address Space



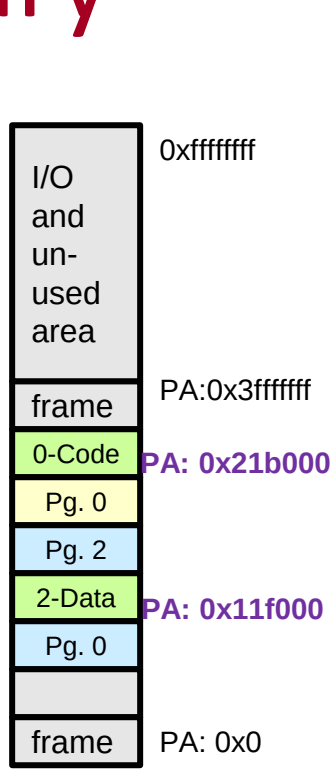
Secondary Storage



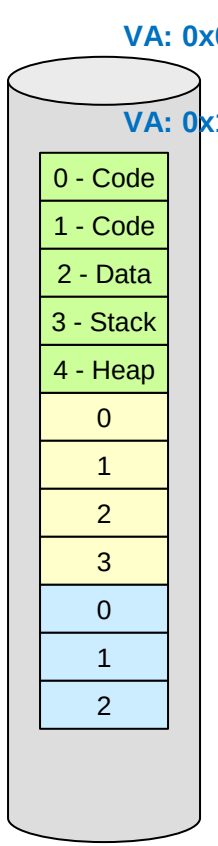
Fictitious Virtual Address Spaces

Summary

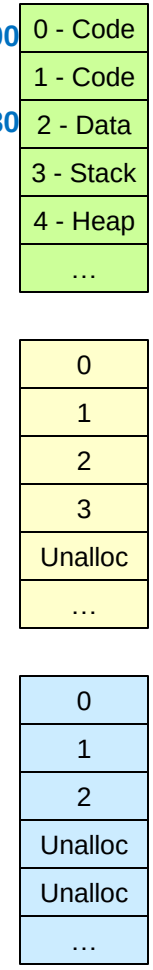
- Program takes an abstract (virtual) view of memory and uses virtual addresses and necessary data is broken into large chunks called pages
- HW and OS work together to bring pages into main memory acting as a cache and allowing sharing
- HW and OS work together to perform translation between:
  - **Virtual address**: Address used by the process (programmer)
  - **Physical address**: Physical memory location of the desired data
- Translation allows protection against other programs



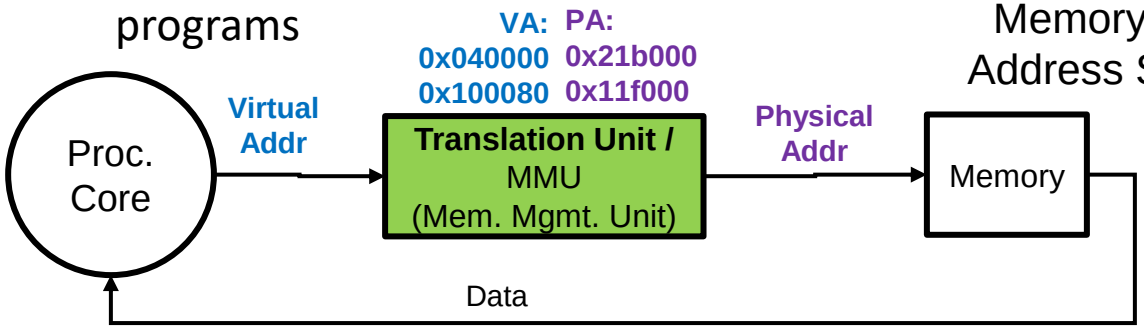
Physical Memory and Address Space



Secondary Storage



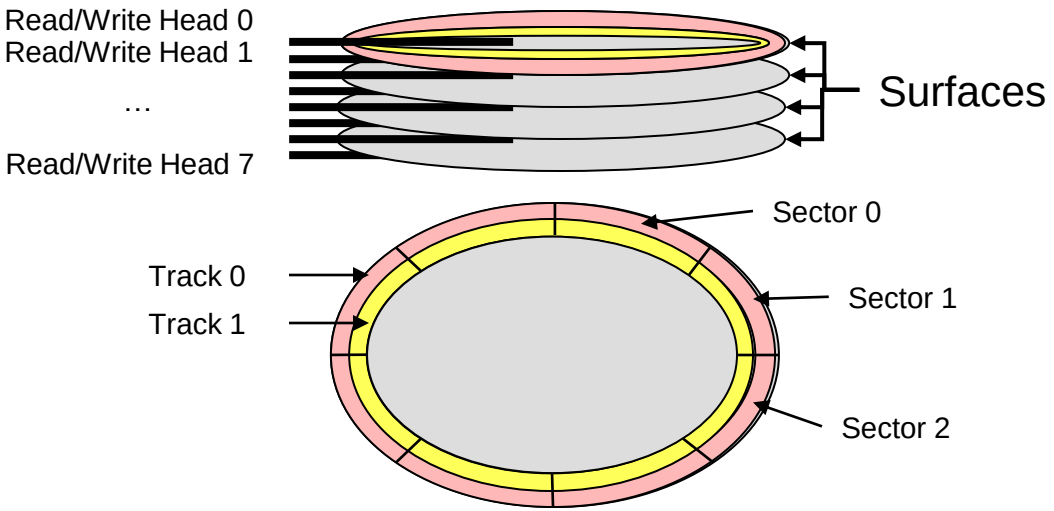
Fictitious Virtual Address Spaces



Virtual Memory Motivation

- Virtual memory is largely discussed in operating systems courses
  - We will focus on HW support for VM
- Disks are SLOW!!
  - Magnetic hard drive consists of double sided surfaces/platters (with R/W head) where each platter is divided into concentric tracks of small sectors that each store several thousand bits (1 – 10 millisecond access time)
  - Even solid state (FLASH-based) drives are slow (10-100 microseconds)

• **Seek Time: Time needed to position the read-head above the proper track**	**3-12 ms**
• **Rotational delay: Time needed to bring the right sector under the read-head**	**5-6 ms**
◦ **Depends on rotation speed (e.g. 5400 RPM)**	
• **Transfer Time:**	**0.1 ms**
• **Disk Controller Overhead:**	**+ 2.0 ms**
	~20 ms



VM Design Implications

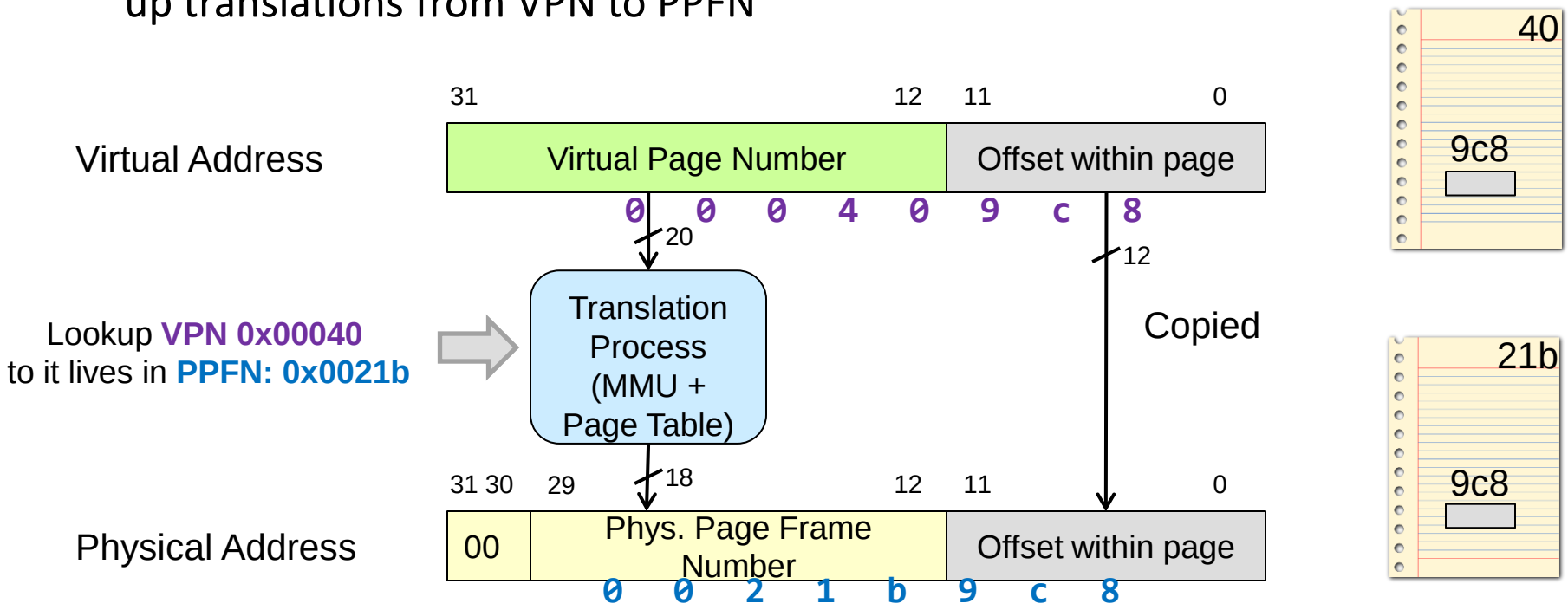
- SLOW secondary storage access on page faults (10 ms)
  - Implies page size should be fairly large (i.e. once we've taken the time to find data on disk, make it worthwhile by accessing a reasonably large amount of data)
  - Implies the placement of pages in main memory should be fully associative to reduce conflicts and maximize page hit rates
  - Implies a "page fault" is going to take so much time to even access the data that we can handle them in software (via an exception) rather than using HW like typical cache misses
  - Implies eviction algorithms like LRU can be used since reducing page miss rates will pay off greatly
  - Implies write-back (write-through would be too expensive)

Page Tables

ADDRESS TRANSLATION

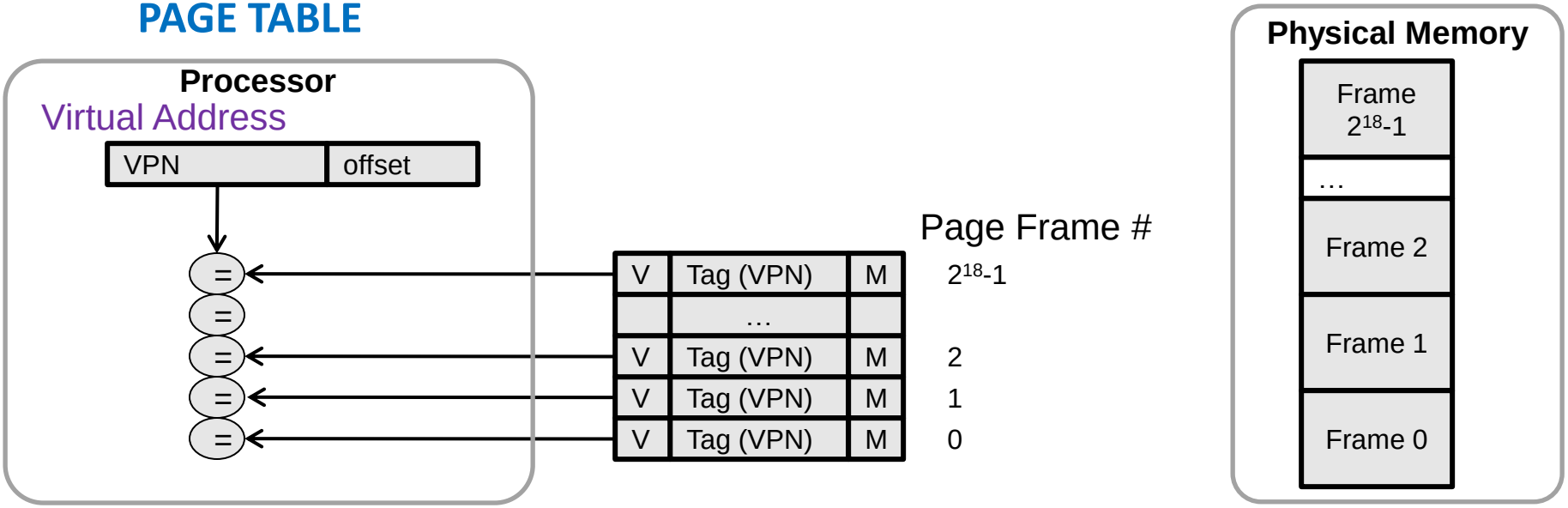
Page Size and Address Translation

- Since pages are usually retrieved from disk, we size them to be fairly large (several KB) to amortize the large access time
- Virtual page number to physical page frame translation performed by HW unit = MMU (Mem. Management Unit)
- Page table is an **in-memory** data structure that the HW MMU will use to look up translations from VPN to PPFN



Address Translation Issues

- We want to take advantage of all the physical memory so page placement should be fully associative
  - For 1GB of physical memory, a 4KB page can be anywhere in the 256K = 2^{18} page frames
- We could potentially track the contents of physical memory using similar techniques to cache
  - TAG = VPN that is currently stored in the frame
  - TAG = 20 + 1 = 21 bits, 2^{18} comparators…TOO MUCH
- Instead, most systems implement full associativity using a **look-up table** = **PAGE TABLE**



Analogy for Page Tables

- Suppose we want to build a caller-ID mechanism for your contacts on your cell phone
  - Let us assume 1000 contacts represented by a 3-digit integer (0-999) in the cell phone (this ID can be used to look up their names)
  - We want to use a simple array (or Look-Up Table (LUT)) to translate phone numbers to contact ID's, how shall we organize/index our LUT

1. LUT indexed w/ contact ID

000	213-745-9823
001	626-454-9985
002	818-329-1980
	…
999	323-823-7104

O(n) - Doesn't Work
We are given phone # and need to translate to ID
(1000 accesses)

2. Sorted LUT indexed w/ used phone #'s

213-730-2198	436
213-745-9823	000
323-823-7104	999
…	…
818-329-1980	002

O(log n) - Could Work
Since its in sorted order we could use a binary search
($\log_2$1000 accesses)

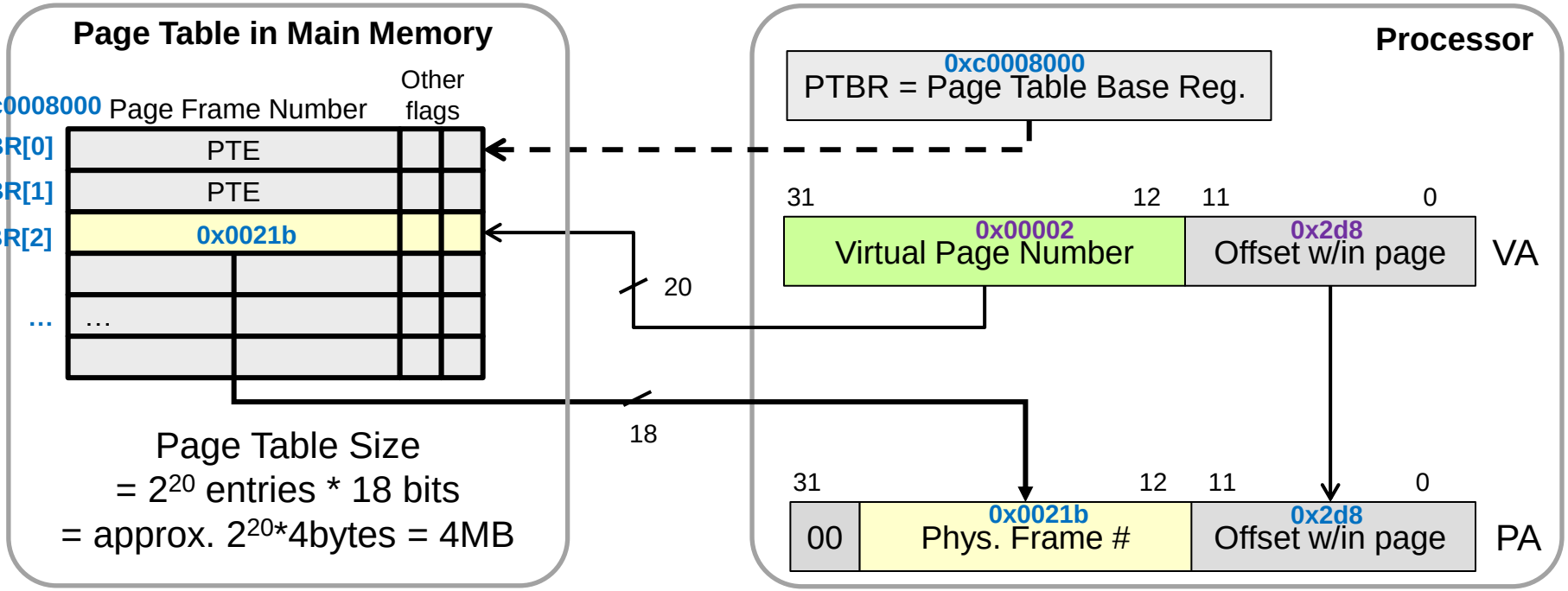
3. LUT indexed w/ all possible phone #'s

000-000-0000	null
	..
213-745-9823	000
	…
999-999-9999	null

O(1) - Could Work
Easy to index & find but LARGE
(1 access)

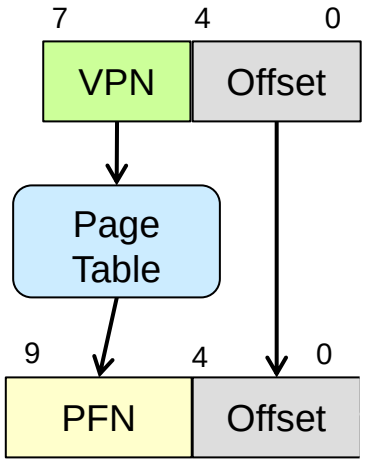
Page Tables

- VA is broken into:
  - VPN (upper bits) + Page offset: Based on page size (i.e. 0 to 4K-1 for 4KB page)
- MMU uses VPN & PTBR to access the page table in memory and lookup physical frame (i.e. like an array access where VPN is the index: PTBR[VPN])
  - Each entry is referred to as a Page Table Entry (PTE) and holds the physical frame number & bookkeeping info
- Physical frame is combined with offset to form physical address
- For 20-bit VPN, how big is the page table? (See below)



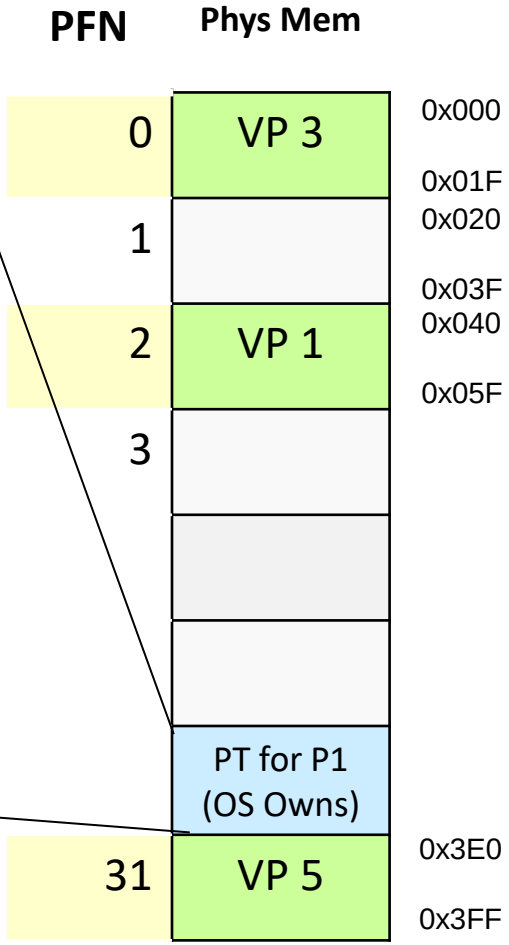
Page Table Example

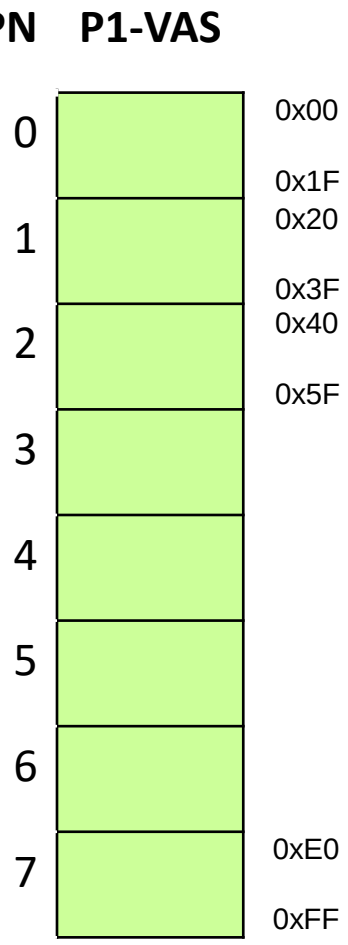
- Suppose a system with 8-bit VAs, 10-bit PAs, and 32-byte pages.



	V	Entry
0	0	0x1a
1	1	0x02
2	1	0x18
3	1	0x00
4	0	0x10
5	1	0x1F
6	0	0x15
7	0	0x0A

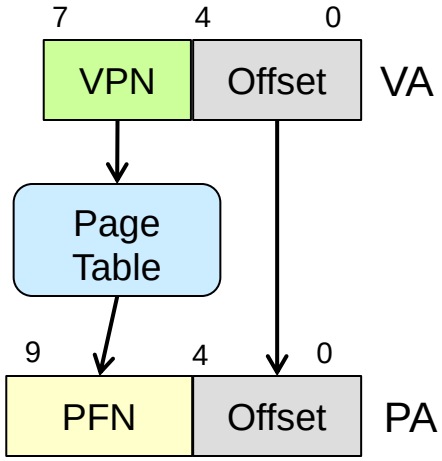
Page Table





Page Table Exercise

- Suppose a system with 8-bit VAs, 10-bit PAs, and 32-byte pages.
- Fill in the table below showing the corresponding physical or virtual address based on the other. If no translation can be made, indicate "INVALID"



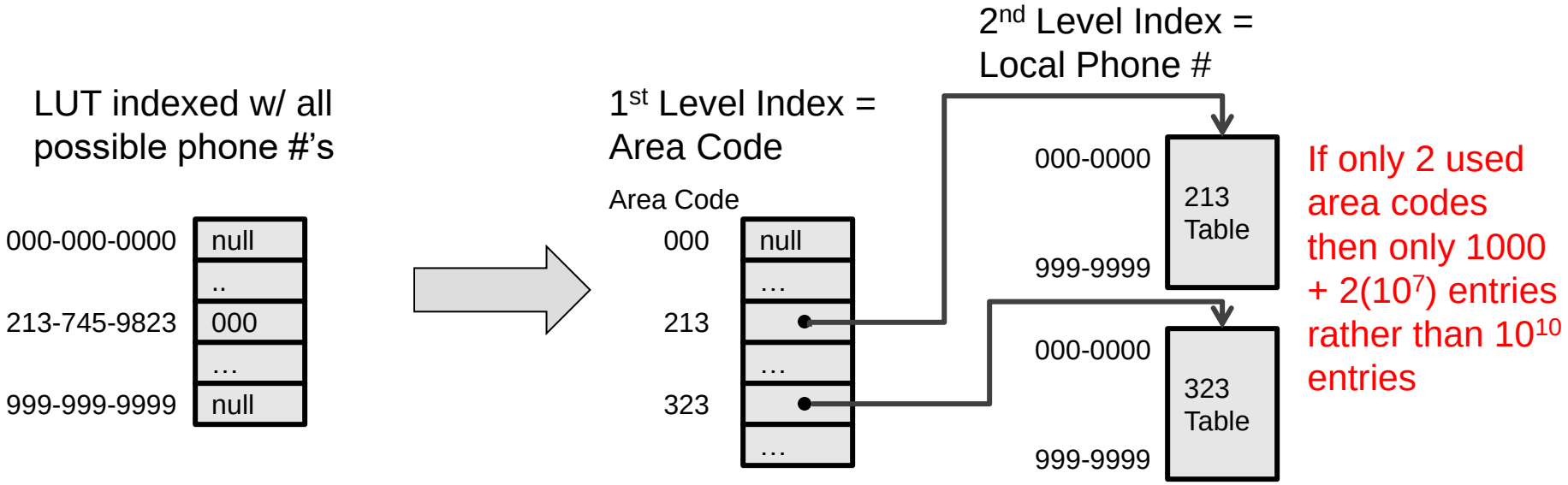
VA	PA
0x2D = 0010 1101	0x3CD
0x7A = 0111 1010	0x0DA
0xEF = 1110 1111	INVALID
0xA8 = 1010 1000	0x3E8

	V	Entry
0	0	0x0E
1	1	0x1E
2	1	0x16
3	1	0x06
4	0	0x0B
5	1	0x1F
6	0	0x15
7	0	0x0A

Page Table

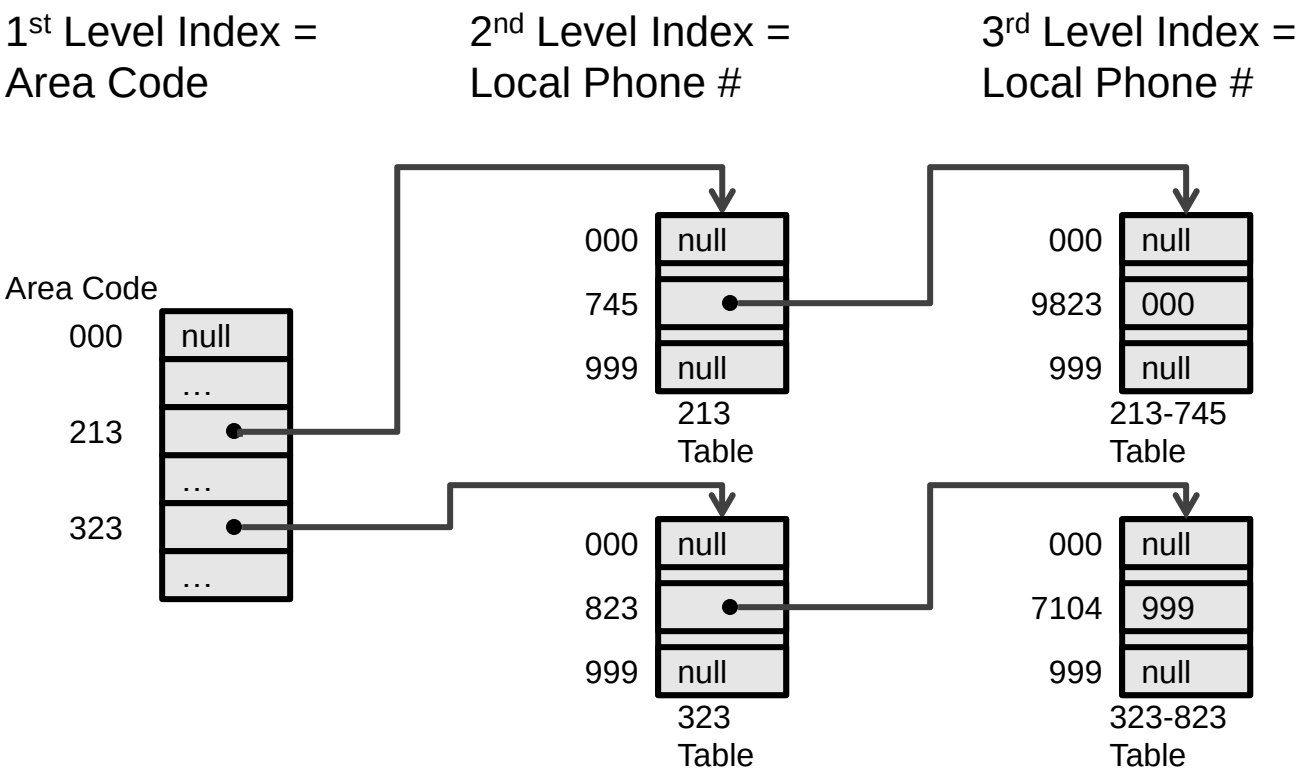
Analogy for Page Tables

- Can we use the table indexed using all possible phone numbers (because it only requires 1 access) but somehow reduce the size especially since much of it is unused?
- Do you have friends from every area code? Likely contacts are clustered in only a few area codes.
- Use a 2-level organization
  - 1st level LUT is indexed on area code and contains pointers to 2nd level tables
  - 2nd level LUT's indexed on local phone numbers and contains contact ID entries



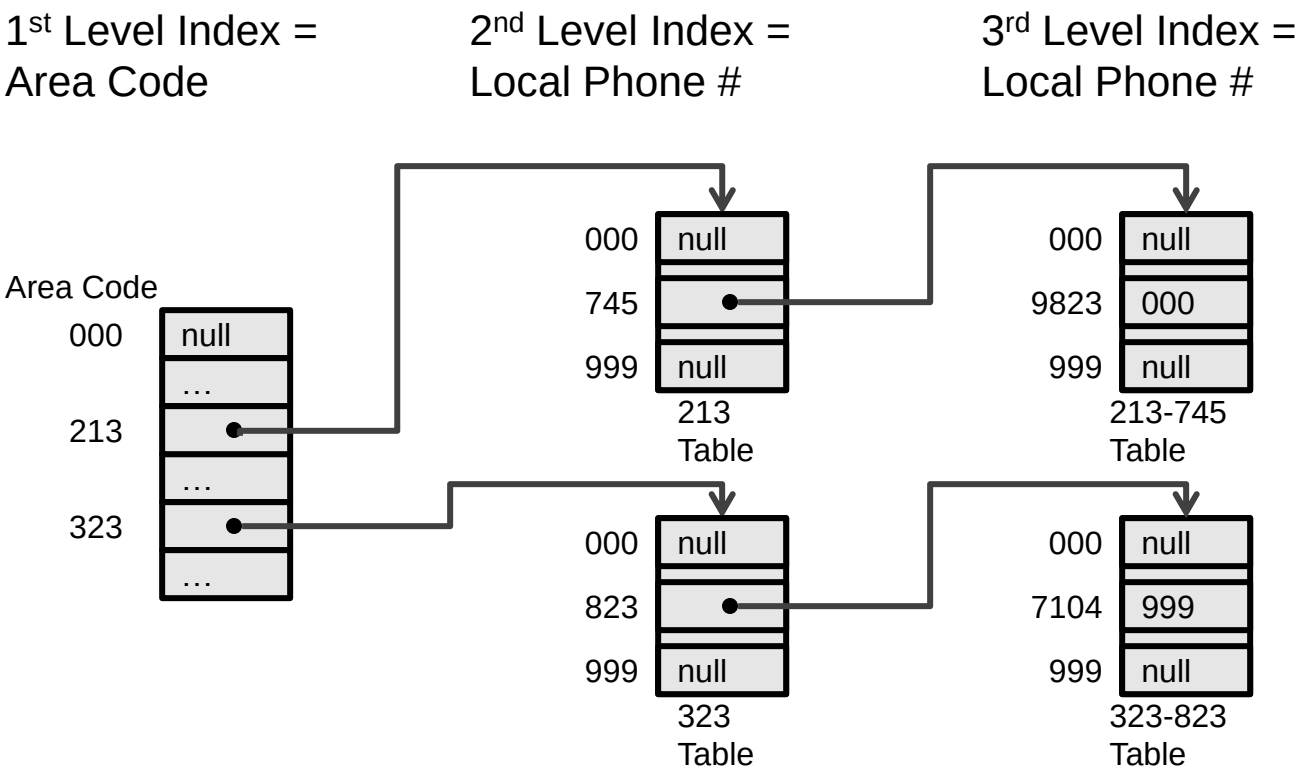
Analogy for Page Tables

- Could extend to 3 levels if desired
  - 1st Level = Area code and pointers to 2nd level tables
  - 2nd Level = First 3-digits of local phone and pointers to 3rd level tables
  - 3rd Level = Contact ID's



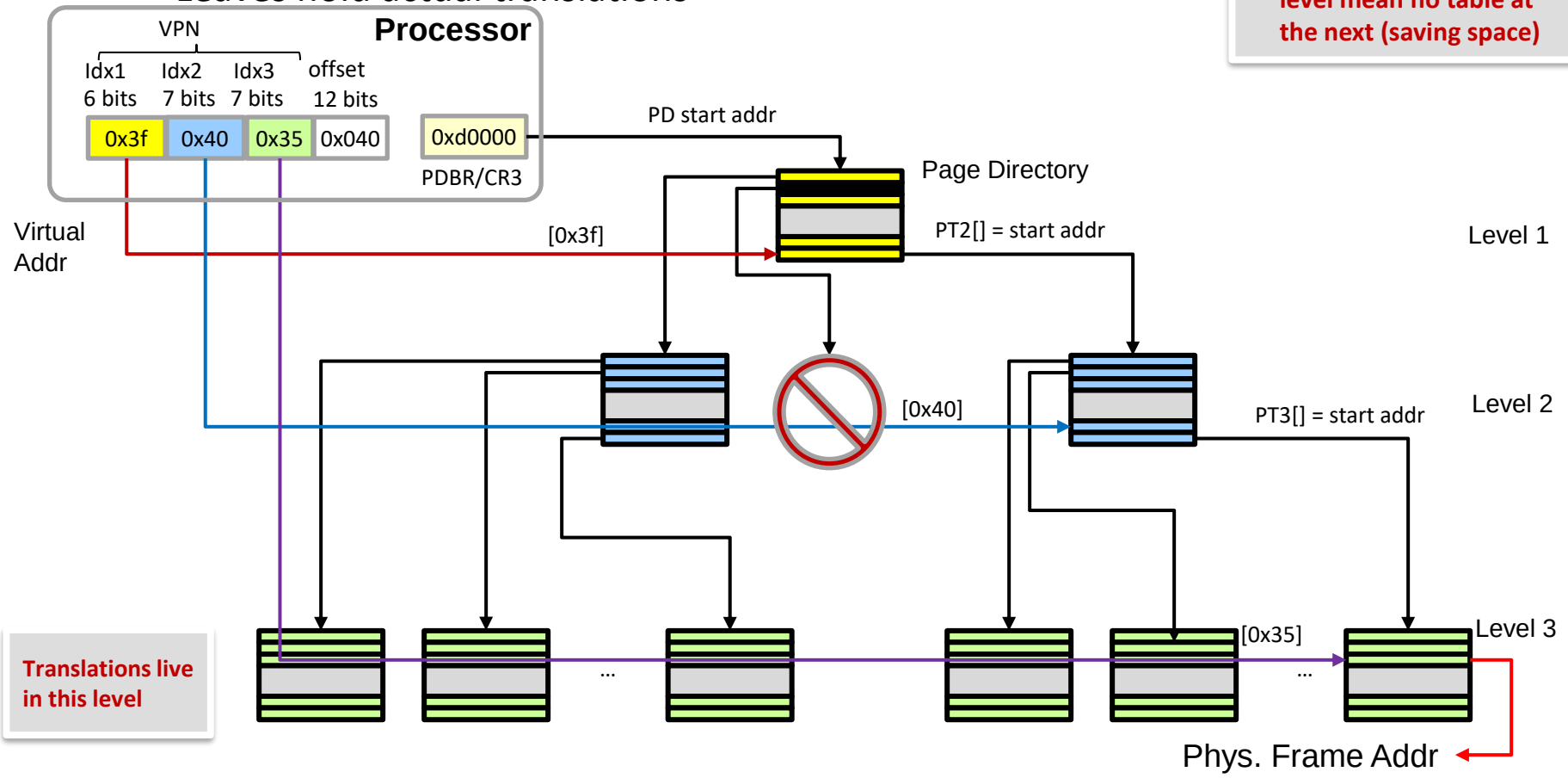
Analogy for Page Tables

- If we add a friend from area code 408 we would have to add a second and third level table for just this entry



Multi-level Page Tables

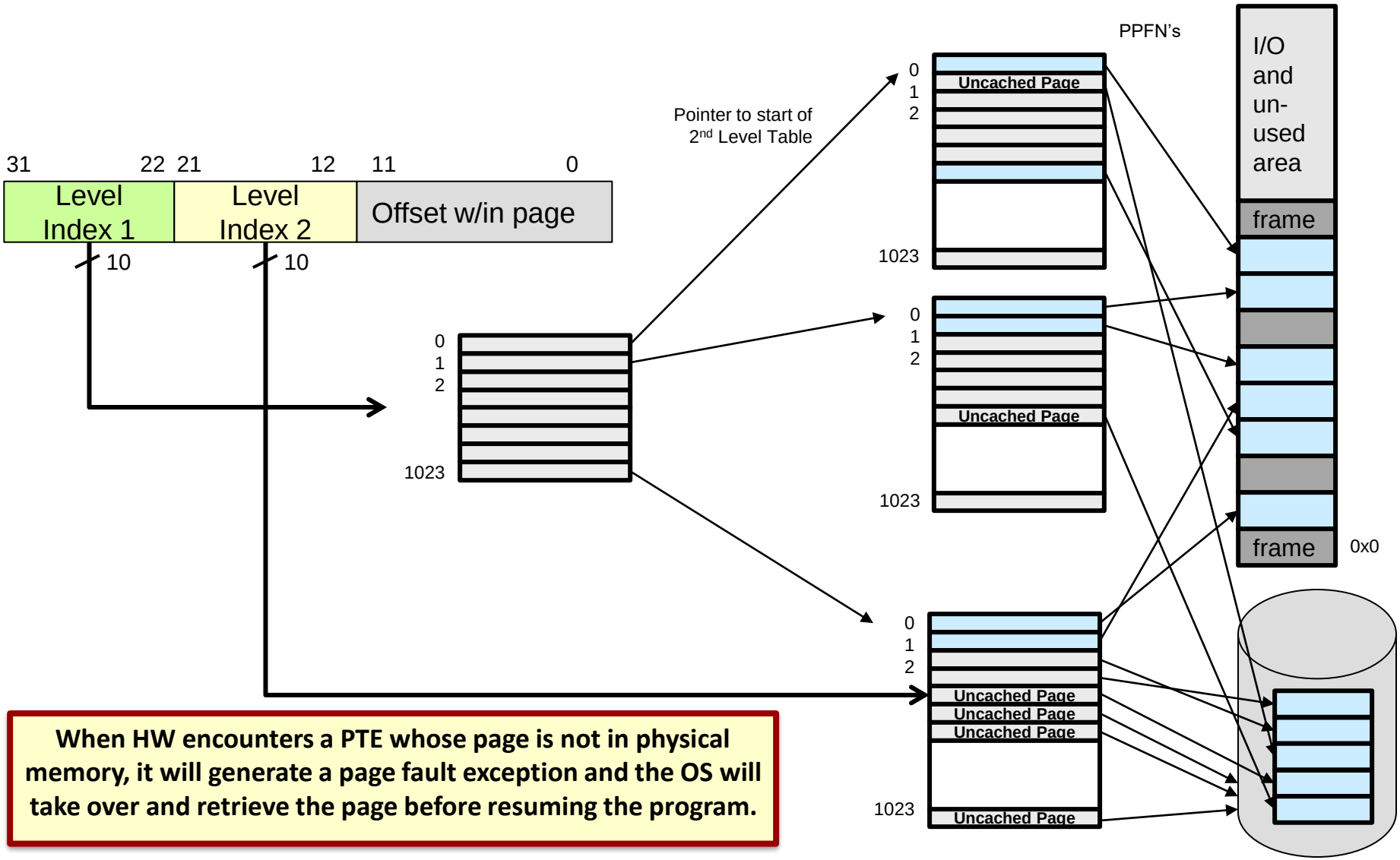
- VPN is broken into fields to index each level of the page table
- Think of a multi-level page table as a tree
  - Internal nodes contain pointers to other page tables
  - Leaves hold actual translations



- **Unused entries in one level mean no table at the next (saving space)**

Translations live in this level

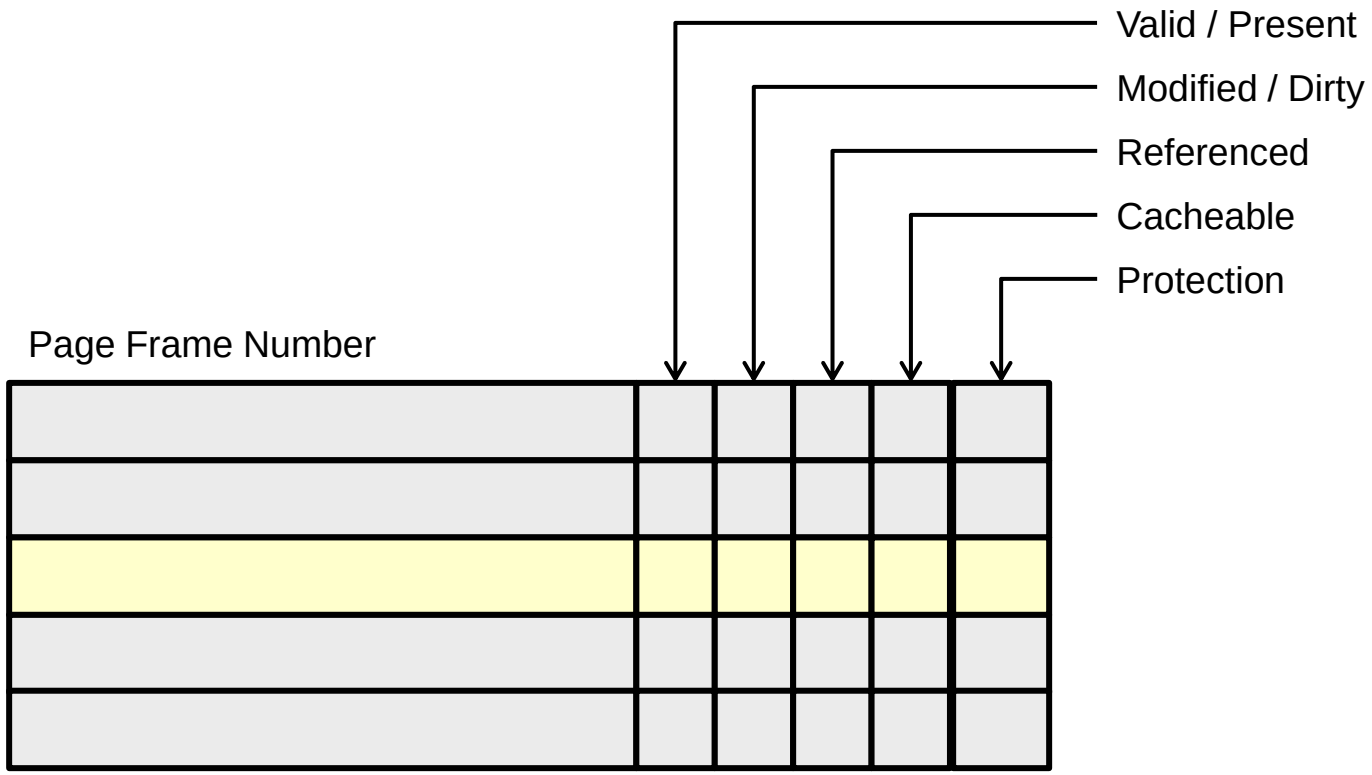
Page Faults



When HW encounters a PTE whose page is not in physical memory, it will generate a page fault exception and the OS will take over and retrieve the page before resuming the program.

Last Level Page Table Entries

- Valid bit (1 = desired page in memory / 0 = page not present / page fault)
- Referenced = To implement pseudo-LRU replacement
- Protection: Read/Write/eXecute



Page Fault Steps

- HW will…
  - Record the offending address, the EPC, and cause (page fault)
- SW will…
  - Pick an empty frame or select a page to evict
  - Writeback the evicted page if it has been modified
    - May block process while waiting and yield processor
  - Bring in the desired page and update the page table
    - May block process while waiting and yield processor
  - Restart the offending instruction

Page Replacement Policies

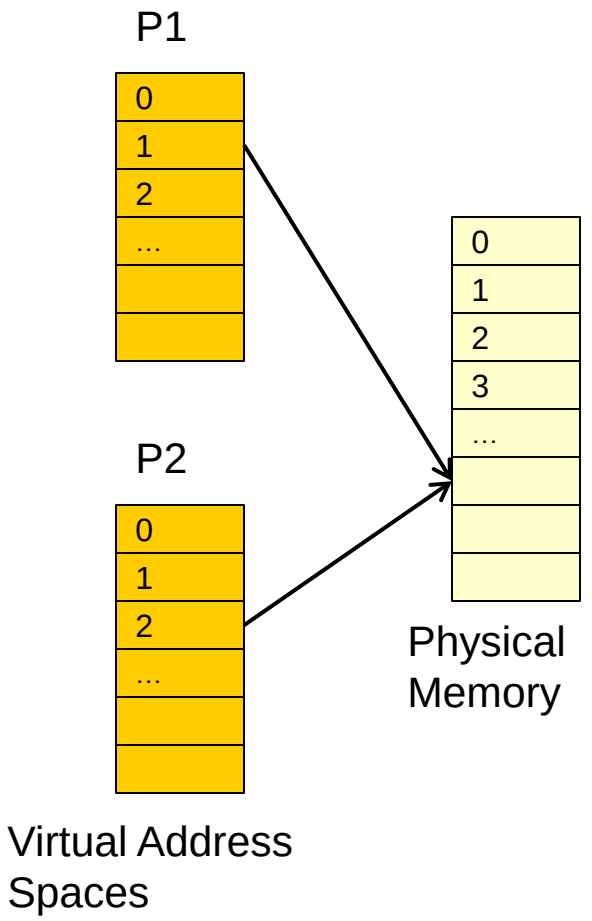
- Possible algorithms: LRU, FIFO, Random
- Since page misses are so costly (slow) we can afford to spend sometime keeping statistics to implement LRU
- Implementing exact LRU would require updating statistics every access (using some kind of timestamp). This is too much to do in HW and we don't want to use SW when we have hits
- HW will implement simple mechanism that allows SW to implement a pseudo-LRU algorithm
  - HW will set the "Referenced" bit when a page is used
  - At certain intervals, SW will use these reference bits to keep statistics on which pages have been used in that interval and then clear the reference bits
  - On replacement, these statistics can be used to find the pseudo-LRU page

Cache & VM Comparison

	Cache	Virtual Memory
Block Size	16-64B	4 KB – 64 MB
Mapping Schemes	Direct or Set Associative	Fully Associative
Miss handling and replacement	HW	SW
Replacement Policy	Full LRU if low associativity / Random is also used	Pseudo-LRU can be implemented

Shared Memory

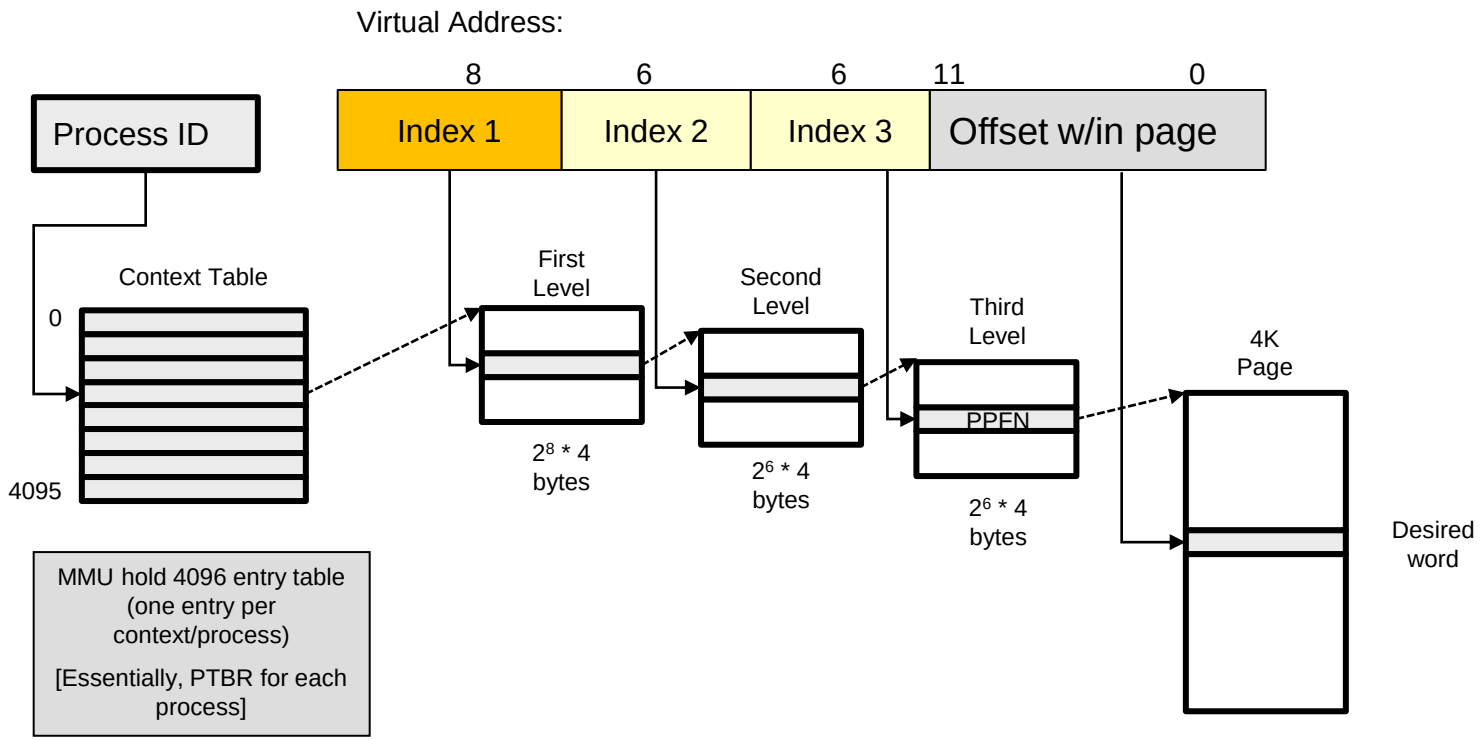
- In current system, all memory is private to each process
- To share memory between two processes, the OS can allocate an entry in each process' page table to point to the same physical page
- Can use different protection bits for each page table entry (e.g. P1 can be R/W while P2 can be read only)



Fast translation

TLBS

SPARC VM Implementation



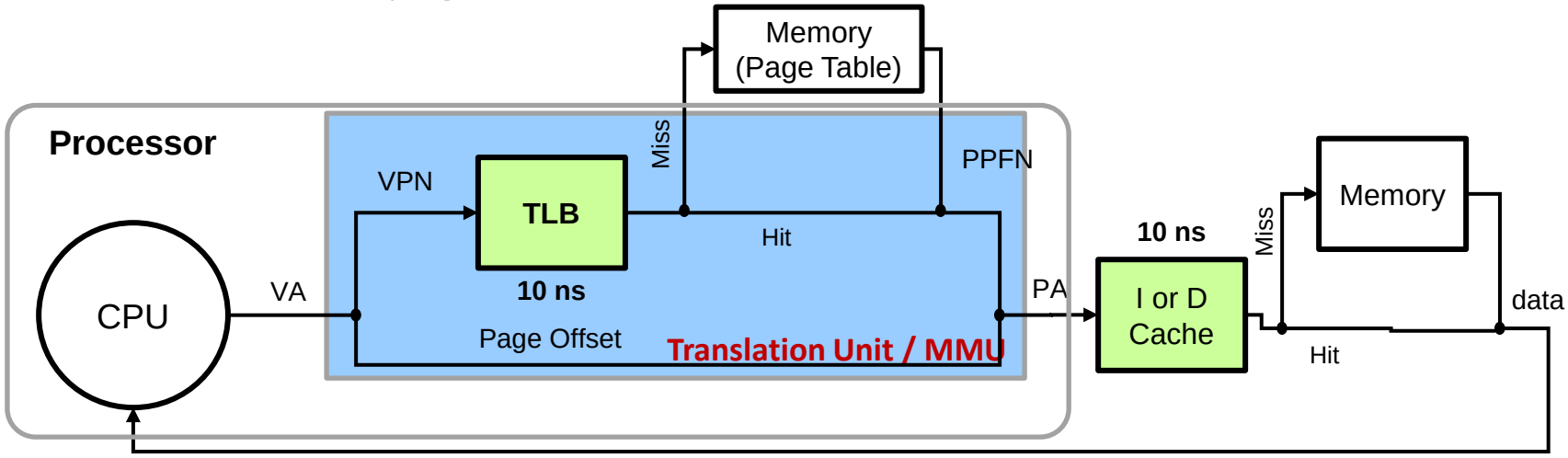
How many accesses to memory does it take to get the desired word that corresponds to the given virtual address? Would that change for a 1- or 2- level table?

Performance Issues

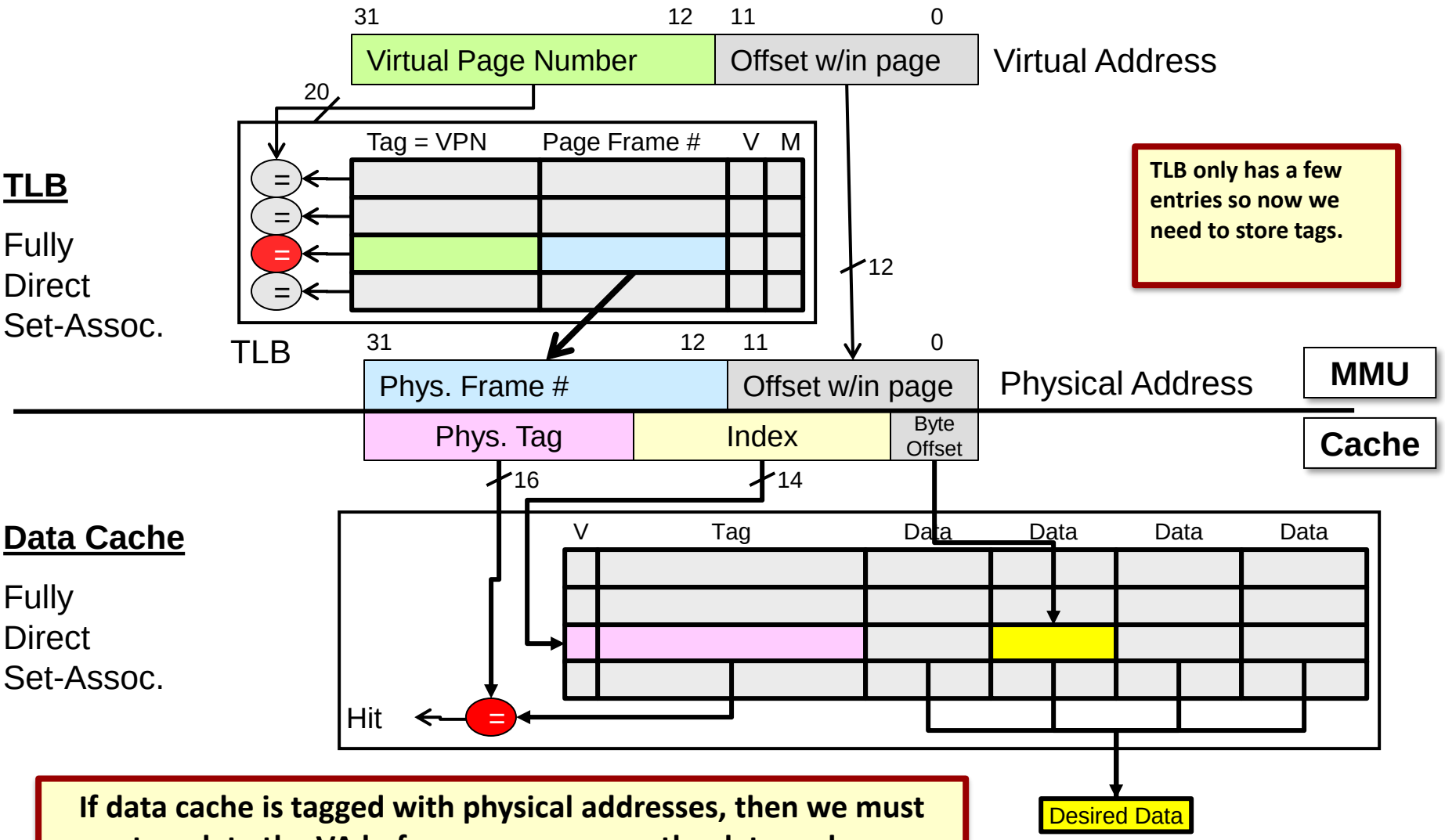
- Let cache hits = 10ns, memory accesses=100ns
- Assume a program makes an access to data located in cache…
  - Without VM, only requires 10ns cache access time
  - With VM, address must first be translated via the page table (recall page table is in memory)
    - If a single-level, one access to the page table (MM) = 100ns
    - If two-levels, two access to the page tables = 200ns
    - If three-levels, three access to the page tables = 300ns
    - Finally, physical address can access cache = 10 ns (if hit)
    - Total time equals 100*L+10 (where L=# of Level of Page Table)
- Translation is extremely costly as currently implemented!!!

Translation Lookaside Buffer (TLB)

- Solution: Let's create a cache for translations = Translation Lookaside Buffer (TLB)
- Needs to be small (64-128 entries) so it can be fast, with high degree of associativity (at least 4-way and many times fully associative) to avoid conflicts
  - On hit, the PPFN is produced and concatenated with the offset
  - On miss, a page table walk is needed



TLB + Data Cache



TLB

Fully
Direct
Set-Assoc.

TLB only has a few entries so now we need to store tags.

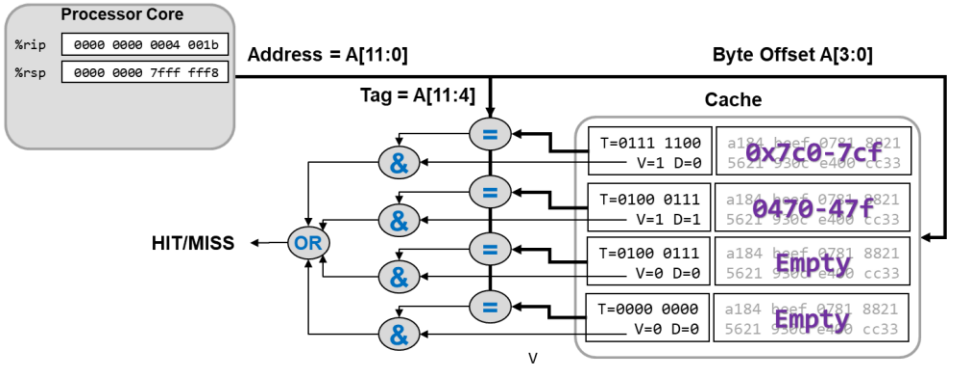
Data Cache

Fully
Direct
Set-Assoc.

If data cache is tagged with physical addresses, then we must translate the VA before we can access the data cache.

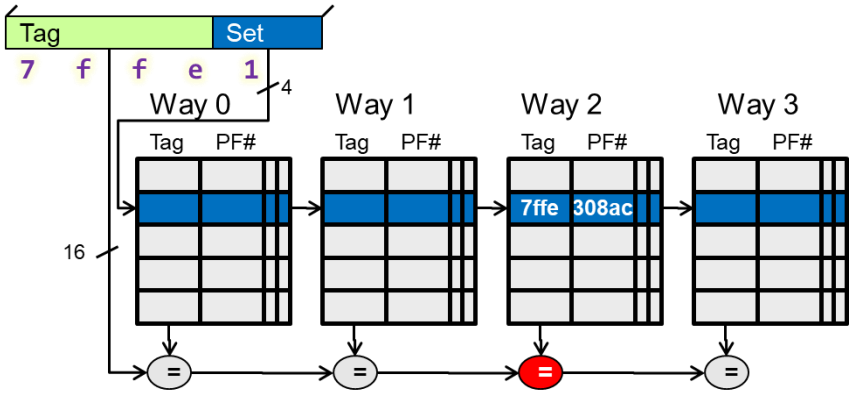
Differences of TLB & Data Cache

- Data cache
  - 1 tag (to identify the block) corresponds to MANY bytes (16-64 bytes)
- TLB
  - 1 tag (VPN) corresponds to 1 translation (PTE/PPFN) which is good for 4KB
- Main Point: TLBs are smaller than normal data caches and faster to access



Instruc./Data Cache

(Offset needed since one tag corresponds to many values)

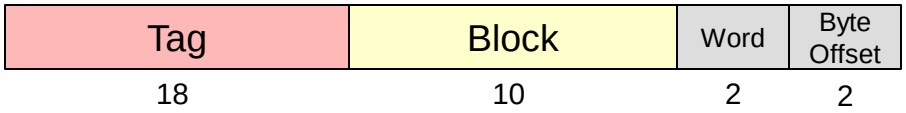


TLB

(1 tag = 1 translation. No Offset needed)

TLB Block Size

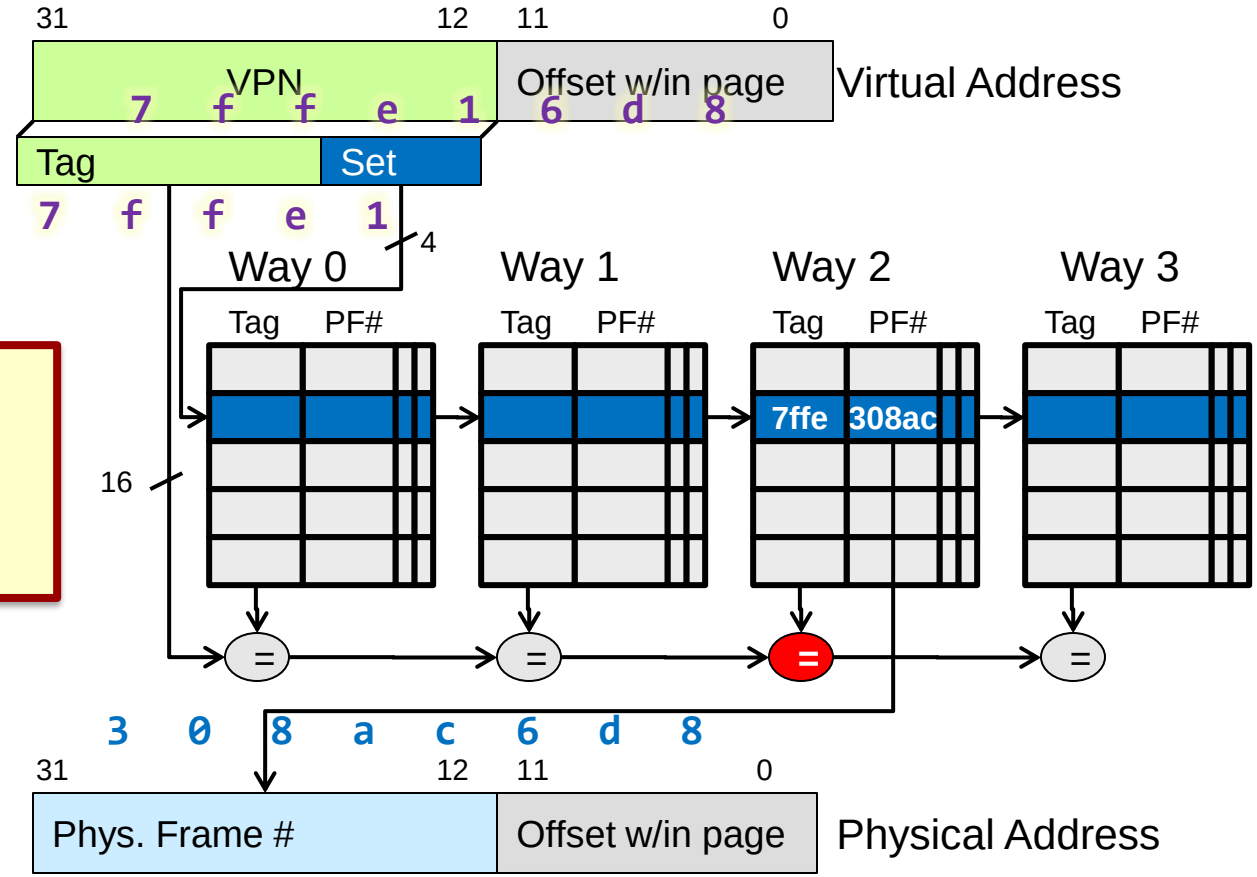
- A block in cache may be
  - 1 word
  - 2 words
  - 4 words
- Consider a direct mapped cache mapping can the word field be 0-bits?



- But an entry in the TLB is 1 value
  - $Log_2(1) = 0$…TLB mappings have no word field

A 4-Way Set Associative TLB

- 64 entry 4-way SA TLB (set field indexes each "way")
  - On hit, page frame # supplied quickly w/o page table access



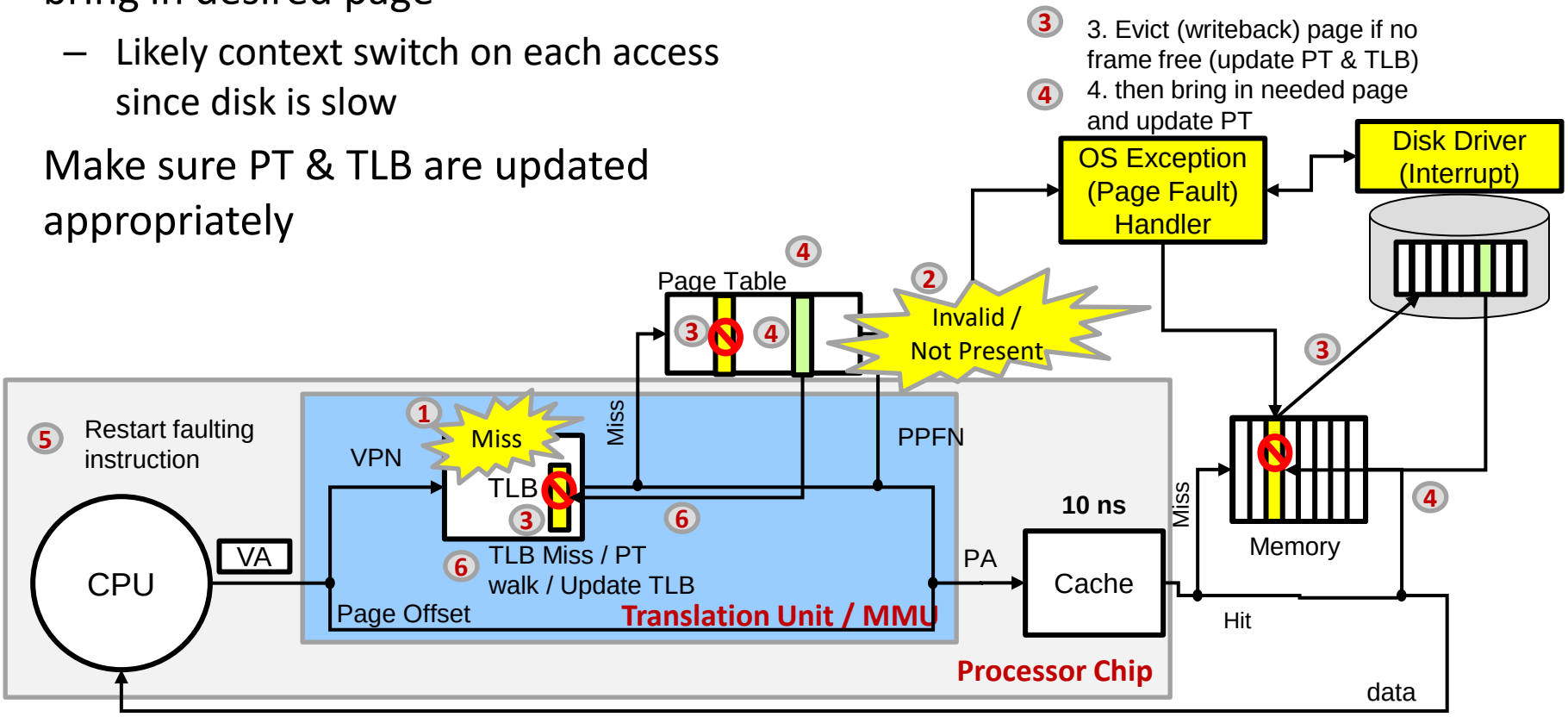
What is the page size? 4KB

Tag size? 16 bits

Comparator Width? 17-bits (16+V)

Page Fault Steps

- On page fault, handler will access disk to evict old page (if dirty) and to bring in desired page
  - Likely context switch on each access since disk is slow
- Make sure PT & TLB are updated appropriately



TLB Miss Process

- On a TLB miss, there is some division of work between the hardware (MMU) and OS
- Option 1
  - MMU can perform the TLB search followed by a page table walk if needed
  - If page fault occurs, OS takes over to bring in the page
- Option 2
  - MMU performs TLB Search
  - If TLB miss, OS can perform page table walk and bring in page if necessary
- When we want to remove a page from MM
  - First flush out blocks belong to that page from cache (writing back if necessary)
  - Invalidate tags of those blocks
  - Invalidate TLB entry (if any) corresponding to that page
    - If D=1, set dirty bit in page table
  - If page is dirty, copy page back to the disk
  - Simple way to remember this…
    - If parents (page) leave a party then the children (cache blocks & TLB entries) must leave too

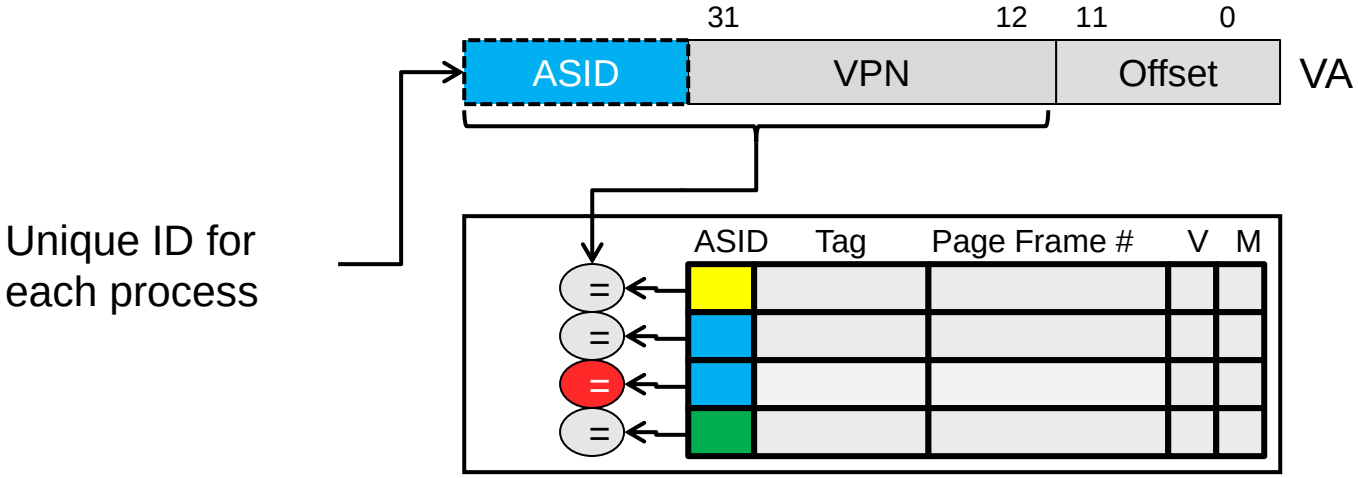
TLB Hit Rates

- Because of high degree of associativity and limited working set of pages (usually) we can get VERY HIGH hit rates for the TLB
  - Variable page size settable by OS to allow for different working set sizes
  - Example: 64 TLB entries and 4 KB pages = 256KB
- Often times, separate TLB's for instruction and data address translation

```
i = 0, sum = 0, dat[MAX];
for(i=0; i < MAX; i++){
  sum += dat[i];
}
```

Multiple Processes and TLB Flushing

- Recall each process has its own virtual address space, page table, and translations
- How does TLB handle context switch
  - Can choose to only hold translations for current process and thus invalidate all entries on context switch
  - Can hold translations for multiple processes concurrently by concatenating a process or address space ID (PID or ASID) to the VPN tag



Property of Inclusion

- Property of Inclusion
  - Cache contents are a (**subset** / superset) of main memory contents
  - Main memory contents are a (**subset** / superset) of page/swap file on disk
  - TLB contents are a (**subset** / superset) of page table contents

Cache, VM, and Main Memory

TLB	VM	Cache	Possible Y/N & Description
Hit	Hit	Hit	Possible: best-case scenario
Hit	Hit	Miss	Possible: TLB hits (hit in VM is implied), then cache miss
Miss	Hit	Hit	TLB misses, then hits in page table, then cache hit
Miss	Hit	Miss	TLB misses, then hits in page table, then cache miss
Miss	Miss	Miss	TLB misses, then page fault, then miss in cache
Hit	Miss	Miss	Impossible: cannot hit in TLB if page not present
Hit	Miss	Hit	Impossible: cannot hit in TLB if page not present
Miss	Miss	Hit	Impossible: data cannot be in cache if page not present

Taken from H & P, “Computer Organization” 3rd, Ed.

Virtual Memory System Examples

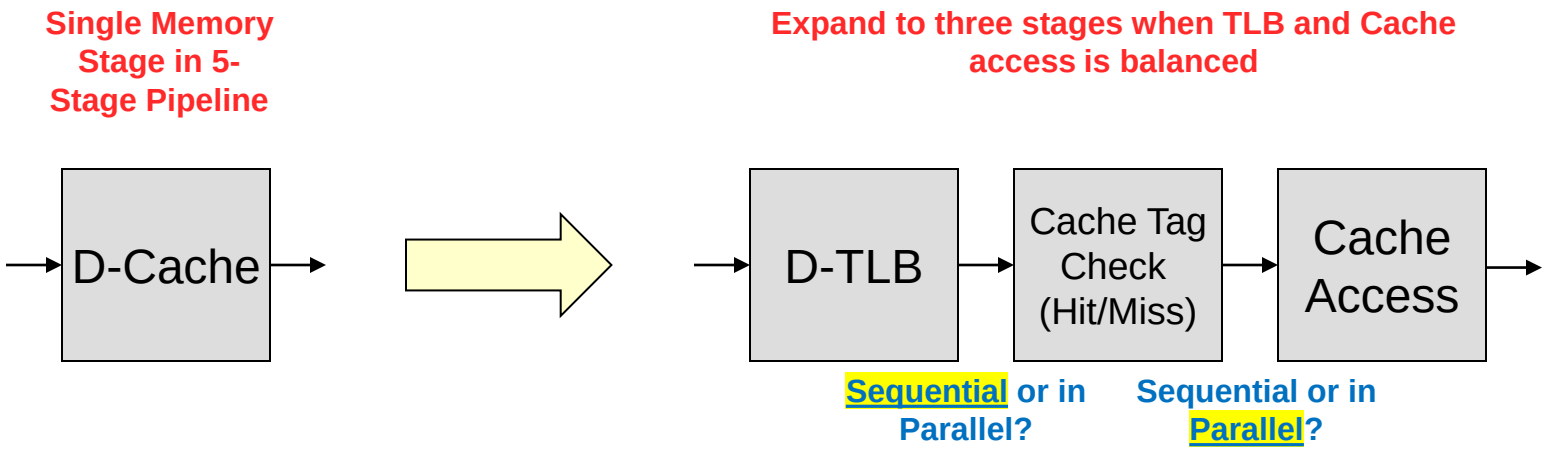
Microprocessor	AMD Opteron	P4	PPC 7447a
Virtual Address	48-bit	32- or 48-bit	52-bit
Physical Address	40-bit	36-bit	32- or 36-bit
TLB Entries (I/D/L2 TLB)	L1: 40/40 L2: 512/512	L1: 128/128	L1: 128 / 128
TLB Mapping	L1: Fully L2: 4-way SA	Fully (? 4-way)	2-way set associative
Min. Page Size	4 KB	4 KB	4 KB

Notes: Large VA's include ASID (process ID's) and other segment information

Sources: H&P, "CO&D", 3rd ed., Freescale.com,

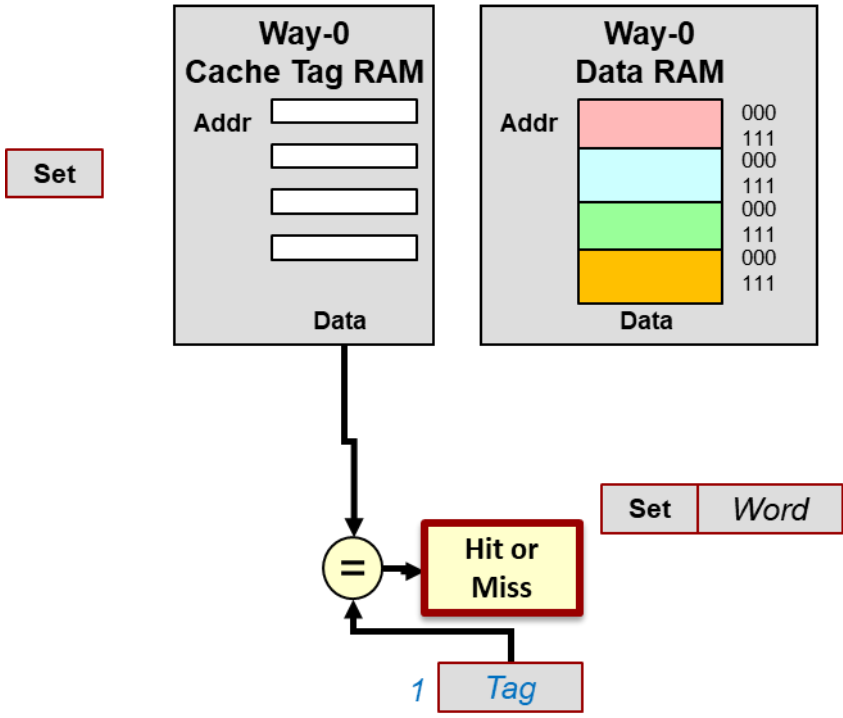
8-Stage Pipeline

- In 5-stage pipeline, we place the I-Cache and D-Cache in one stage each
- When we add VM translation (i.e. TLB), cache tag check, and data access we need more stages to balance the delay



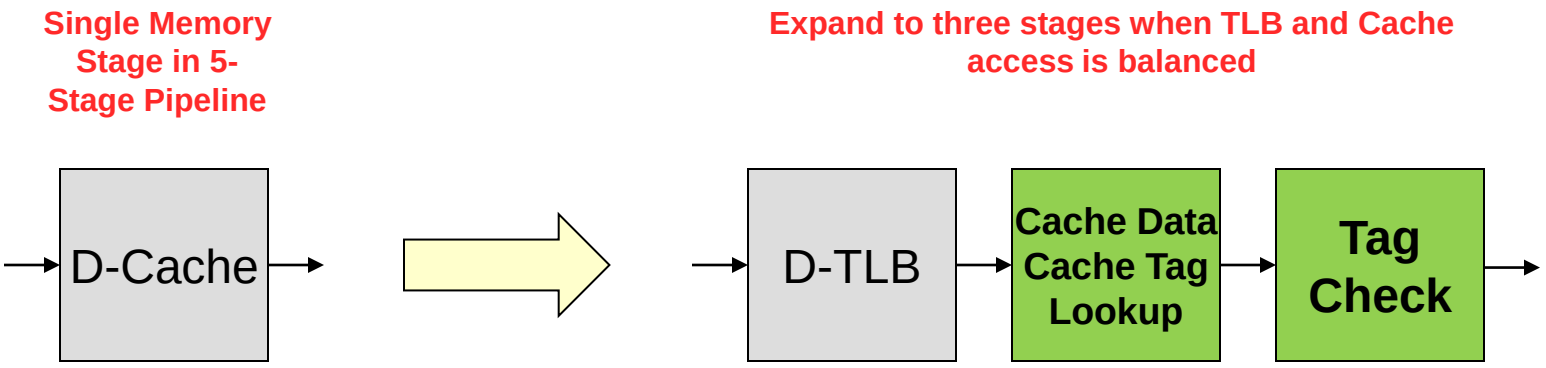
Cache Access Review

- Set or block field indexes LUT holding tags
- 2 steps to determine hit:
  - Index (lookup) to find tags (using block or set bits)
  - Compare tags to determine hit
  - Sequential connection between indexing and tag comparison
- But can we start to access data from DATA RAM speculatively (expecting we will HIT)?
  - YES!



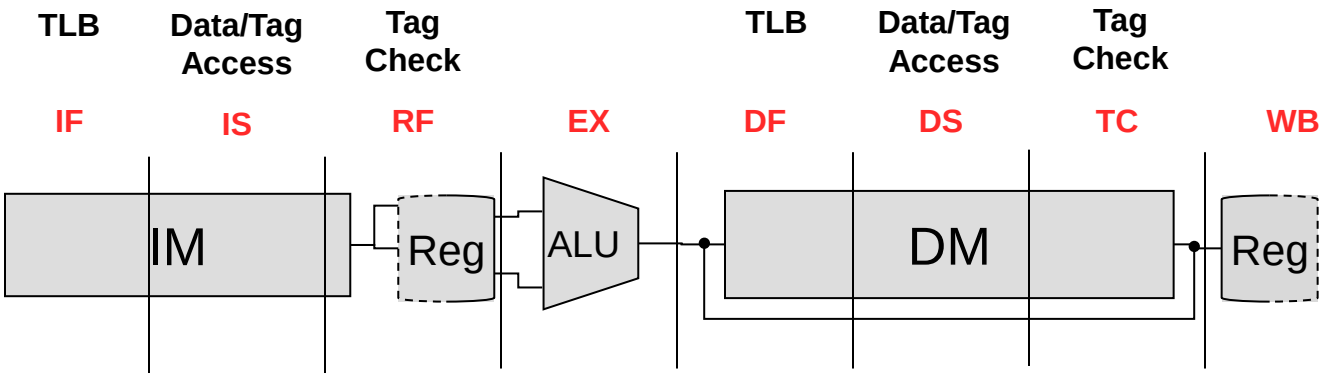
Memory/Cache Stages

- We can overlap Data and Tag access/lookup at the same time and then use the tag check to determine validity before using the data



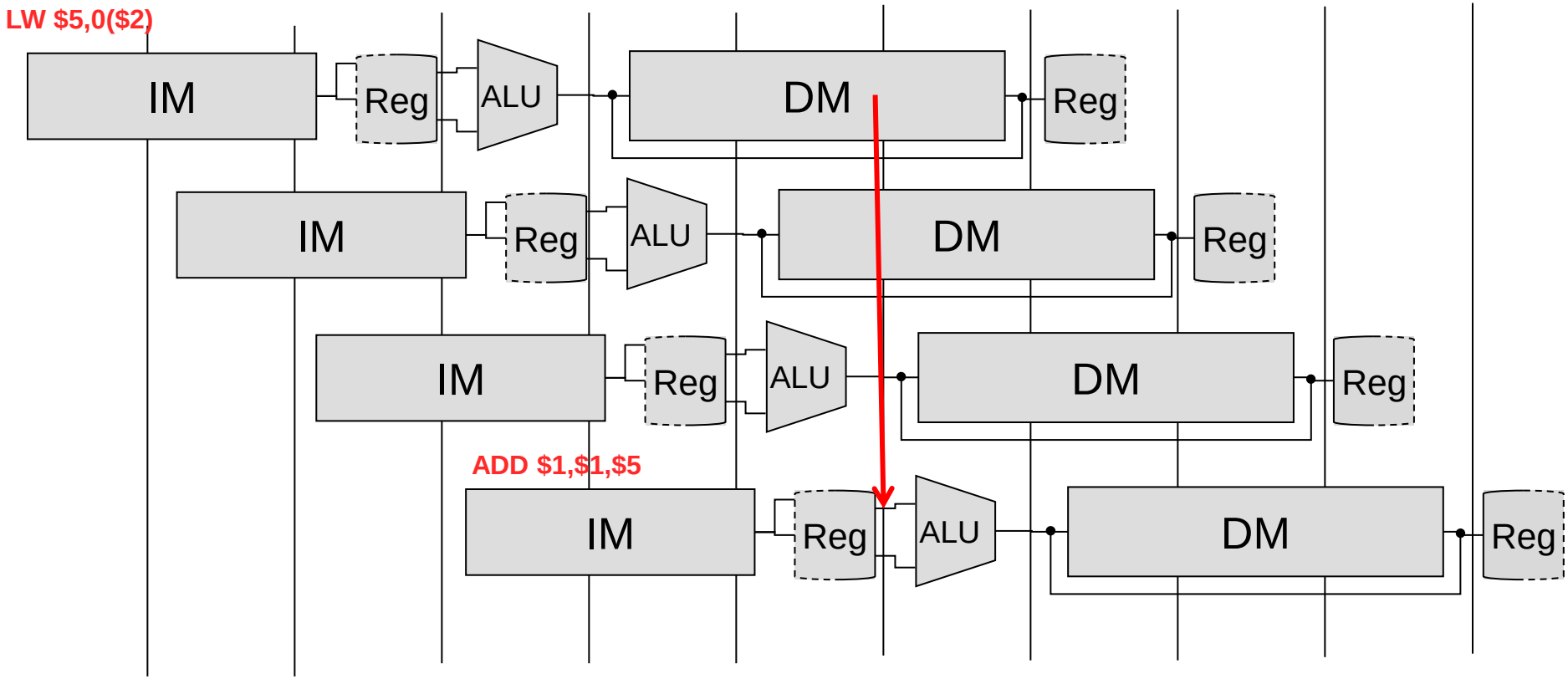
8-Stage Pipeline

- MIPS R4000 uses pipelined instruction and data caches
- The instruction is available at the end of the IS stage but the tag check is done in RF
  - This is harmless since it's fine to start the decode and *read* registers even if it's an invalid tag so long as we know by the end of the clock
- For the data cache we must perform the tag check before we can start writing into the register file



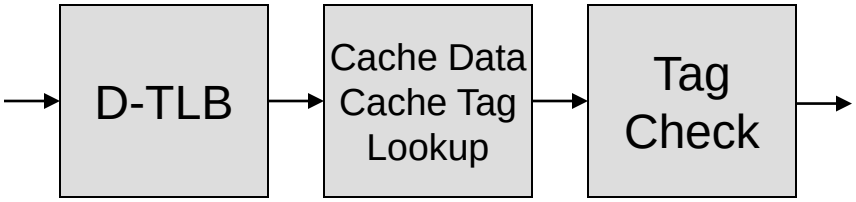
Load Dependency Issue Latency

- 8-stage pipeline has a *two-cycle load delay*
  - Possible since data is available at end of DS and can be forwarded
  - If tag check indicates a miss the pipeline can be "backed up" (re-execute the instruction when the data is available



Cache Addressing with VM

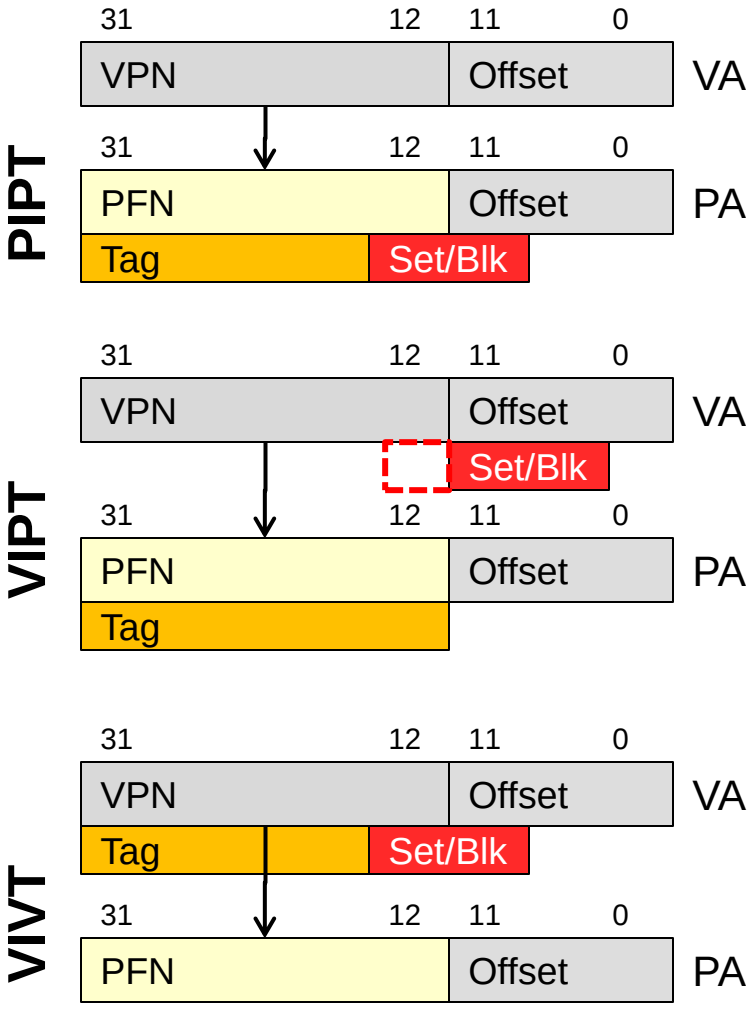
- Rather than waiting for address translation and then performing cache access/hit process, can we overlap the translation and portions of the cache sequence?
  - Yes, if we choose page size, block size, and set/direct mapping carefully



Is sequential operation required between DTLB and Cache access?

Cache Index/Tag Options

- Physically indexed, physically tagged (PIPT)
  - Wait for full address translation
  - Then use physical address for both indexing and tag comparison
- Virtually indexed, physically tagged (VIPT)
  - Use portion of the virtual address for indexing the Tag RAM then wait for address translation and use physical address for tag comparisons
  - Easiest when index portion of virtual address w/in offset (page size) address bits, otherwise aliasing may occur
- Virtually indexed, virtually tagged (VIVT)
  - Use virtual address for both TAG RAM indexing and tagging…No TLB access unless cache miss
  - Since TAGs are NOT UNIQUE across processes, this approach requires invalidation of cache lines on context switch or use of process ID as part of tags

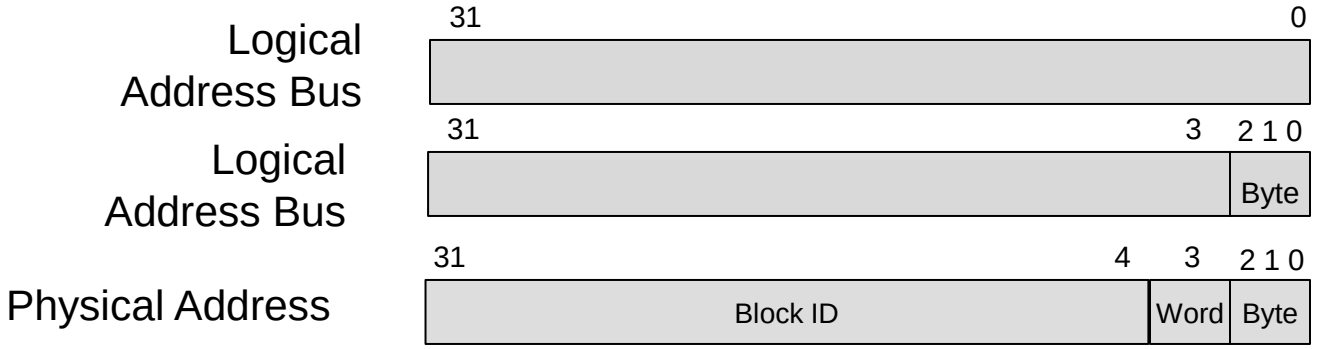


A Complete VM / Cache Example

- Use the following specification for the following questions
  - 64-bit data, 32-bit virtual/physical address
  - Page Size: 128KB
  - TLB Size: 256 entry 4-way set associative
  - Page Table Org.: 3-levels
    - A 64 entry A-Table (page directory) followed by several 32 entry B-Tables (2^{nd} level tables) followed by some number of C-Tables (3^{rd} level)
  - Cache Organization
    - Cache Size: 512KB
    - 8-way set associative
    - Block size: 2 words [Word = 64-bits = 8 bytes]

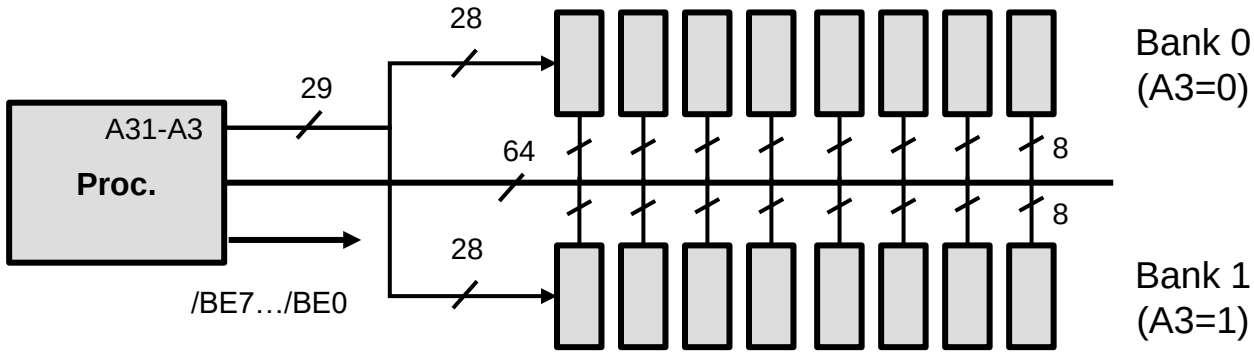
Address Bus and Interleaving

- Use the following specification for the following questions
  - 64-bit data, 32-bit virtual/physical address
  - Cache Organization: Block size: 2 words [1 Word = 64-bits = 8 bytes]
- How many banks would you suggest for interleaving purposes?



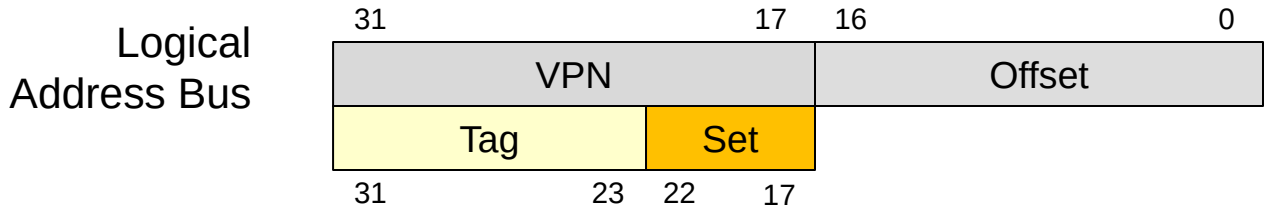
64-bit Data bus = 8 bytes = 8 Byte enables (/BE7…/BE0)

2 Banks so we can quickly get two words to the data cache when a block is transferred



TLB Mapping

- Use the following specification for the following questions
  - 64-bit data, 32-bit virtual/physical address
  - Page Size: 128KB
  - TLB Size: 256 entry 4-way set associative

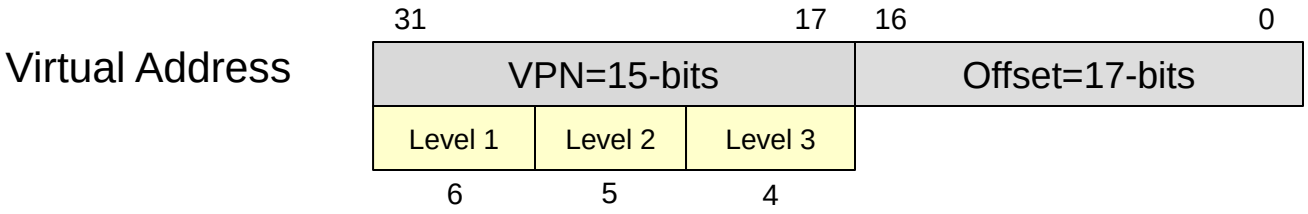


Page Size = 128KB =>
Offset Field Size = 17-bits

# of TLB Sets: 256 entries / 4 entries per set
= 64 sets => 6 set bits

Page Table Mapping

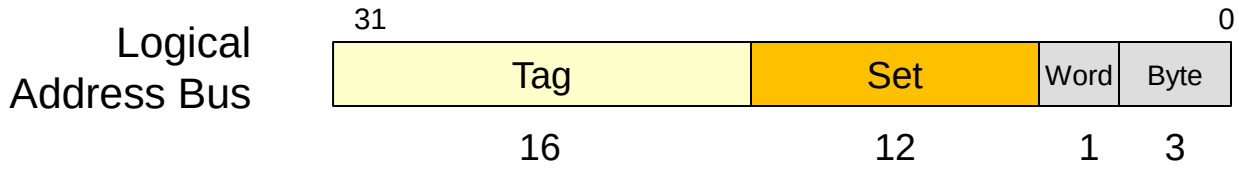
- Use the following specification for the following questions
  - Page Size: 128KB
  - Page Table Org.: 3-levels
    - A 64 entry A-Table (page directory)
      32 entry B-Tables (2nd level tables)
      some number of C-Tables (3rd level)



VPN = 15-bits
Level 1 Page Table = 64 (2^6) entries => 6-bits
Level 2 Page Table = 32 (2^5) entries => 5-bits
Level 3 Page Table = 15-6-5= 4-bits => 16 (2^4) entries

Data Cache Design

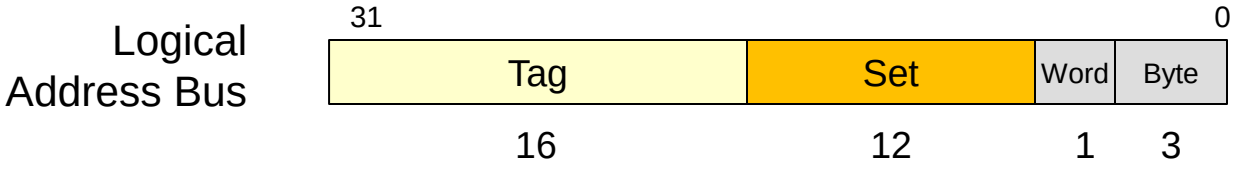
- Use the following specification for the following questions
  - Cache Organization
    - Cache Size: 512KB
    - 8-way set associative
    - Block size: 2 words [Word = 64-bits = 8 bytes]

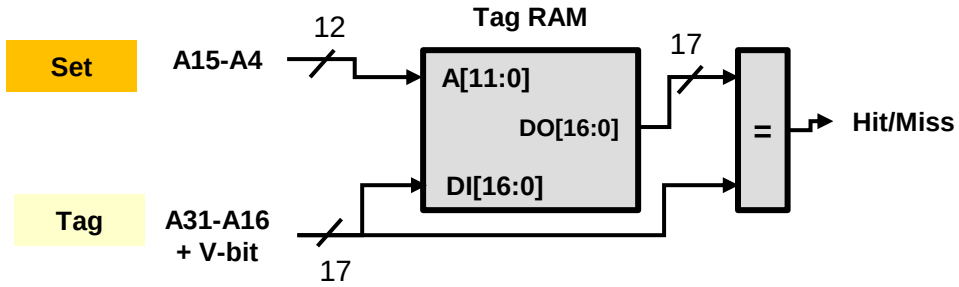


64-bit data = 8 (2^3) bytes per word
Block Size = 2 (2^1) words = 1 word bit
# of Cache blocks = 512KB / 16 bytes per block
= 2^{19} / 2^4 = 2^{15}
# of Sets = 2^{15} / 2^3 ways per set = 2^{12} sets => 12 set bits
# of Tag bits = 32 – 12 – 1 – 3 bits = 16-bits

Data Cache Implementation

- How many comparators and of what size are needed to determine cache hit or miss?
- What is the size of the TAG RAM's?





x 8

8 comparators of 17-bits
Tag RAM Size = 4K x 17