National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 11: Virtual Memory

CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

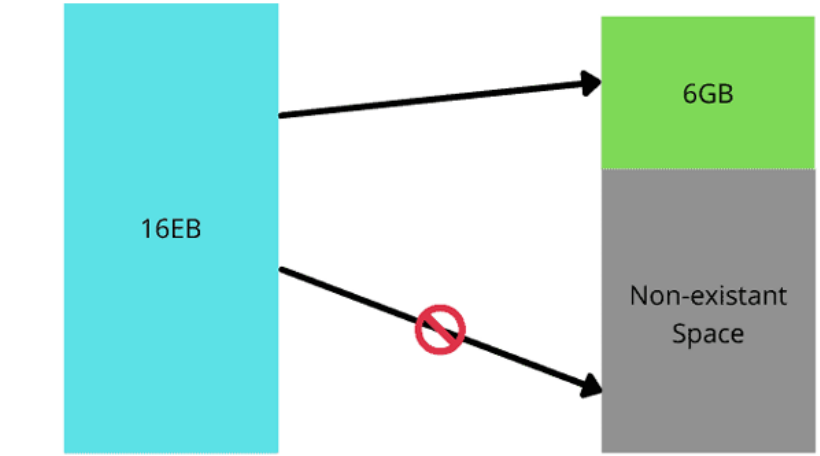
- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS252 at ETHZ
    - https://safari.ethz.ch/digitaltechnik/spring2023
  - CIS510 at Upenn
    - https://www.cis.upenn.edu/~cis5710/spring2019/

Outline

- Virtual Memory
- Paged Memory
- Paged Table
- Multi-Level Page Table
- Translation Lookaside Buffer (TLB)
- Handling TLB Misses

Virtual Memory Motivation

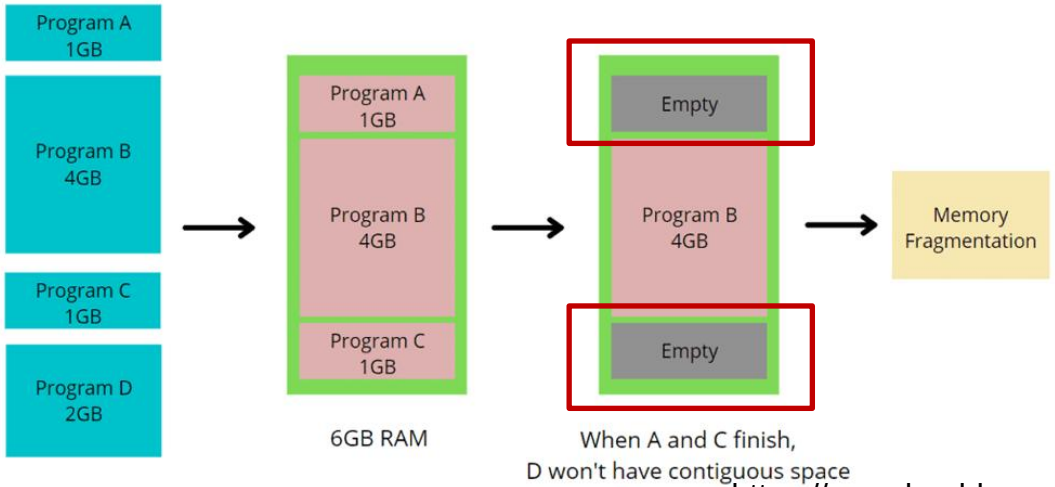
- OS allows a program to use certain range of RAM addresses
  - Try to use addresses that are out of range will crash the computer



https://www.baeldung.com/cs/virtual-memory-why

Virtual Memory Motivation

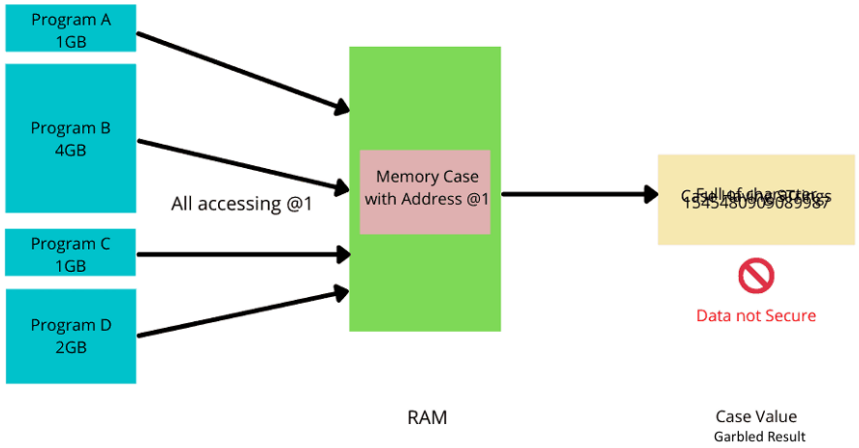
- OS assign programs a continuous partition of RAM
  - If the space freed up by the two programs is not continuous
  - Not enough for other programs to run
  - The RAM will have holes in different places



https://www.baeldung.com/cs/virtual-memory-why

Virtual Memory Motivation

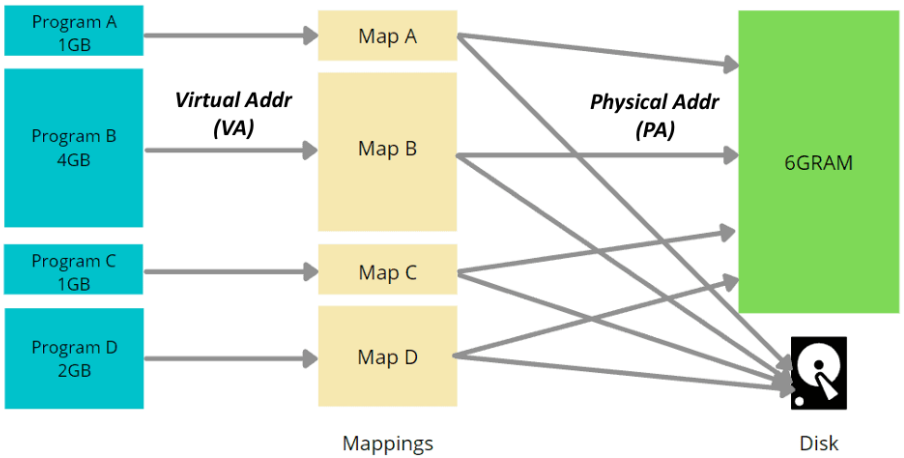
- Many programs are executed simultaneously
  - More than one program can access the same case of memory
  - To change their value, programs can collide with each other
  - Corrupt the memory or crash the system



https://www.baeldung.com/cs/virtual-memory-why

Virtual Memory Mapping

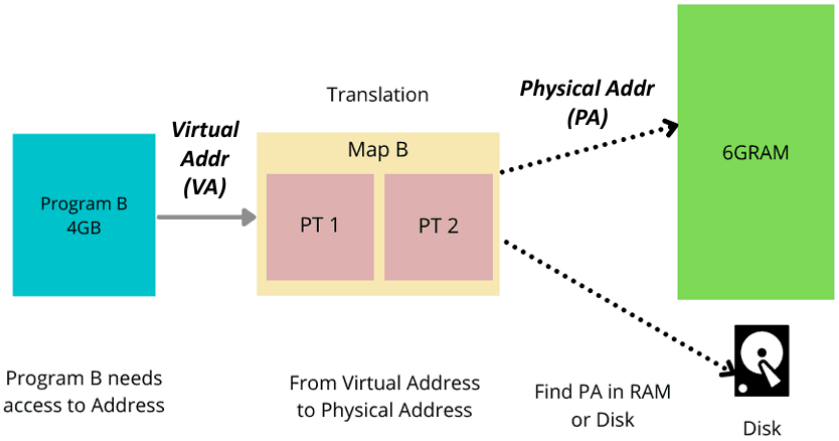
- Virtual memory maps program addresses in to RAM address
  - If no more space is available, addresses will be mapped in disk
  - Virtual memory enables OS to load programs larger than its physical memory



https://www.baeldung.com/cs/virtual-memory-why

Virtual Memory Mapping

- Each program sees a virtual address (VA) that is mapped to a physical address (PA)
  - Virtual memory needs to use page tables and translations
  - Page table aims to map VA to PA -> translations



https://www.baeldung.com/cs/virtual-memory-why

Virtual Memory Principles

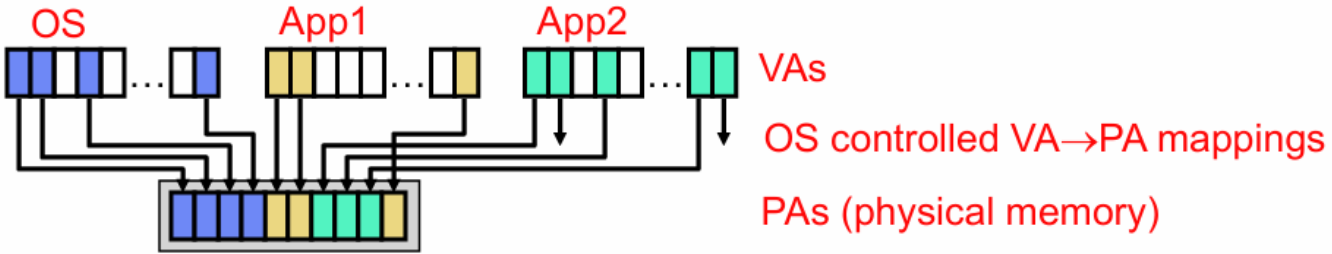
- **Virtual memory provides**
  - Illusion of a large memory while only keeping a subset of data in physical memory
  - Ability to run concurrent processes
    - Protection between processes
      - Processes don’t access each other’s memory

Virtual Memory Logistics

- **Processors generate virtual addresses**
  - Each process can use entire virtual address space, which provides illusion of a large, private, contiguous memory
- Physical memory acts as **cache** for virtual memory
  - A subset of virtual addresses are kept in physical memory
  - Page: chuck of memory moved from virtual to physical memory, typically 4 KiB
- The **MMU** (a hardware block in the processor)
  - Translate virtual addresses to physical addresses

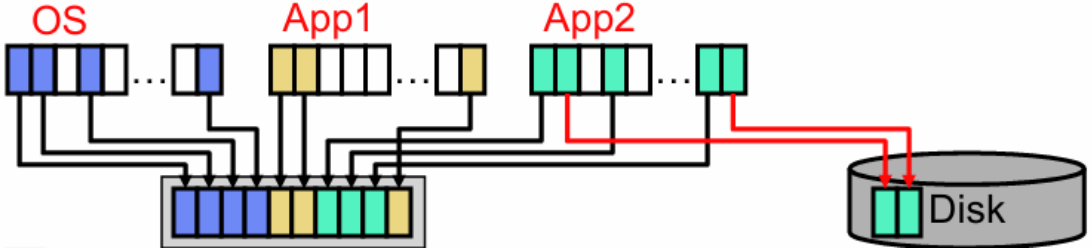
Virtualizing Memory

- **Virtual Memory (VM)**
  - Level of indirection
  - Application generated addresses are **virtual addresses (VAs)**
    - A process ***thinks*** it has its own 2^N bytes of address space
  - Memory accessed using **physical addresses (PAs)**
  - VAs translated to PAs at coarse (page) granularity
  - OS controls VA to PA mapping for itself and all processes
  - Logically: translate before every insn fetch, load, store
    - but hardware acceleration removes translation overhead



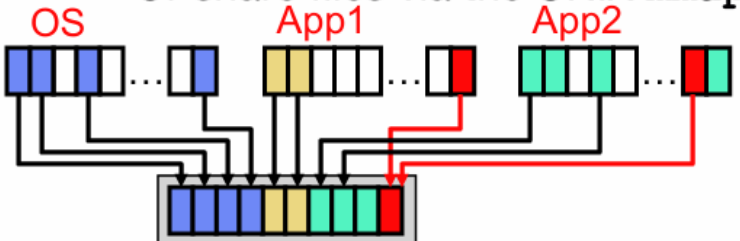
Virtualizing Memory

- Programs use **virtual addresses (VA)**
  - VA size (N) aka pointer size (these days, 64 bits)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, often M=48/56 and N=64
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if unaccessed)



Uses of Virtualizing Memory

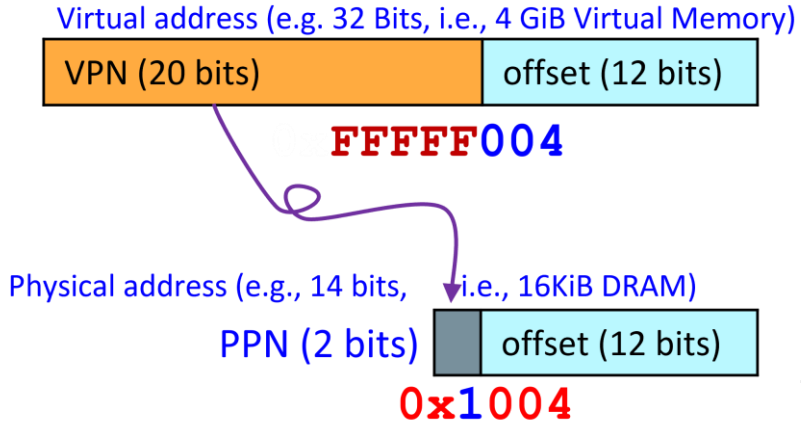
- More recently: **isolation** and **multi-programming**
  - Stack always starts at 0xFFFFFFFF for each process
  - Apps prevented from reading/writing each other's memory
    - Can't even address the other program's memory!
- **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware
- **Inter-process communication**
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX `mmap()` call



Paged Memory

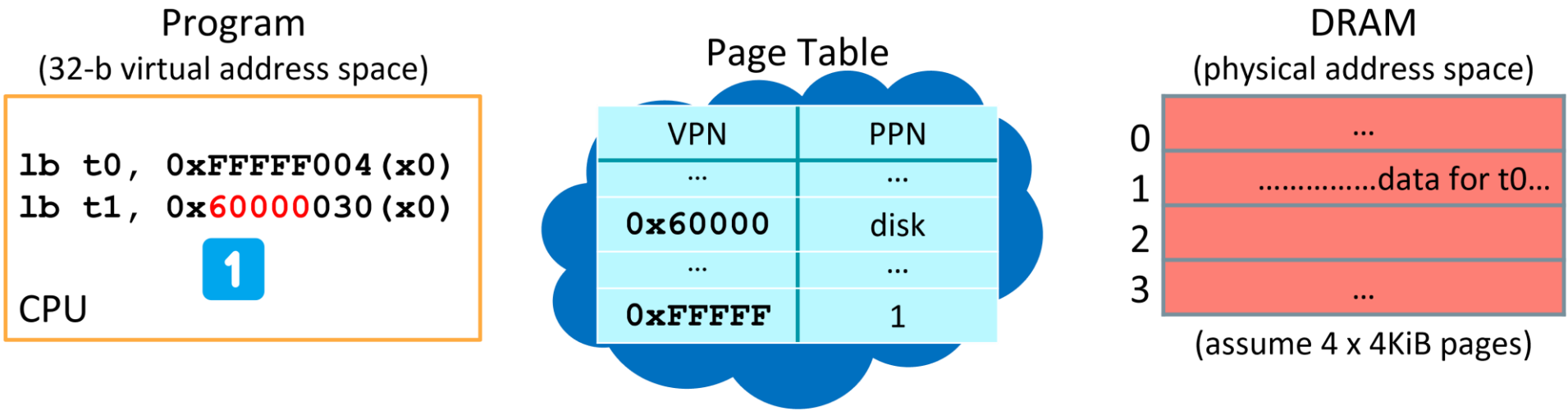
- The concept of “paged memory” dominates
  - Physical memory (DRAM) is broken into pages
  - A disk access loads an entire page into memory
  - Typical page size: 4KiB+ (on modern OSs)
    - Need 12 bits of page offset to address all 4KiB

Memory translation maps
Virtual Page Number (VPN) to
a Physical Page Number (PPN)



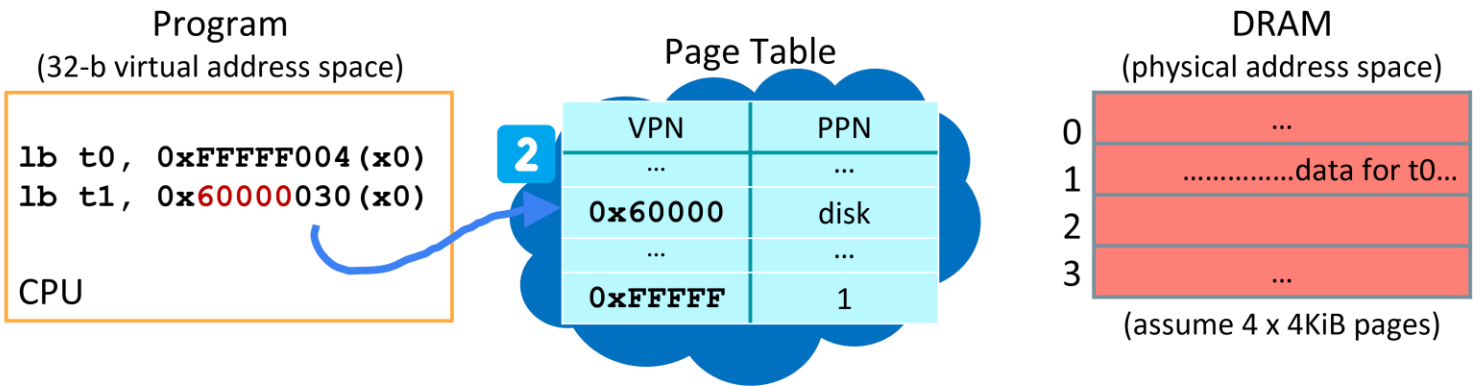
Paged Memory

- How a program accesses memory?
  - Program executes a load specifying a virtual address (VA)



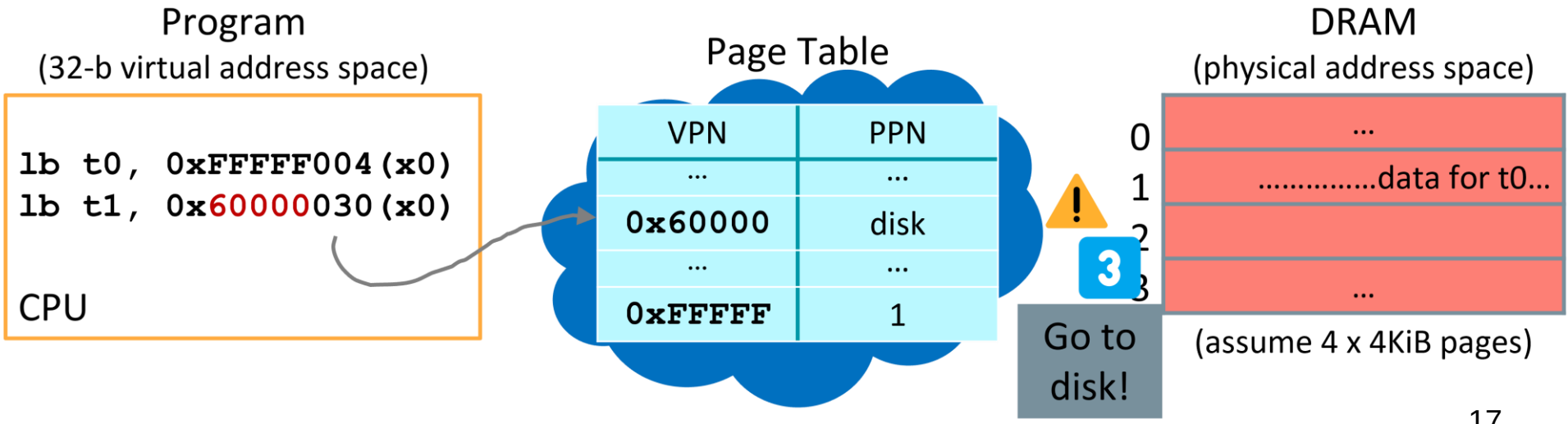
Paged Memory

- How a program accesses memory?
  - Computer translates VA to the physical address (PA) in memory
    - Extract virtual page number (VPN) from VA, e.g. top 20 bits if page size 4KiB = 2^{12} B
    - Look up physical page number (PPN) in page table
    - Construct PA: physical page number + offset (from virtual address)



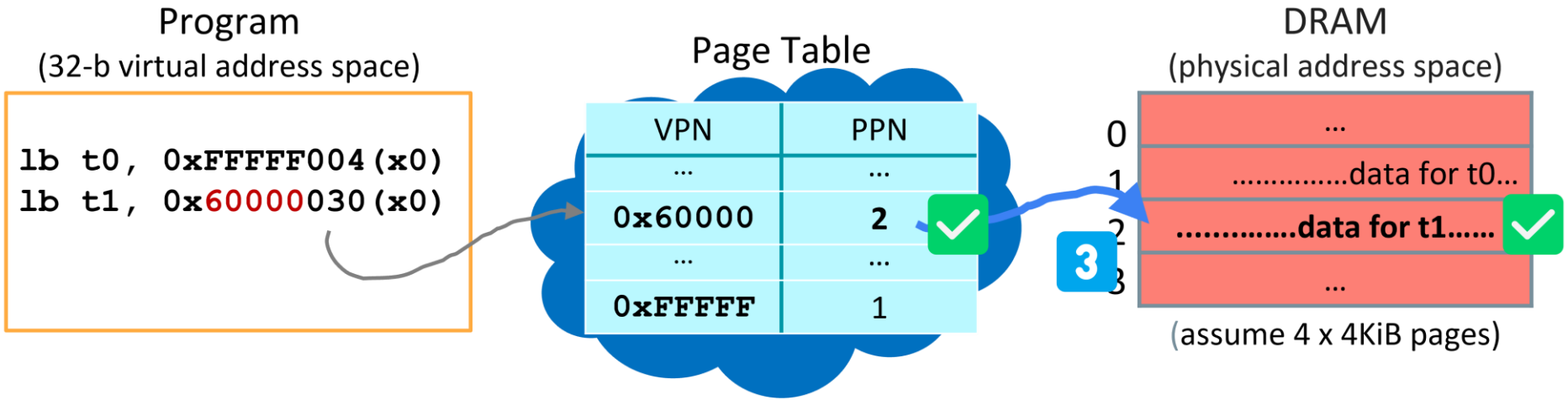
Paged Memory

- How a program accesses memory?
  - If the physical page is not in memory, then OS loads it in from disk

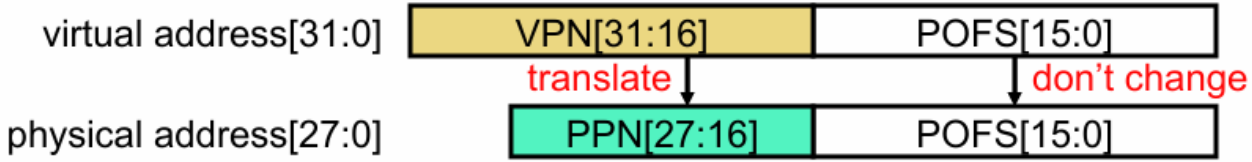


Paged Memory

- How a program accesses memory?
  - The OS reads memory at the PA and returns the data to the program

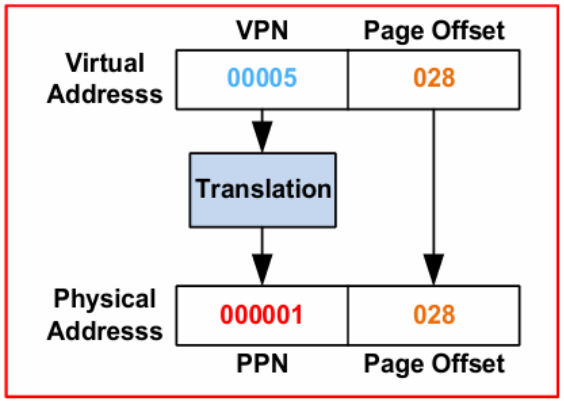


Address Translation



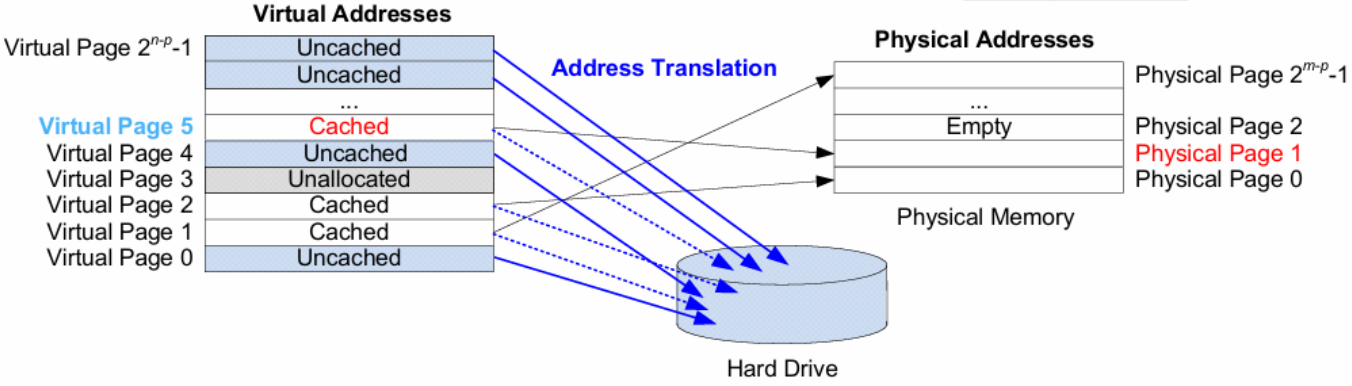
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Address Translation



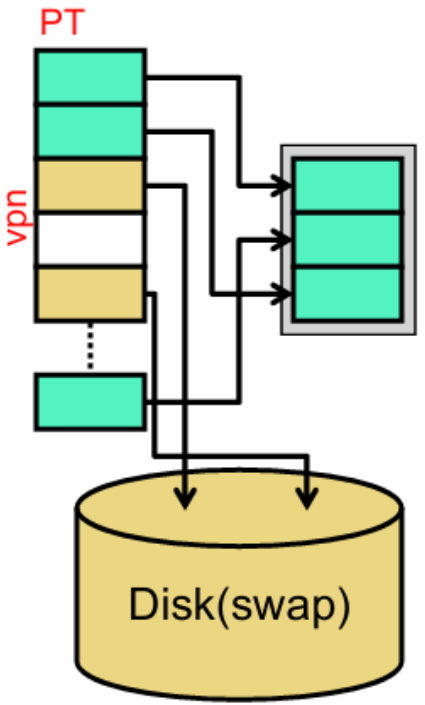
Page offset remains the same – only the page number is translated.

VPN	PPN	Status
2^{n-p}- 1		Uncached
2^{n-p}- 2		Uncached
...	...	...
5	000001	Cached
4		Uncached
3		Unallocated
2	000000	Cached
1	3FFFFF	Cached
0		Uncached



Address Translation Mechanics

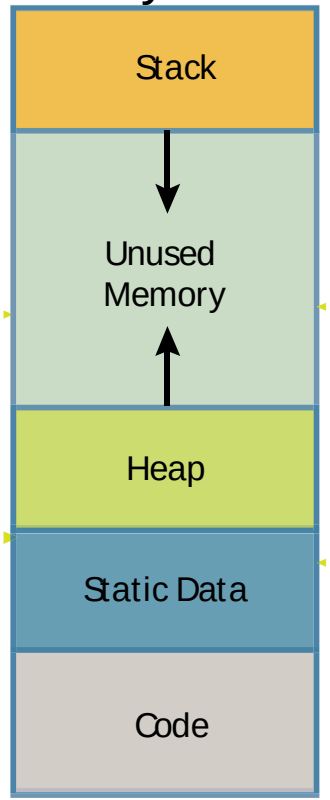
- How are addresses translated?
  - In software (for now) but with hardware acceleration (a little later)
- Each process has a **page table (PT)**
  - **Software data structure constructed by OS**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup



Process's VA Space

- Each process contains 4 regions in VA
  - **Stack:**
    - local variables, grows downward
  - **Heap:**
    - space requested for pointers via malloc(); resizes dynamically, grows upward
  - **Static data:**
    - Variables declared outside main, does not grow or shrink
  - **Code:**
    - Loaded when program starts, does not change

A process memory address layout



Page Table

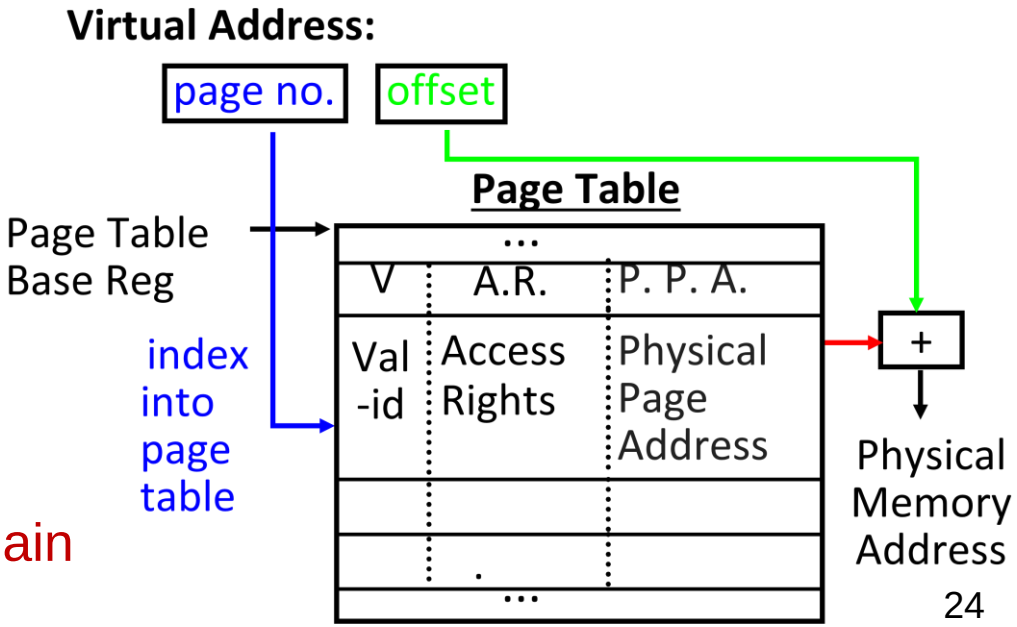
- **Page table**
  - Equal-sized frames
  - List mapping of virtual to physical pages
  - It also holds the virtual page's status and the PPN (physical page number)
  - Each VPN (virtual page number) has an entry

VPN	PPN	Status
$2^{n-p}-1$		Uncached
$2^{n-p}-2$		Uncached
...	...	...
5	000001	Cached
4		Uncached
3		Unallocated
2	000000	Cached
1	3FFFFF	Cached
0		Uncached

Paged Table

- A page table contains the mapping of virtual address to physical locations
  - Each process has its own page table
  - OS changes page tables by changing contents of Page Table Base Register

Page tables are stored in main memory

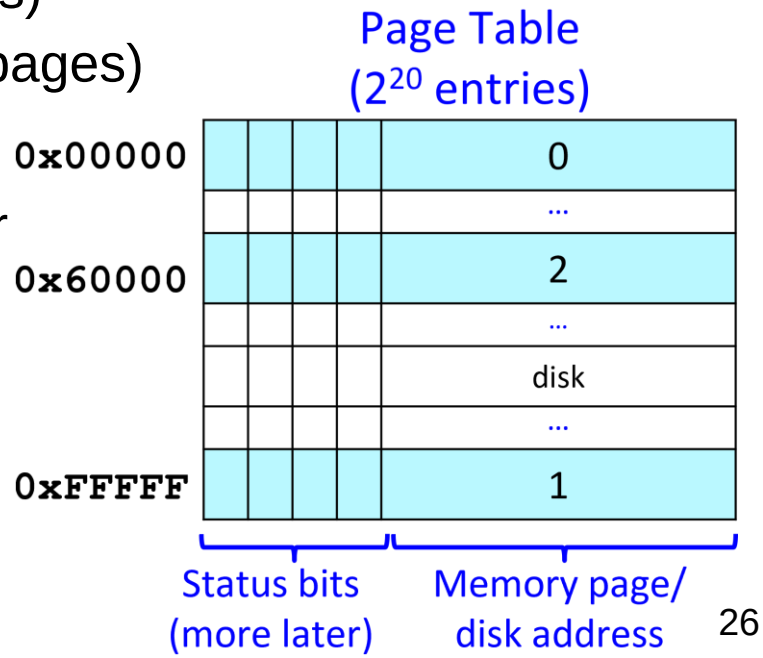


Pages

- **Page size:** P = 2^p bytes
- **Virtual address:** n bits (N = 2^n bytes of virtual memory)
  - Typically, n = 32/64 for 32/64-bit systems
  - Upper bits in 64-bit systems may not be used (few applications yet require N = 2^{64} = 16 exabytes (EBs) of memory)
- **Physical address:** m bits (M = 2^m bytes of physical memory)
- Number of virtual pages: 2^{n-p}
- Number of physical pages: 2^{m-p}

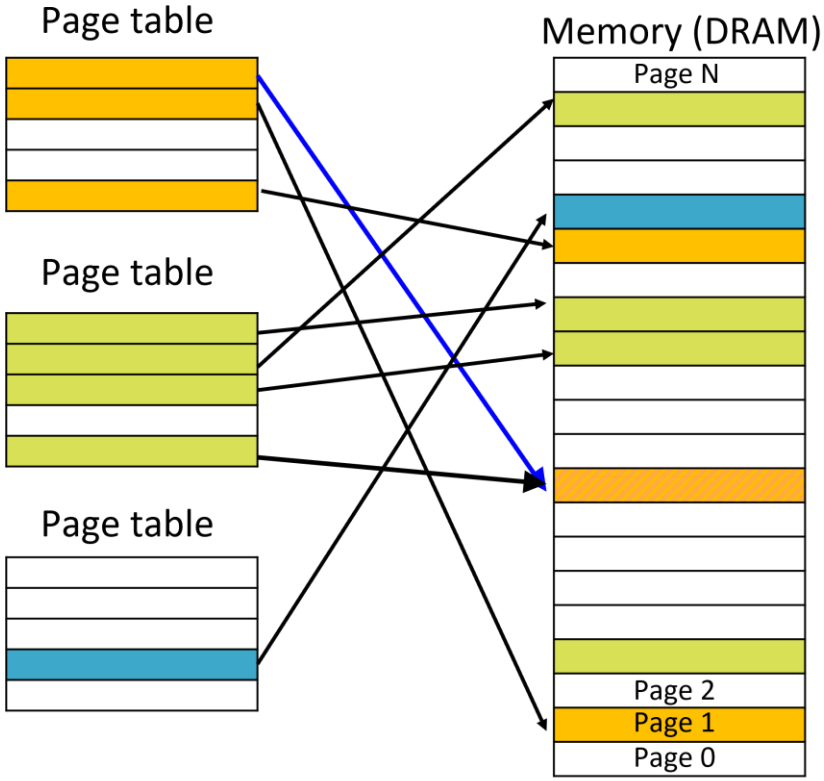
Paged Table

- 32-bit virtual address space, 4-KiB pages
  - 2^{32} virtual addresses / (2^{12} B/pages)
  - = 2^{20} virtual page numbers (1MB pages)
- One page table per process
  - One entry per virtual page number
  - Entry has physical page number

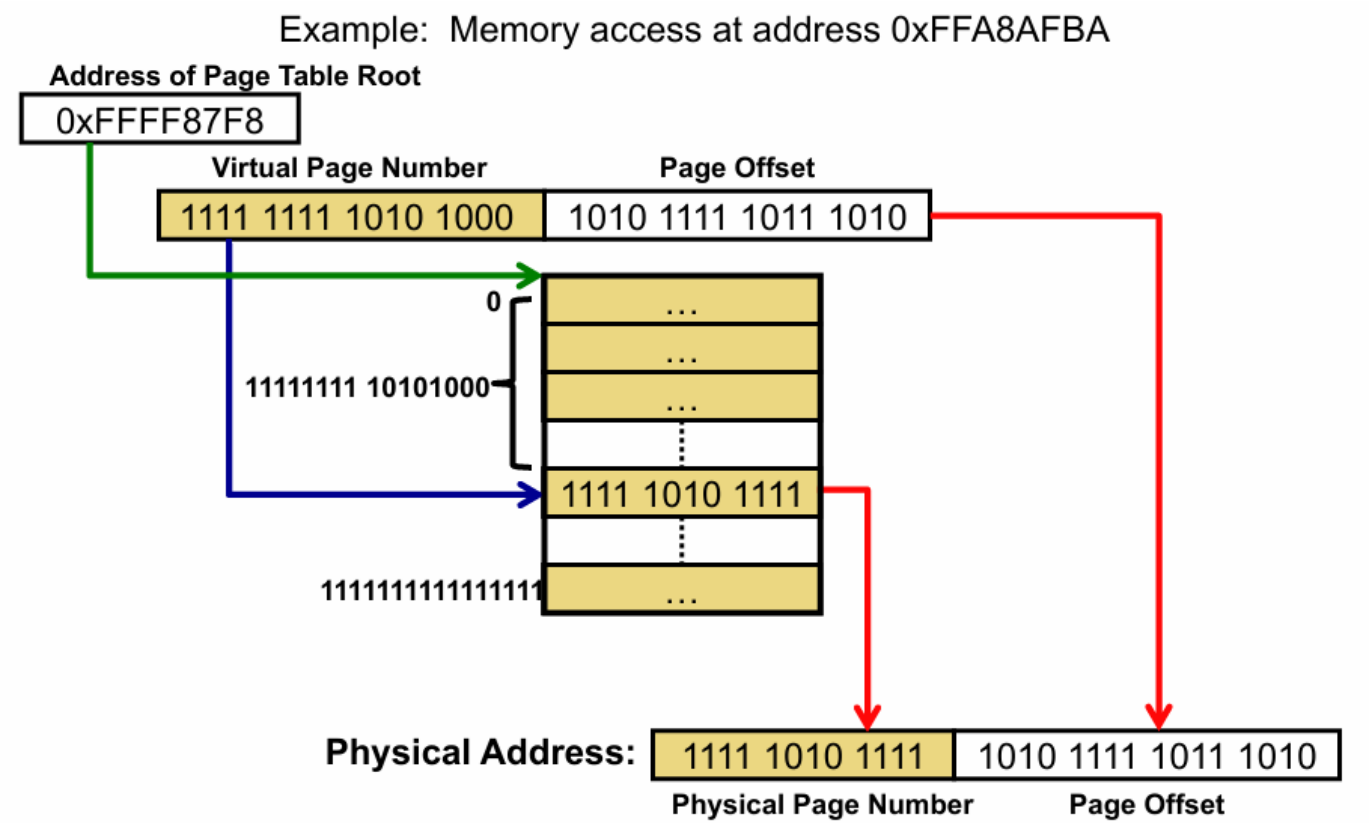


Paged Table

- Each process has a dedicated page table
  - OS keeps track of which process is active
- **Isolation**: Assign processes different pages in DRAM
  - Prevent accessing other processes' memory
  - OS may assign some physical page to several processes (e.g. system data), sharing is also possible

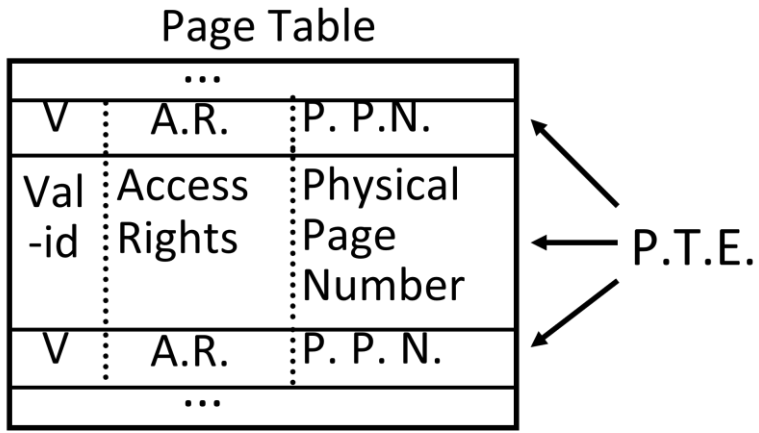


Translation Using a Paged Table



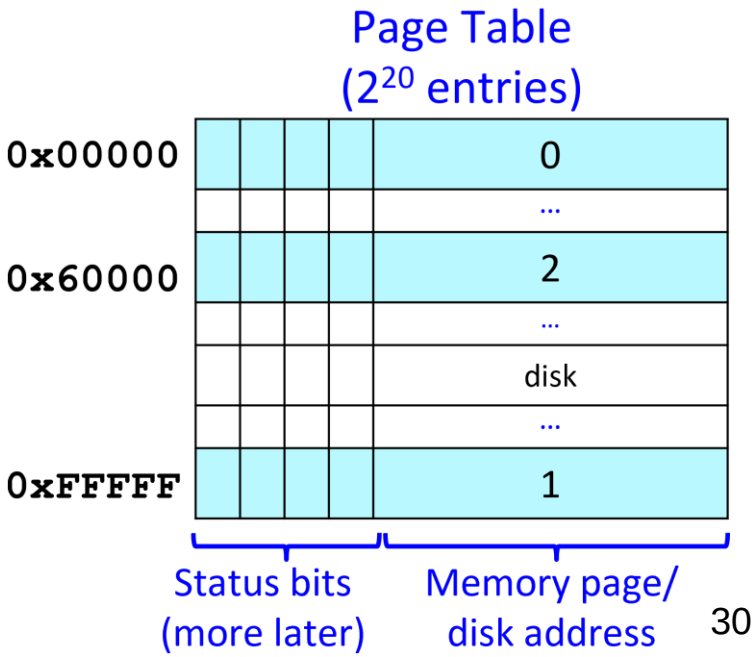
Paged Table

- **Page Table Entry (PTE) format**
  - Contains either physical page number or indication not in main memory
  - OS maps to disk if Not Valid (V=0)
  - If valid, also check if have permission to use page
    - Access Rights (A.R.) may be Read Only, Read/Write, Executable



Paged Table

- **Status Bits**
  - Write protection bit
    - On: If process writes to page trigger exception
  - Valid bit
    - On: Page is in RAM
  - Dirty bit
    - On: page on RAM is more up-to-date than page on disk



Paged Table Size

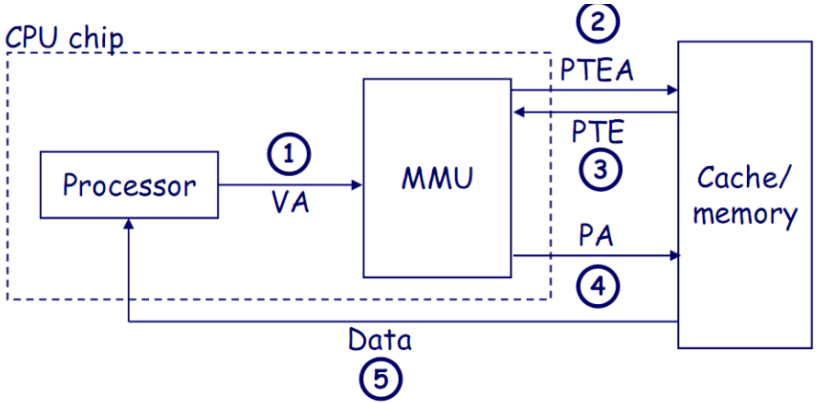
- How big is a page table on the following machine?
  - 32-bit machine->32-bit VA -> 2^{32} = 4GB virtual memory
  - 4B page table entries (PTEs)
  - 4KB pages
  - 4GB virtual memory / 4KB page size -> 1M VPs
  - 1M VPs * 4 bytes per PTE -> 4MB
- What is the problem when increasing page size from 4 KB to 16 KB?
  - **Internal fragmentation** (big pages lead to waste within each page)
- Page tables can get big
  - There are ways of making them smaller

Paged Table

Cache version	Virtual Memory vers.
Block or Line	Page
Miss	Page Fault
Block Size: 32-64B	Page Size: 4K-8KB
Placement:	Fully Associative
Direct Mapped, N-way Set Associative	
Replacement:	Least Recently Used
LRU or Random	(LRU)
Write Thru or Back	Write Back

Paged Table

- **Page hit**
  - 1) Processor sends virtual address to MMU
  - 2 – 3) MMU fetches PTE from page table in memory
  - 4) MMU sends physical address to L1 cache
  - 5) L1 cache sends data word to processor



Paged Table

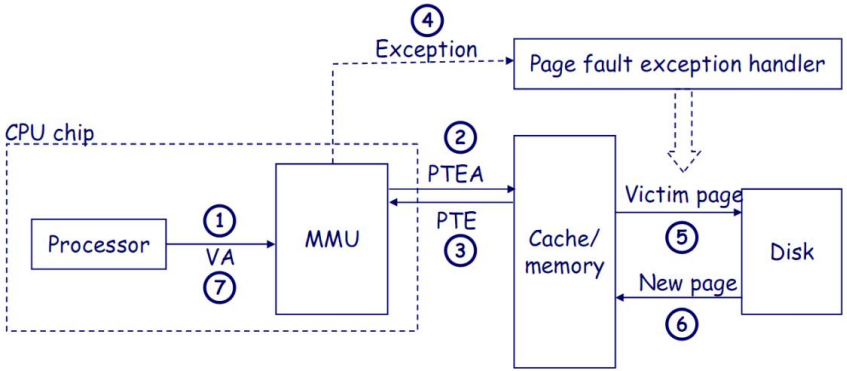
- **Page faults**
  - Page table entries store status to indicate if the page is in memory (DRAM) or only on disk
    - One each memory access, check the page table entry “valid” status bit
  - Valid -> in DRAM
    - Read/write data in DRAM
  - Not valid -> on disk
    - Trigger a page fault; OS intervenes to allocate the page into DRAM
    - If out of memory, first evict a page from DRAM (LRU/FIFO/random)
    - Read request page from disk into DRAM
    - Finally, read/write data in DRAM

Page Fault

- **Page fault**: PTE not in TLB or page table
  - page is not in memory
  - If no valid mapping for this page → segmentation fault
  - Starts out as a TLB miss, detected by OS/hardware handler
- OS chooses a page to replace
  - **"Working set"**: refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Whose page are we evicting?
    - **frame map** maps physical pages to <process,virtual page> pairs
  - Update page tables, flush TLBs, retry memory access

Page Fault

- **Page faults**
  - 1) Process sends virtual address to MMU
  - 2-3) MMU fetches PTE from page table in memory
  - 4) Valid bit is 0, so MMU triggers page fault exception
  - 5) Handler identifies victim, and if dirty pages it out to disk
  - 6) Handler pages in new page and updates PTE in memory
  - 7) Handler returns to original process, restarting faulting instruction

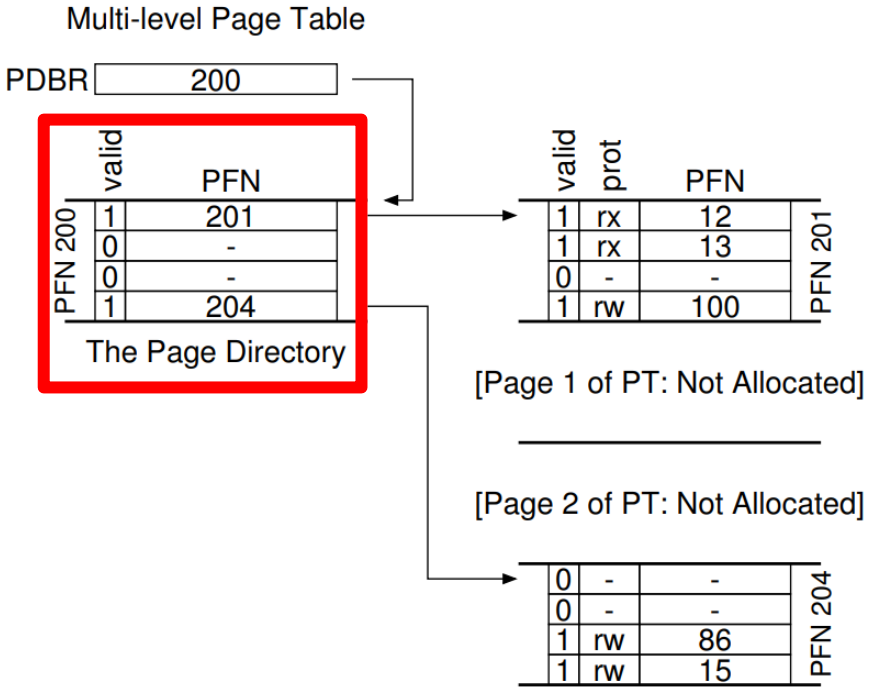


Multilevel Page Tables

- Page table can be
  - unreasonably large and be mostly unused (many PTEs are often unallocated)
  - Example 1: Determine page table size for:
    - n = 32 (virtual address size), p = 12 (4 KiB pages)
    - # PTEs = $2^{32-12} = 2^{20}$
    - At 4 bytes/PTE, page table size is 2^{20} x 4 bytes = 4MiB

Multi-Level Page Table

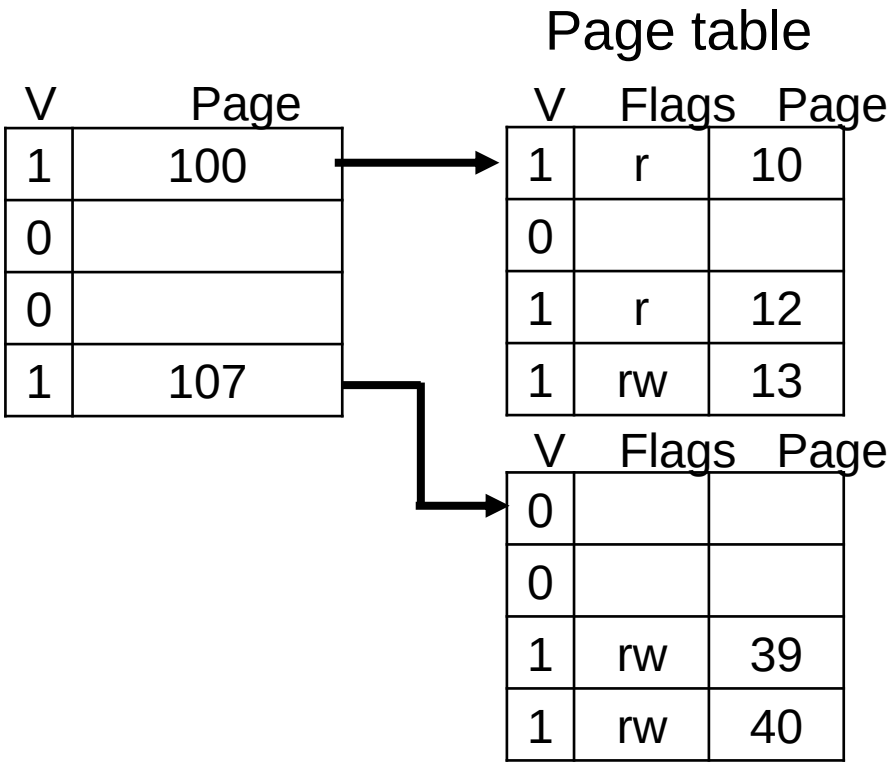
- **Multi-level page table**
  - Chop up the page table into page-sized units
  - **Page directory** tells where a page of the page table is
    - A number of **page directory entries (PDE)**
    - A **page frame number (PFN),** and a valid bit



https://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf

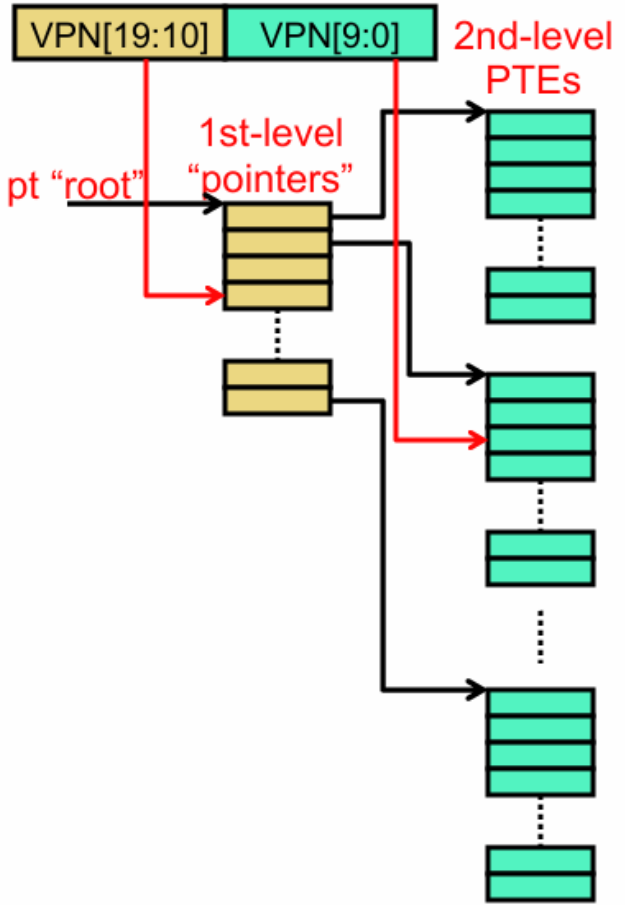
Multi-Level Page Table

- What are the advantages of multi-level page table?
  - Only allocate “using” page-table space
  - Compact and supports **sparse** address space

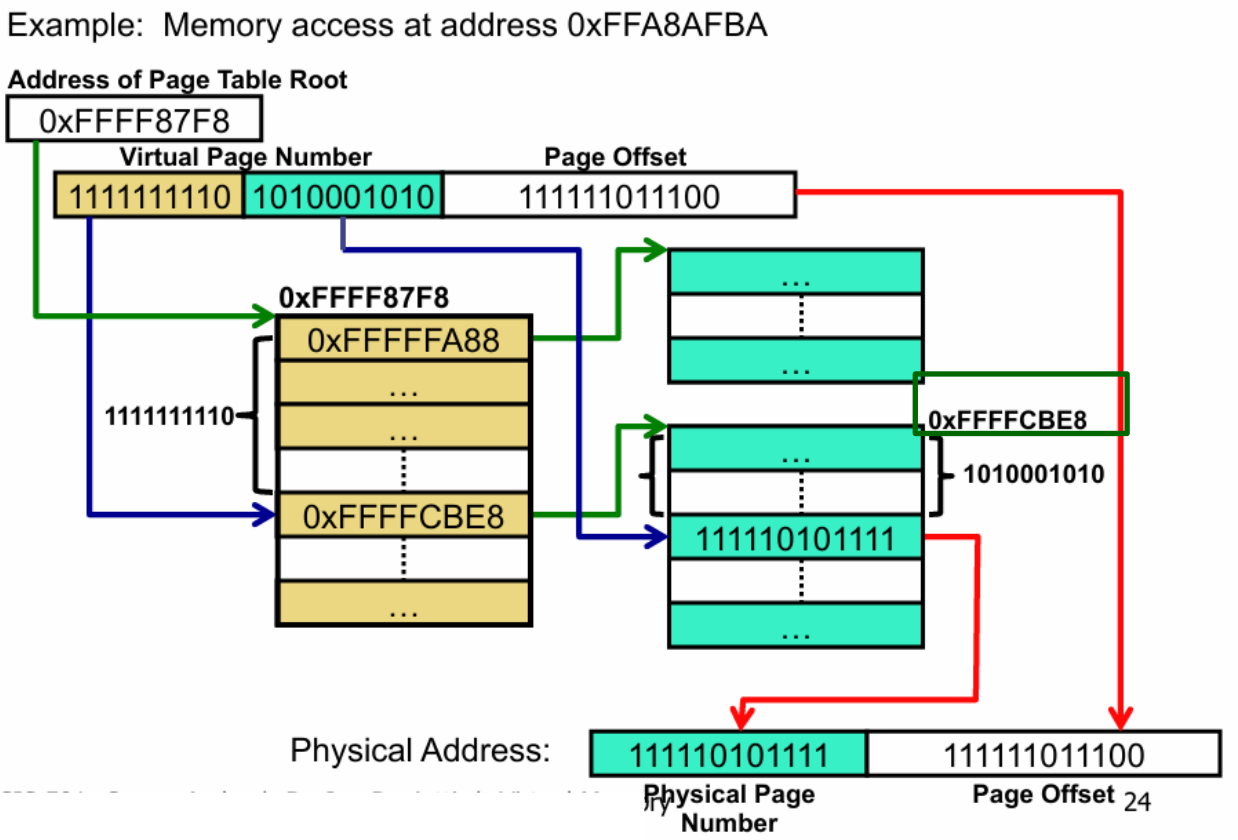


Multi-Level Page Table

- **Multi-level page tables**
  - Tree of page tables (“trie”)
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels
- 20-bit VPN
  - Upper 10 bits index 1st –level table
  - Lower 10 bits index 2nd-level table



Multi-Level Page Table



Multi-Level Page Table

- **Have we saved any space?**
  - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but
- **Large virtual address regions unused**
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- **How large for contiguous layout of 256 MB?**
  - Each 2nd-level table maps 4MB of virtual addresses
  - One 1st-level + 64 2nd-level pages
  - 64 total pages = 260 KB (much less than 4MB)

Multi-Level Page Table

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 Bytes
- Initially
  - Page size = 8 KB = 2^{13} Bytes
  - Virtual address space size = 2^{46} Bytes
  - PTE = 4 Bytes = 2^{2} Bytes
  - Number of pages or number of entries in page table = 2^{46} Bytes / 2^{13} Bytes = 2^{33}
  - Size of page table = 2^{33} x 2^{2} Bytes = 2^{35} Bytes

Multi-Level Page Table

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{35} B > 2^{13} B)
  - Create one more level
  - Number of page tables in last level
    2^{35} Bytes / 2^{13} Bytes = 2^{22}
  - Size of page table [second last level]
  - 2^{22} x 2^{2} Bytes = 2^{24} Bytes

Multi-Level Page Table

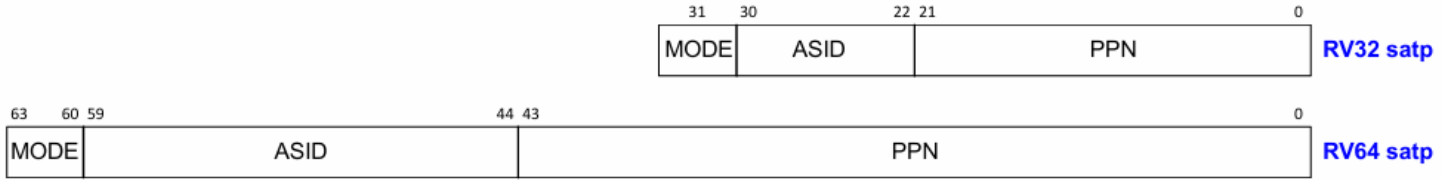
- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{24} B > 2^{13} B)
  - Create one more level [third last level]
  - Number of page tables in second last level
    = 2^{24} Bytes / 2^{13} Bytes = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} Bytes = 2^{13} Bytes = page size

Multi-Level Page Table

- How many levels of page tables would be required ?
  - A virtual memory system with physical memory of 8 GB, a page size of 8 KB, 46 bit virtual address, and PTE size is 4 B
- Now, size of page table > page size (2^{24} B > 2^{13} B)
  - Create one more level [third last level]
  - Number of page tables in second last level
    = 2^{24} Bytes / 2^{13} Bytes = 2^{11}
  - Size of page table [third last level]=
    = 2^{11} x 2^{2} Bytes = 2^{13} Bytes = page size

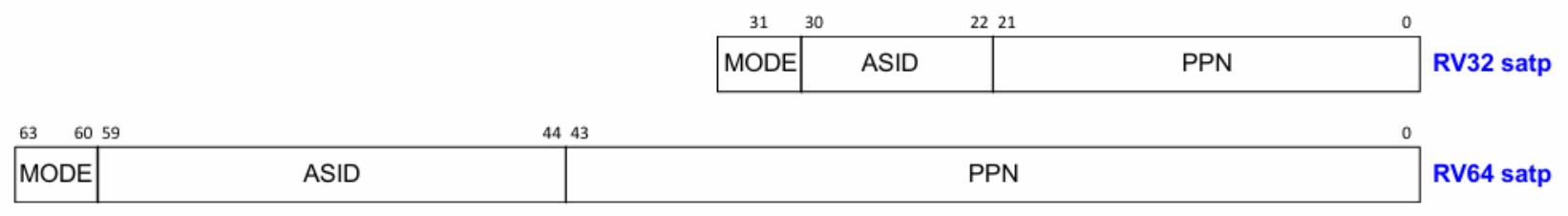
RISC-V MMU

- **satp register**
  - Supervisor address translation and protection register
  - Enables virtual memory
  - **PPN**: base address of the root (highest-level) page table
    - Shift left by 12 bits to get physical address
  - **ASID**: address space identifier
    - Short PID (process ID) to differentiate multiple virtual address space in the TLB

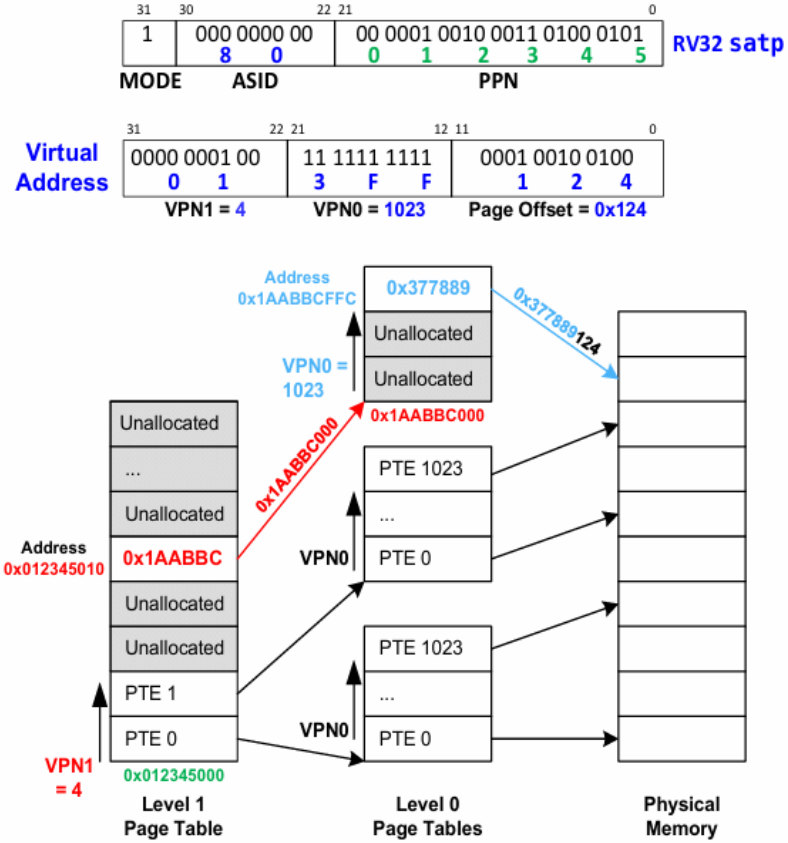


RISC-V MMU

- **satp register**
  - **MODE**: bare metal (no virtual memory) or virtual memory mode
  - For RV32
    - MODE = 0: no virtual memory (Bare mode)
    - MODE = 1: virtual memory



Example 2-Level Page Table



satp = 0x8012345

- **MODE = 1** (virtual memory enabled)
- **PPN = 0x012345.** Address of Level 1 page table is 0x12345 << 12 = **0x012345000** (physical memory address is 34 bits)

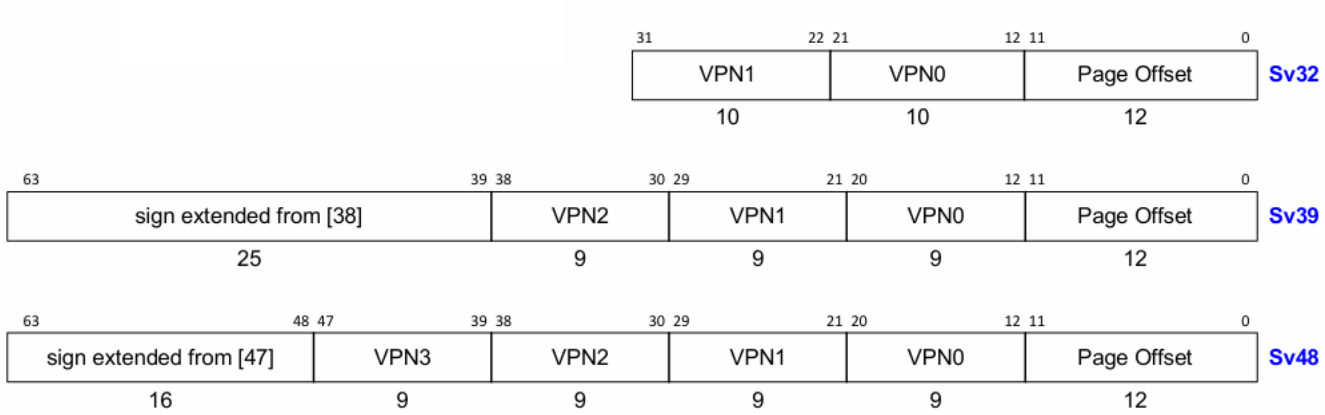
Virtual Address = 0x013FF124

- **VPN1 = 4:** entry 4 of Level 1 page table = **0x1AABBC**, so Level 0 page table is at address: 0x1AABBC << 12 = **0x1AABBC000**
- **VPN0 = 1023:** entry 1023 of Level 0 page table = **0x377889**, so append Page Offset (**0x124**) to this to get physical address:

0x377889124

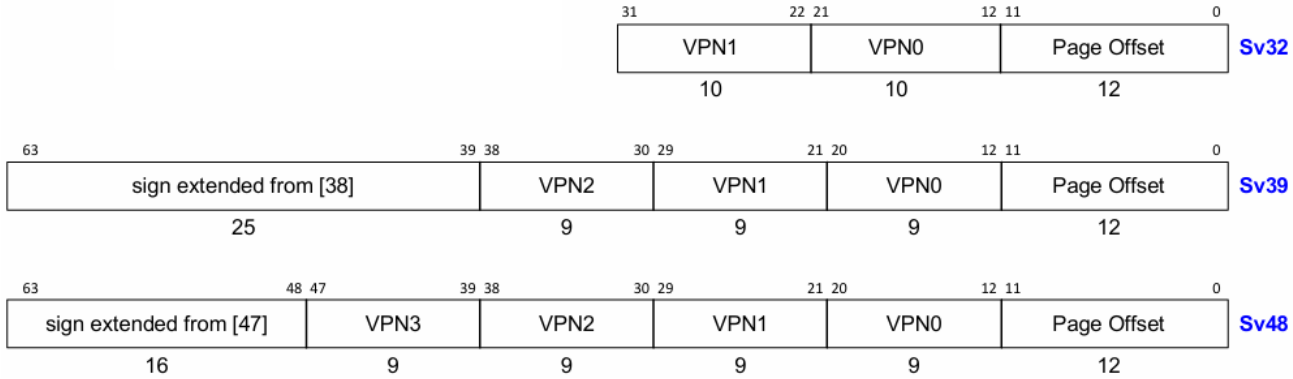
RISC-V Virtual Memory Modes (Sv*)

- **RV32**
  - **Sv32:** virtual memory address = 32 bits
  - PTEs are 4 bytes (32 bits)



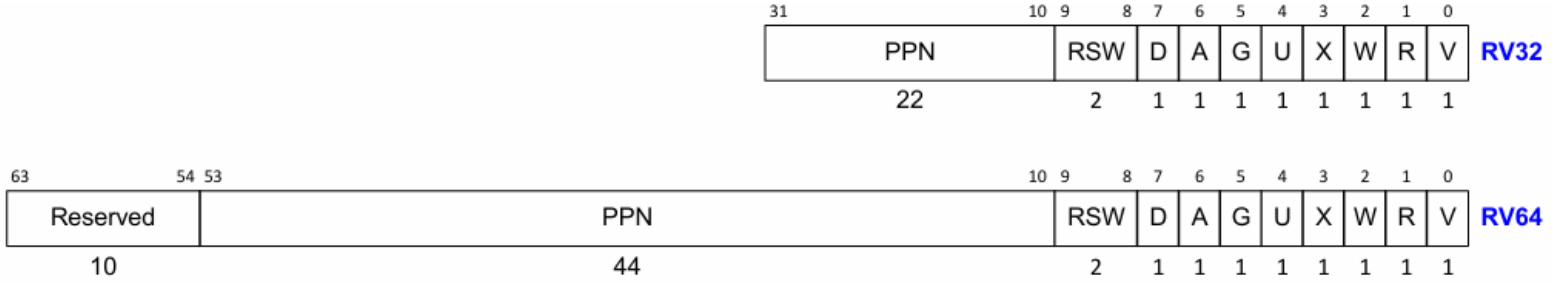
RISC-V Virtual Memory Modes (Sv*)

- **RV64**
  - **Sv39:** virtual memory address = 39 bits (satp MODE = 1000)
  - **Sv48:** virtual memory address = 48 bits (satp MODE = 1001)
  - **Sv57:** virtual memory address = 57 bits (satp MODE = 1010)
  - PTEs are 8 bytes (64 bits)



RISC-V Virtual Memory Modes (Sv*)

- **RV32**
  - **PPN = 22 bits:** physical address = 22 + 12 (page offsets) = 34 bits (up to 2^{34} = 16 GiB)
- **RV 64**
  - **PPN = 44 bits:** physical address = 44 + 12 (page offsets) = 56 bits (up to 2^{56} = 64 GiB)



PTE Flags

Flag	Meaning
V	**Valid:** 1 if the page is in main memory, 0 if not
X, W, R	**Execute, Write, Read** Permissions: 000: Pointer to next level of page table 001: Read only 010: <illegal> 011: Read-write 100: Execute only 101: Read-execute 111: Read-write-execute
U	**User** mode
G	**Global:** page exists for all address spaces; ignore ASID in TLB matching
A	**Accessed:** page has been accessed at least once since A bits were last cleared
D	**Dirty:** at least one byte from the page has been written
RSW	**Reserved** for operating system

Fast Translation Using TLB

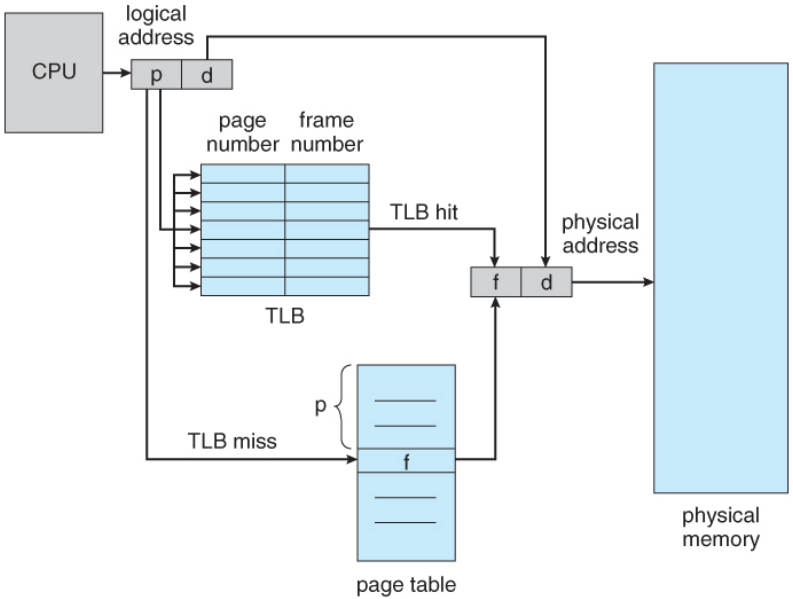
- Good virtual memory design should be fast (~1 clock cycle) and space efficient
  - Every instruction/data access needs address translation
- Address translation would appear to require extra memory references
  - One to access the PTE
  - Then the actual memory access
  - We must perform a page table walk per instruction/data access
    - Single-level page table: 2 memory accesses
    - Two-level page table: 3 memory accesses

Fast Translation Using TLB

- But access to page tables has good locality
  - Use a fast cache of PTEs within the CPU
  - Called a **Translation Look-aside Buffer (TLB)**
  - Typical: 16-512 PTEs, 0.5-1 cycle for hit, 10-100 cycles for miss, 0.01%-1% miss rate

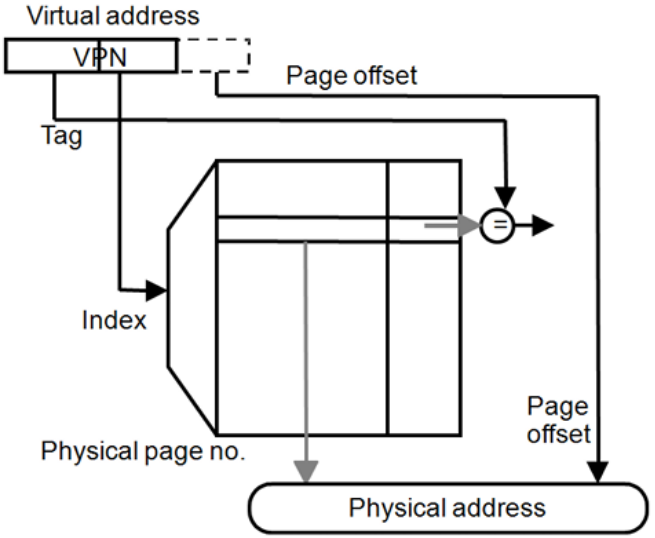
Translation Lookaside Buffer (TLB)

- Translation lookaside buffer (TLB)
  - **Small cache**: 16-64 entries
  - **Associative** (4+ way or fully associative common)
  - Exploit temporal locality in page table
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler, walk page table



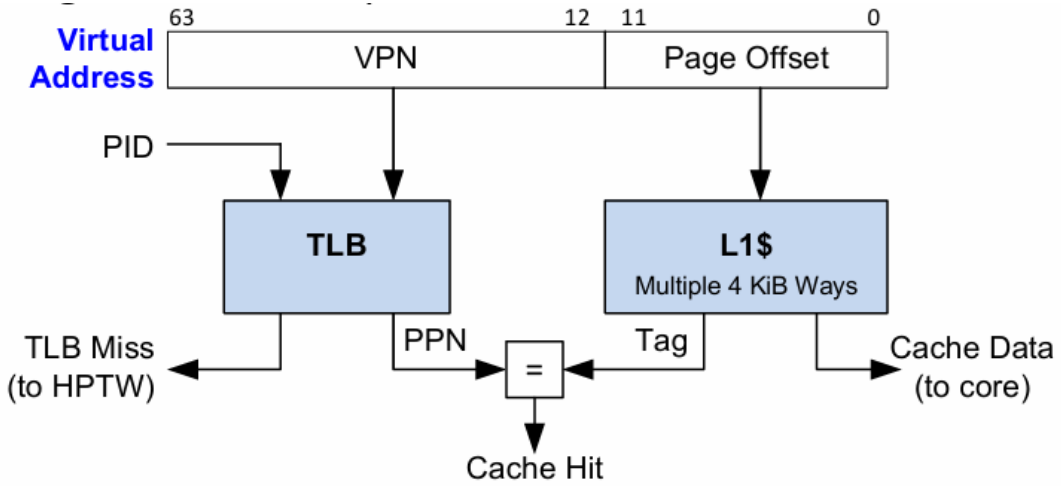
Translation Lookaside Buffer (TLB)

- Translation lookaside buffer (TLB)
  - A cache of address translations
  - Avoid accessing the page table on every memory access
  - **Index** = lower bits of VPN (virtual page #)
  - **Tag** = unused bits of VPN + process ID
  - **Data** = a page-table entry
  - **Status** = valid, dirty



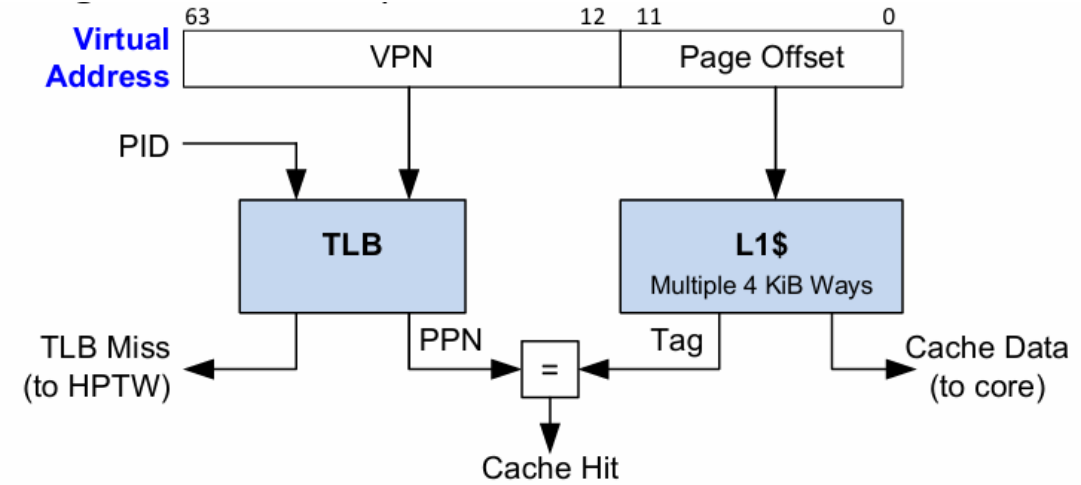
TLB: Translation Lookaside Buffer

- Most systems have a D$ and I$
  - Usually physically addressed
  - Set is given by page offset bits (lower bits of address)
  - TLB & cache access occurs simultaneously (using page offset bits)

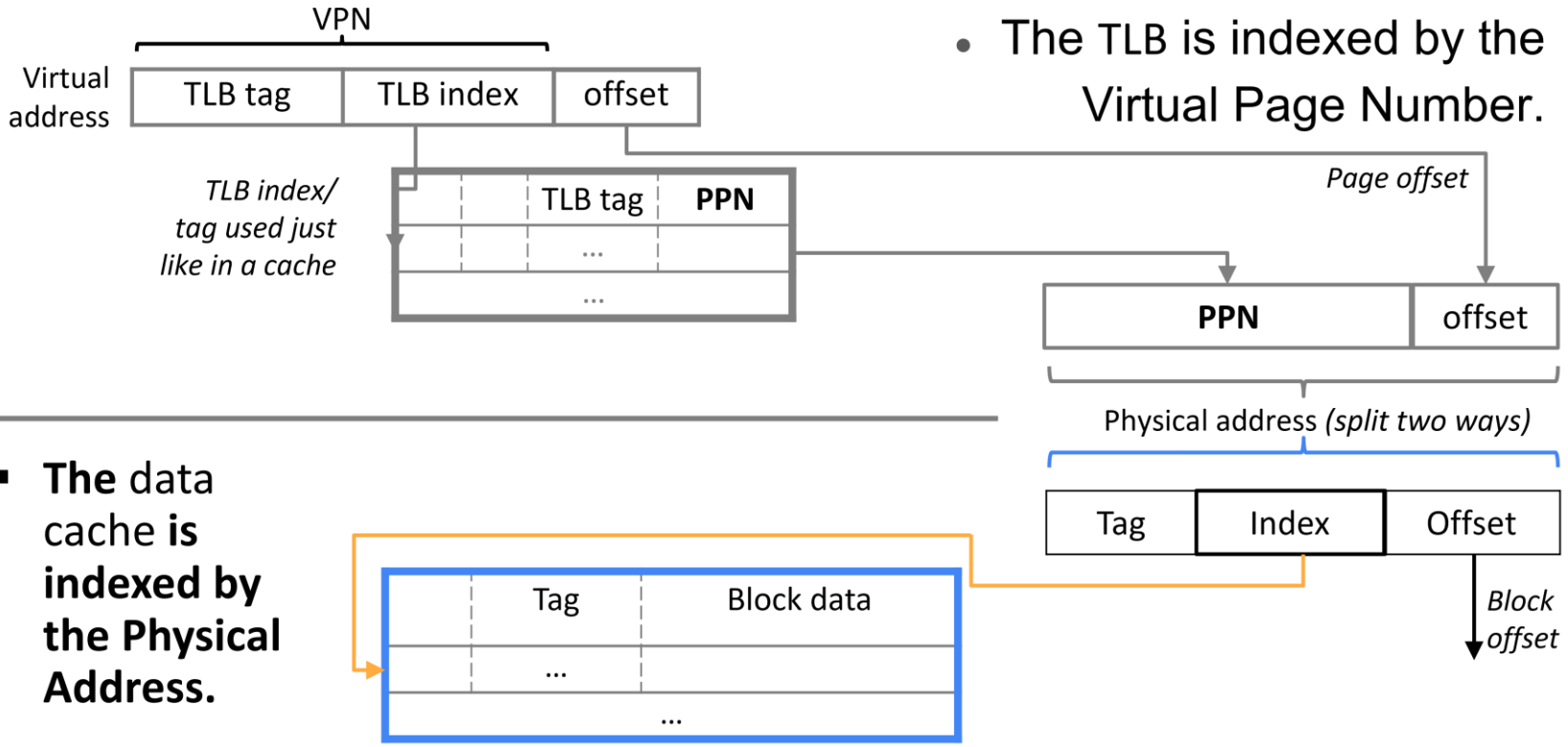


TLB: Translation Lookaside Buffer

- Most systems have a D$ and I$
  - Tag and PPN compared after these accesses occurs



Translation Lookaside Buffer (TLB)



- The TLB is indexed by the Virtual Page Number.

- **The** data cache **is indexed by the Physical Address.**

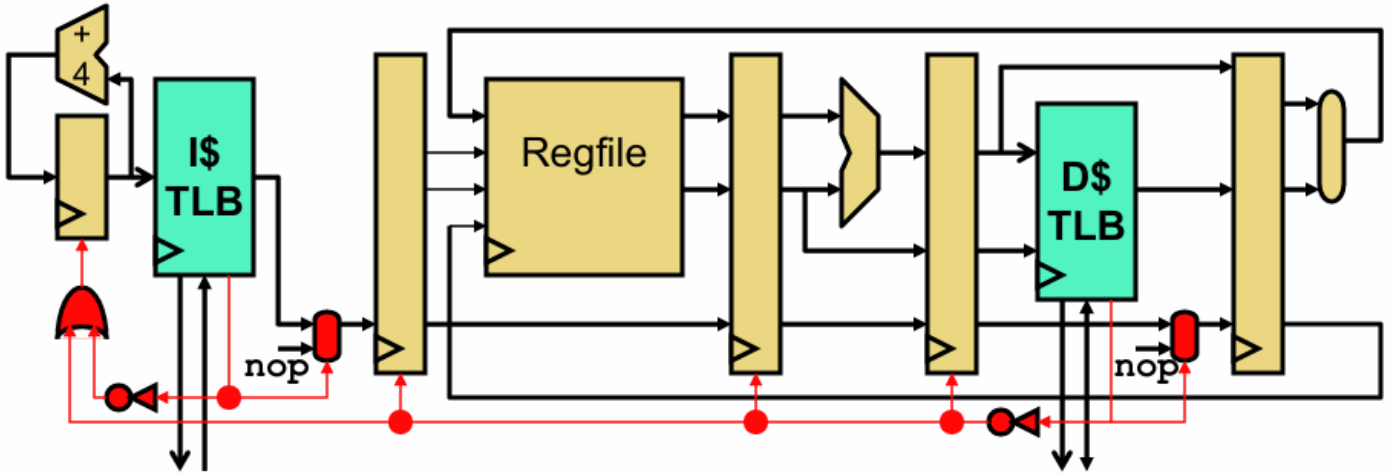
TLB Organization

- **Like caches**: TLBs also have ABCs
  - **Capacity**
  - **Associativity** (≥4-way associative, full-assoc common)
  - What does it mean for a TLB to have a **block size** of two?
    - Two VPs can a single tag
    - VPs must be aligned and consecutive
  - **Like caches**: there are second-level TLBs
- Example: AMD Opteron
  - 32-entry FA TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table
- TLB should **"cover"** size of on-chip caches
  - (#PTEs in TLB) * page_size ≥ cache_size
  - Why? Consider relative miss latency in each…

TLB Misses

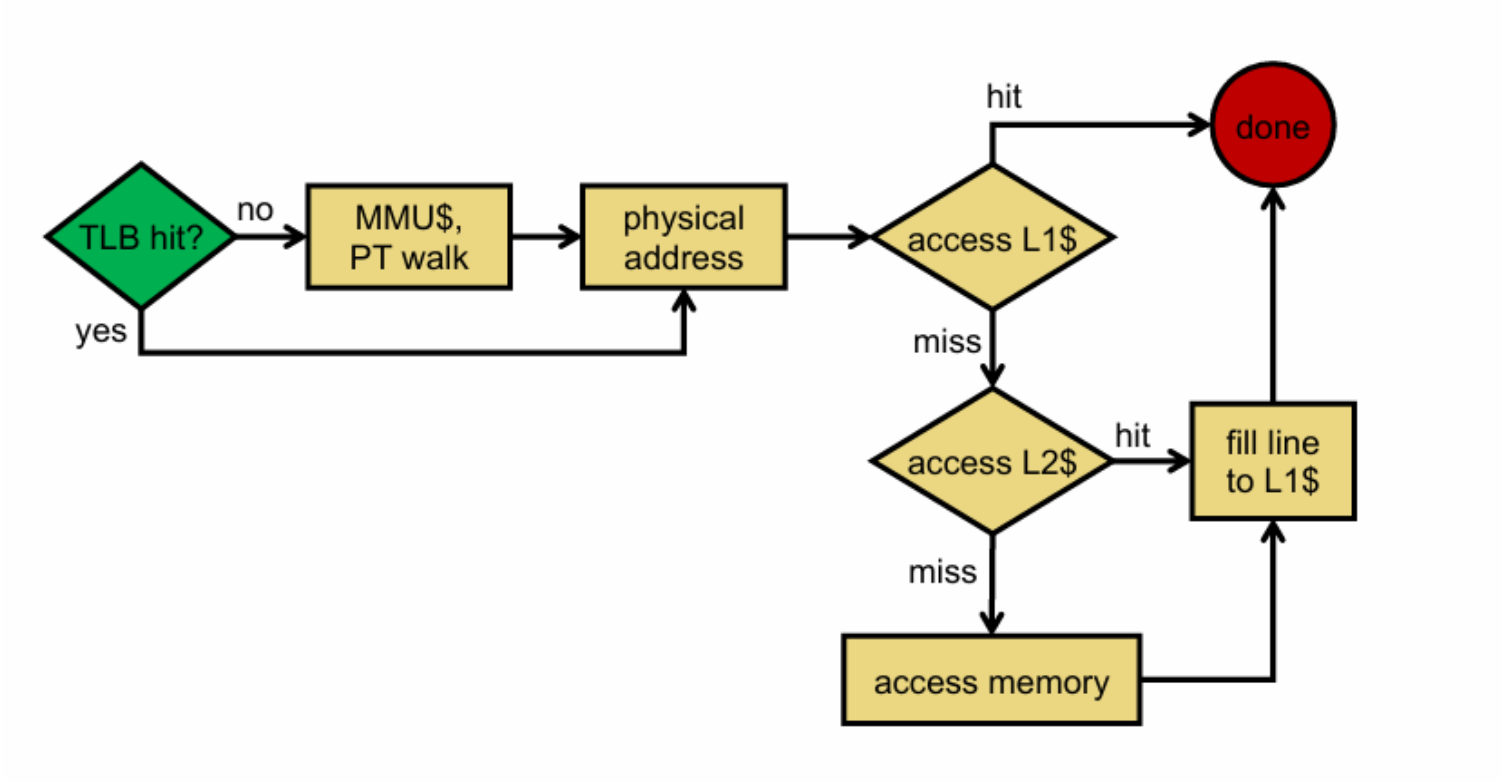
- **TLB miss:** translation not in TLB, but in page table
  - Two ways to "fill" it, both relatively fast
- **Hardware-managed TLB**: x86, ARM, RV
  - Page table root in hardware register, hardware "walks" table
  + Latency: saves cost of OS call (avoids pipeline flush)
  – Page table format is hard-coded
- **Software-managed TLB**: Alpha, MIPS
  - Short (~10 insn) OS routine walks page table, updates TLB
  + Keeps page table format flexible
  – Latency: 1-2 memory accesses + OS call (pipeline flush)

TLB Misses and Pipeline Stalls



- TLB misses stall pipeline just like data hazards...
  - ...if TLB is hardware-managed
- If TLB is software-managed...
  - ...must generate an interrupt
  - Hardware will not handle TLB miss

The Life of Virtual Memory Access



Page Sizes

- More ISAs support multiple page sizes
  - x86: 4KB, 2MB, 1GB
- larger pages have pros and cons
  - \+ reduce page table size
    - fewer entries needed to map a given amount of address space
  - \+ page table can be shallower
    - makes page table lookups faster
  - – “internal fragmentation” that wastes physical memory
    - allocate 2MB page but use only 5KB of it
  - – complex implementation
    - OS looks for opportunities to use large pages

Super Pages

- RISC-V super pages
  - Memory pages larger than the standard 4KiB base page
  - Reduce the number of page table mappings, translation overhead
  - Used to reduce TLB misses and speed up page table walk

Format	Leaf Page Table Level	Page Offset Bits	Page Size	Page Type
Sv32	0	12	4 KiB	kilopage (normal)
Sv32	1	12 + 10 = 22	4 MiB	**megapage**
Sv39	0	12	4 KiB	kilopage (normal)
Sv39	1	12 + 9 = 21	2 MiB	**megapage**
Sv39	2	12 + 9 × 2= 30	1 GiB	**gigapage**
Sv48	0	12	4 KiB	kilopage (normal)
Sv48	1	12 + 9 = 21	2 MiB	**megapage**
Sv48	2	12 + 9 × 2 = 30	1 GiB	**gigapage**
Sv48	3	12 + 9 × 3 = 39	512 GiB	**terapage**

Conclusion

- Virtual memory gives the illustration of “infinite” capacity
- A subset of virtual pages are located in physical memory
- A page table maps virtual pages to physical pages
  - Address translation
- A TLB speeds up address translation
- Multi-level page tables keep the page table size in check