ECE 250 / CPS 250
Computer Architecture

Main Memory and Virtual Memory

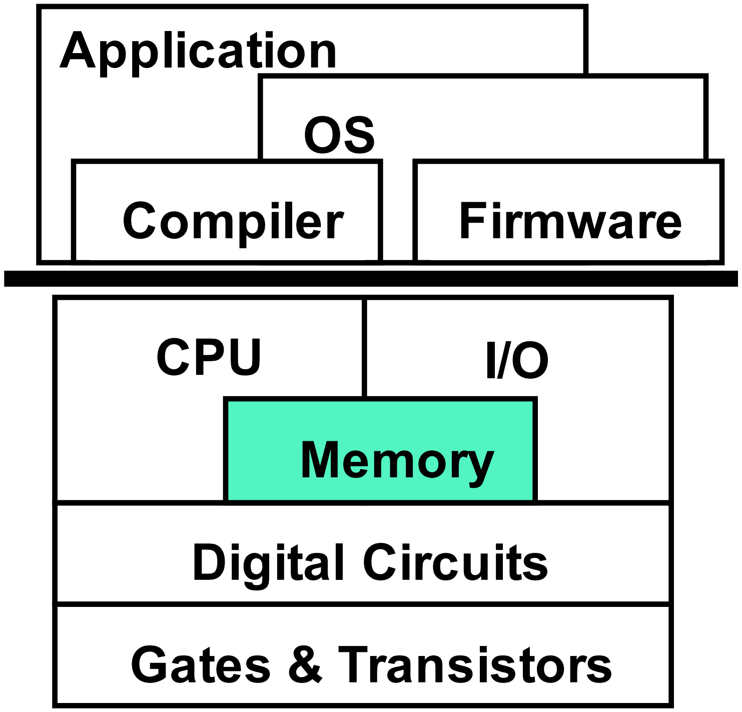
Benjamin Lee

Slides based on those from
Andrew Hilton (Duke), Alvy Lebeck (Duke)
Benjamin Lee (Duke), and Amir Roth (Penn)

Where We Are in This Course Right Now

- So far:
  - We know how to design a processor that can fetch, decode, and execute the instructions in an ISA
  - We understand how to design caches
- Now:
  - We learn how to implement main memory in DRAM
  - We learn about virtual memory
- Next:
  - We learn about the lowest level of storage (disks) and I/O

This Unit: Main Memory



- Building main memory out of DRAM
- Virtual memory
- Putting it all together

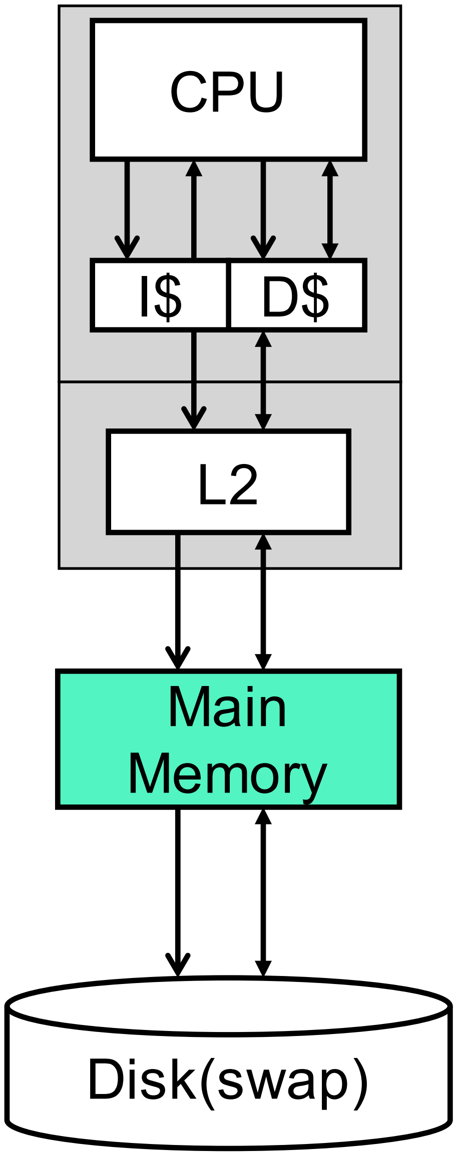
Readings

- Patterson and Hennessy
  - Still in Chapter 5

Two Issues

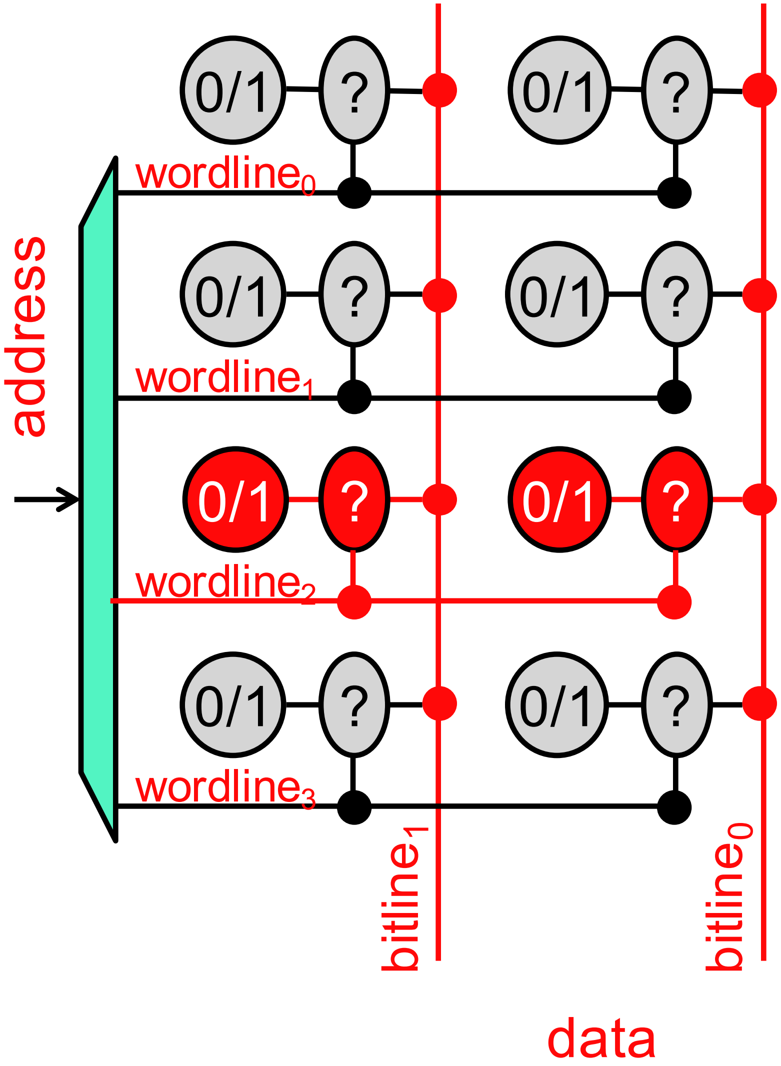
- How do we build main memory out of DRAM?
  - And what exactly is DRAM?
- How do we provide the illusion that each program sees 2^{32} bytes of address space (or 2^{64} on 64-bit machine)?
  - Foreshadowing: virtual memory

Main Memory Needs Big Capacity



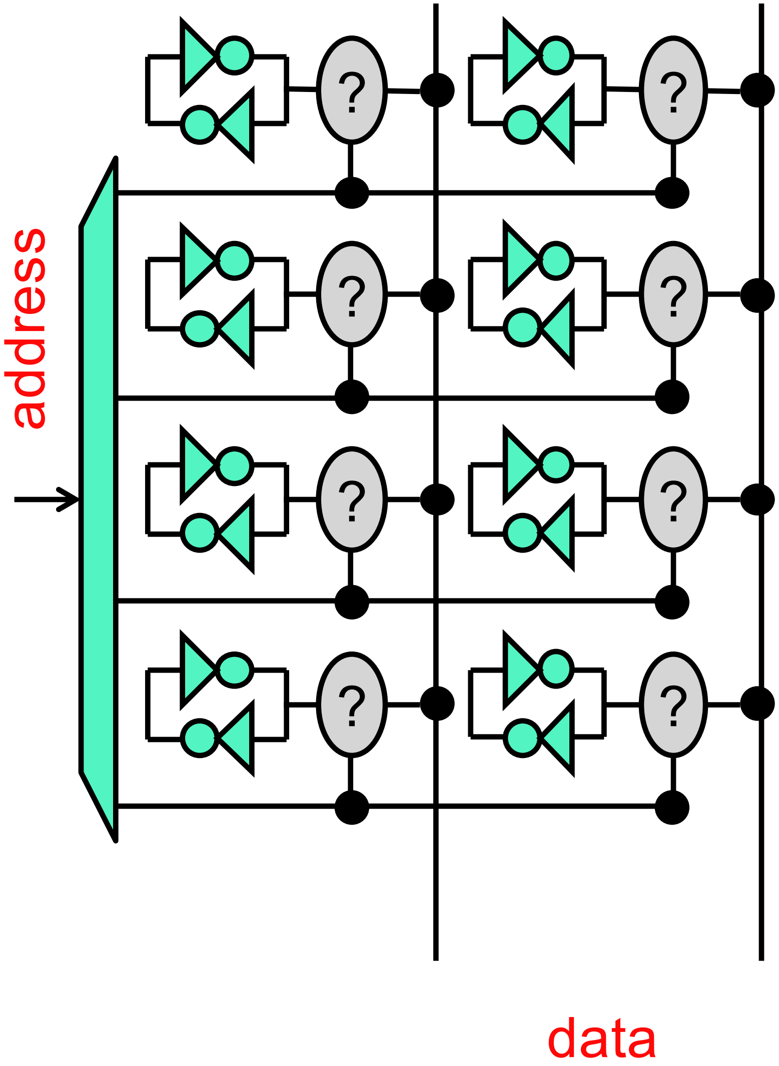
- Main memory is “bottom” of memory hierarchy
  - Caches are closer to CPU
  - Yeah, there’s disk below main memory, but ignore that for now
- Want memory to be very large → needs to be very dense
  - Isn’t SRAM dense? OK for caches
  - But there’s something even denser!

RAM in General (SRAM and DRAM)



- RAM: large storage arrays
- Basic structure
  - MxN array of bits (M N-bit words)
    - This one is 4x2
  - Bits in word connected by **wordline**
  - Bits in position connected by **bitline**
- Operation
  - Address decodes into M wordlines
  - Assert wordline → word on bitlines
  - Bit/bitline connection → read/write
- Access latency
  - #ports * √#bits

Remember Static RAM (SRAM)?

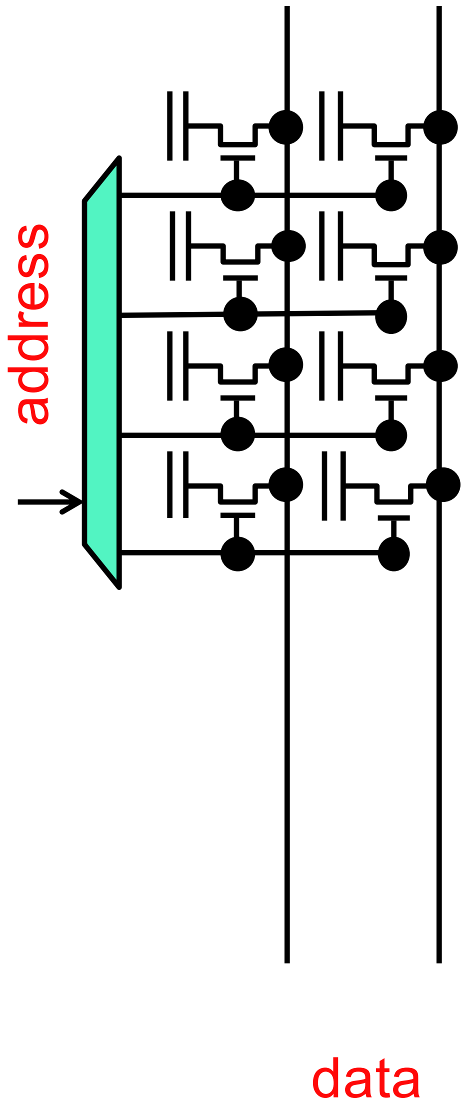


- **SRAM**: static RAM
  - Bits as cross-coupled inverters
  - Four transistors per bit
  - More transistors for ports

- **"Static"** means
  - Inverters connected to power/ground
  - Bits naturally/continuously "refreshed"
  - Bit values never decay

- Designed for speed

Dynamic RAM (DRAM)



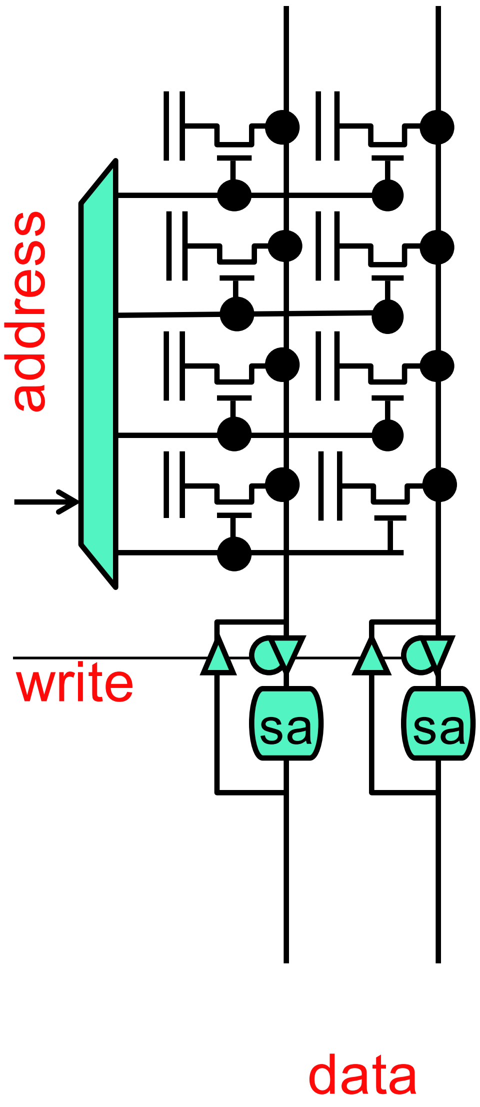
- **DRAM**: dynamic RAM
  - Bits as capacitors (if charge, bit=1)
  - "Pass transistors" as ports
  - One transistor per bit/port
- **"Dynamic"** means
  - Capacitors not connected to power/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed
- Designed for density

Moore's Law (DRAM chip capacity)

Year	Capacity	$/MB	Access time
1980	64Kb	$1500	250ns
1988	4Mb	$50	120ns
1996	64Mb	$10	60ns
2004	1Gb	$0.5	35ns
2008	2Gb	~$0.15	20ns
2013	8Gb	~$0	<10ns

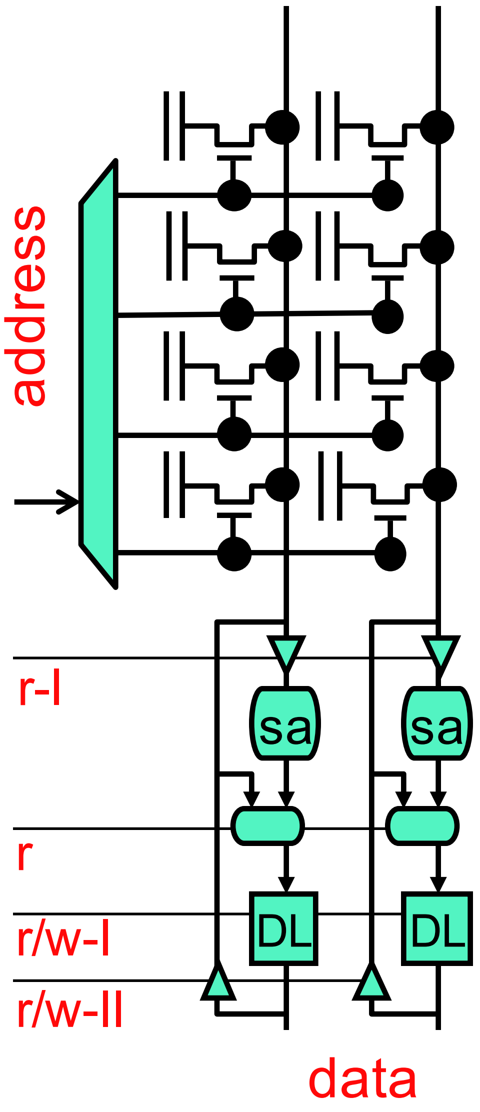
- Commodity DRAM parameters
  - 16X increase in capacity every 8 years = 2X every 2 years
    - Not quite 2X every 18 months (Moore's Law) but still close

DRAM Operation I



- Read: similar to SRAM read
  - Phase I: pre-charge bitlines
  - Phase II: decode address, enable wordline
    - Capacitor swings bitline voltage up (down)
    - Sense-amplifier interprets swing as 1 (0)
  - – **Destructive read**: word bits now discharged
    - Unlike SRAM
- Write: similar to SRAM write
  - Phase I: decode address, enable wordline
  - Phase II: enable bitlines
    - High bitlines charge corresponding capacitors
- – What about **leakage over time**?

DRAM Operation II

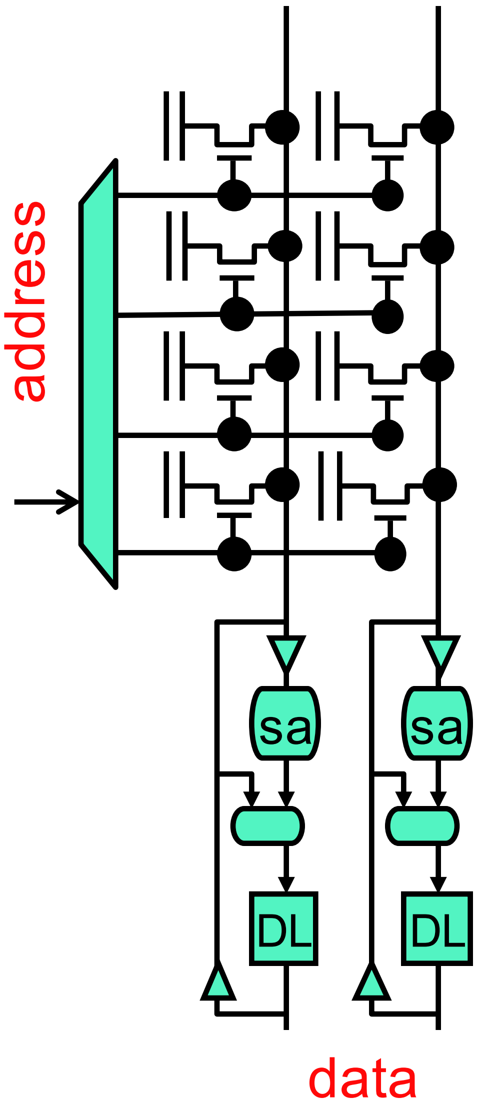


- Solution: add set of D-latches (**row buffer**)

- Read: two steps
  - Step I: read selected word into row buffer
  - Step IIA: read row buffer out to pins
  - Step IIB: write row buffer back to selected word
  - + Solves "destructive read" problem
- Write: two steps
  - Step IA: read selected word into row buffer
    - Deletes what was in that word before
  - Step IB: write data into row buffer
  - Step II: write row buffer back to selected word

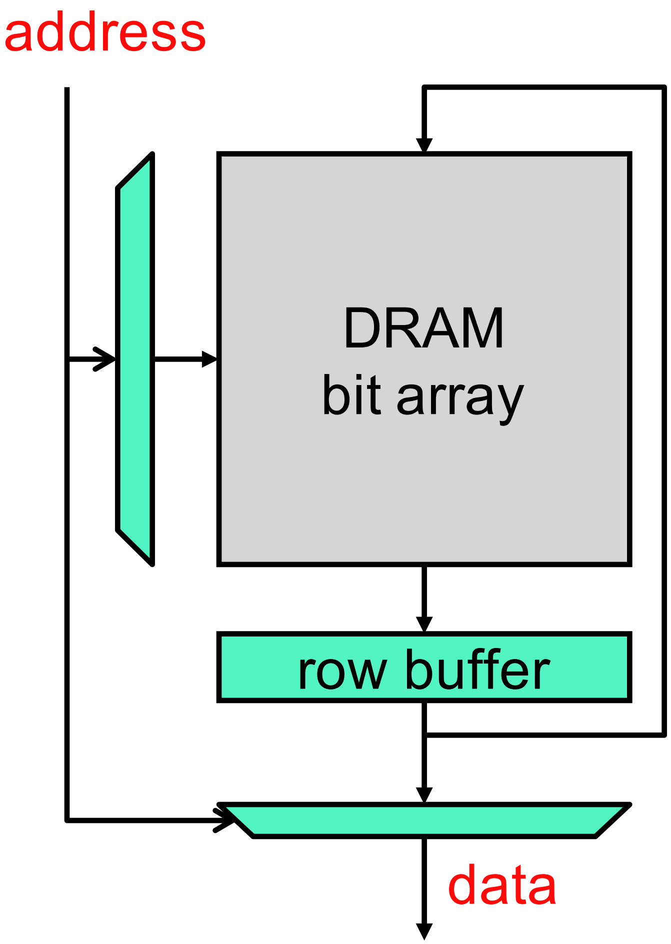
+ Also helps to solve leakage problem …

DRAM Refresh



- DRAM periodically refreshes all contents
  - Loops through all words
    - Reads word into row buffer
    - Writes row buffer back into DRAM array
  - 1–2% of DRAM time occupied by refresh

DRAM Parameters



- DRAM chip parameters
  - Large capacity: e.g., 8Gb
    - Arranged as square
    - + Minimizes wire length
    - + Maximizes refresh efficiency
  - Narrow data interface: 1–16 bit
    - Cheap packages → few bus pins
    - Pins are expensive

Access Time and Cycle Time

- DRAM access much slower than SRAM
  - More bits → longer wires
  - SRAM access latency: 2–3ns
  - DRAM access latency: 20-35ns

- DRAM cycle time also longer than access time
  - **Cycle time**: time between start of consecutive accesses
  - SRAM: cycle time = access time
    - Begin second access as soon as first access finishes
  - DRAM: cycle time = 2 * access time
    - Why? Can't begin new access while DRAM is refreshing row

Memory Access and Clock Frequency

- Computer's advertised **clock frequency** applies to CPU and caches
  - DRAM connects to processor chip via memory "bus"
  - Memory bus has its own clock, typically much slower

- Another reason why processor clock frequency isn't perfect performance metric
  - Clock frequency increases don't reduce memory or bus latency
  - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in (core) clock speed won't help at all

Brief History of DRAM

- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - Core more closely resembles today's disks than memory
    - "Core dump" is legacy terminology
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

Memoirs of a Computer Programmer, 1985

A Few Flavors of DRAM

- DRAM comes in several different varieties
  - Go to dell.com and see what kinds you can get for your laptop
- SDRAM = synchronous DRAM
  - Fast, clocked DRAM technology
  - Very common now
  - Several flavors: DDR, DDR2, DDR3, DDR4, etc.
- Graphics DRAM: GDDR

DRAM Packaging

- DIMM = dual inline memory module
  - E.g., 8 DRAM chips, each chip is 4 or 8 bits wide

DRAM: A Vast Topic

- Many flavors of DRAMs
  - DDR3 SDRAM, RDRAM, etc.
- Many ways to package them
  - SIMM, DIMM, FB-DIMM, etc.
- Many different parameters to characterize their timing
  - t_{RC}, t_{RAC}, t_{RCD}, t_{RAS}, etc.
- Many ways of using row buffer for "caching"

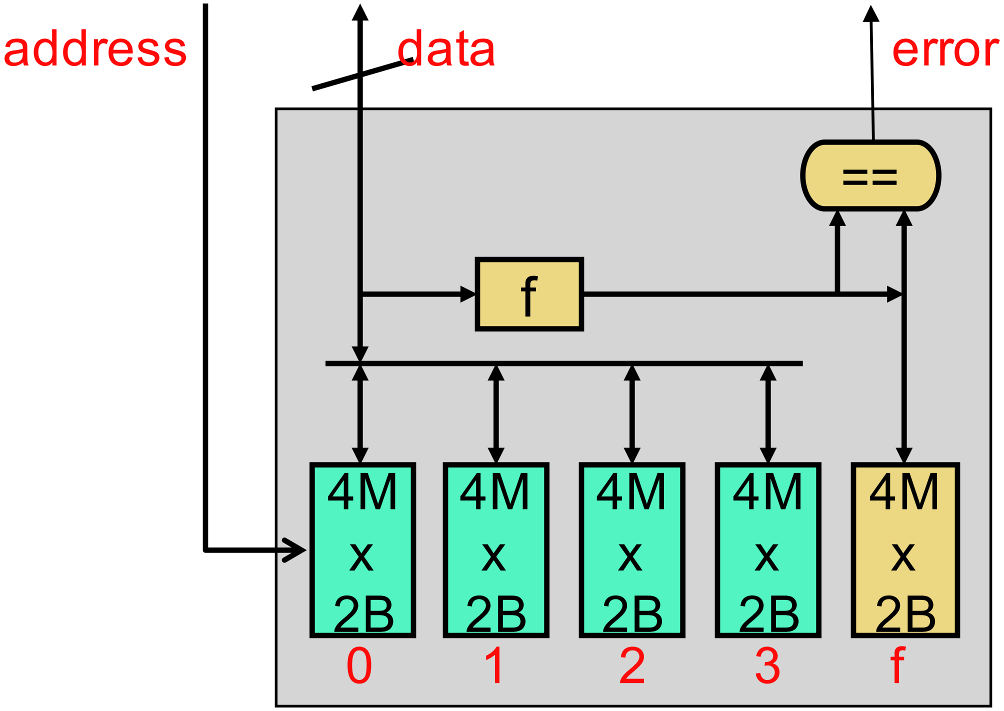
- There's at least one whole textbook on this topic!
  - And it has ~1K pages
- We could, but won't, spend rest of semester on DRAM

Error Detection and Correction

- One last thing about DRAM technology: **errors**
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic α-particle strikes
    - Many more bits
  - Modern DRAM systems: built-in error detection/correction

- **Key idea: checksum-style redundancy**
  - Main DRAM chips store data, additional chips store f(data)
    - |f(data)| < |data|
  - On read: re-compute f(data), compare with stored f(data)
    - Different ? Error…
  - Option I (**detect**): kill program
  - Option II (**correct**): enough information to fix error? fix and go on

Error Detection and Correction



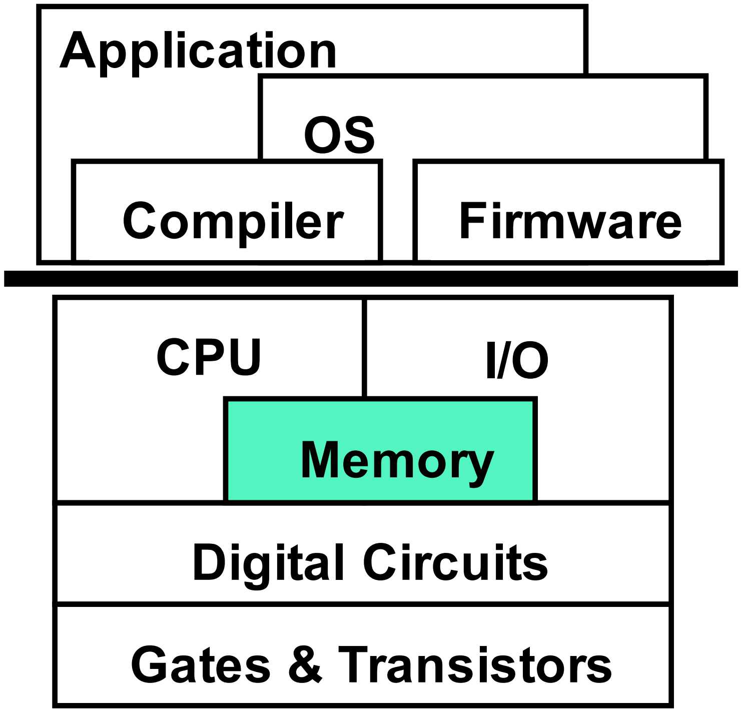
- Error detection/correction schemes distinguished by…
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection Example: Parity

- **Parity**: simplest scheme
  - f($\text{data}_{N-1\ldots0}$) = XOR(data_{N-1}, …, data_1, data_0)
  - + Single-error detect: detects a single bit flip (common case)
    - Will miss two simultaneous bit flips…
    - But what are the odds of that happening?
  - – Zero-error correct: no way to tell which bit flipped

– Many other schemes exist for detecting/correcting errors

This Unit: Main Memory



- Building main memory out of DRAM
- Virtual memory
- Putting it all together

Problem: Physical Memory Constraints

- What is a “reasonable” memory capacity? 4 – 64GB?
  - 32-bit address space: 4 GB per program
  - 64-bit address space: 16-Billion GB per program

- Virtual Memory
  - Give every program the **illusion** of entire address space
  - Hardware and Operating System move data transparently

Virtual Memory

- Idea of treating memory (DRAM) like a cache
  - Contents are a dynamic subset of program's address space
  - Dynamic data management is transparent to program
- Actually predates "caches" (by a little)

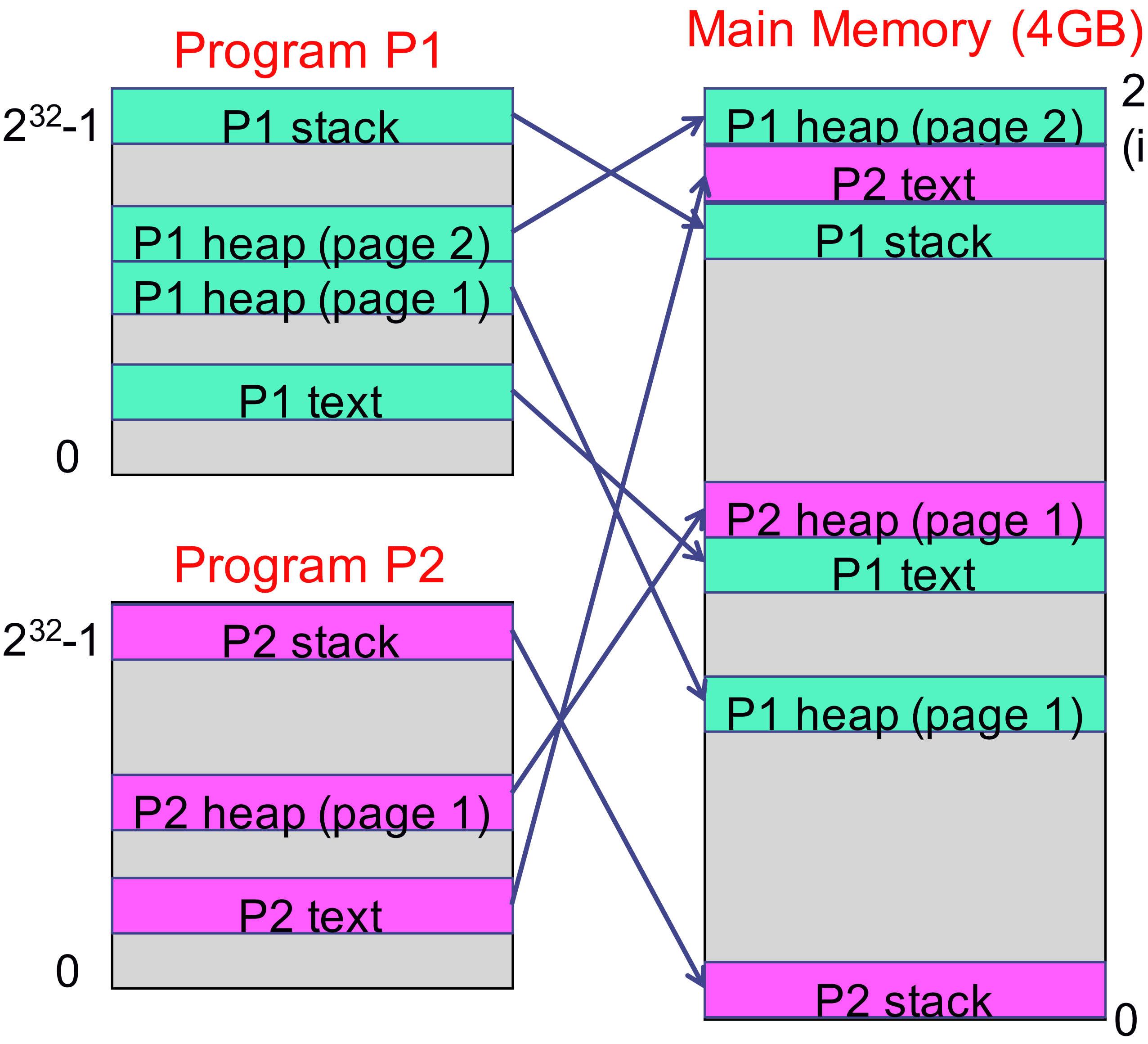
- Original motivation: **compatibility**
  - IBM System 370: a family of computers with one software suite

  + Same program could run on machines with different memory sizes
    - Caching mechanism made it appear as if memory was 2^N bytes
    - Regardless of how much memory there actually was

  – Prior, programmers explicitly accounted for memory size

- **Virtual memory**
  - Virtual: "in effect, but not in actuality" (i.e., appears to be, but isn't)

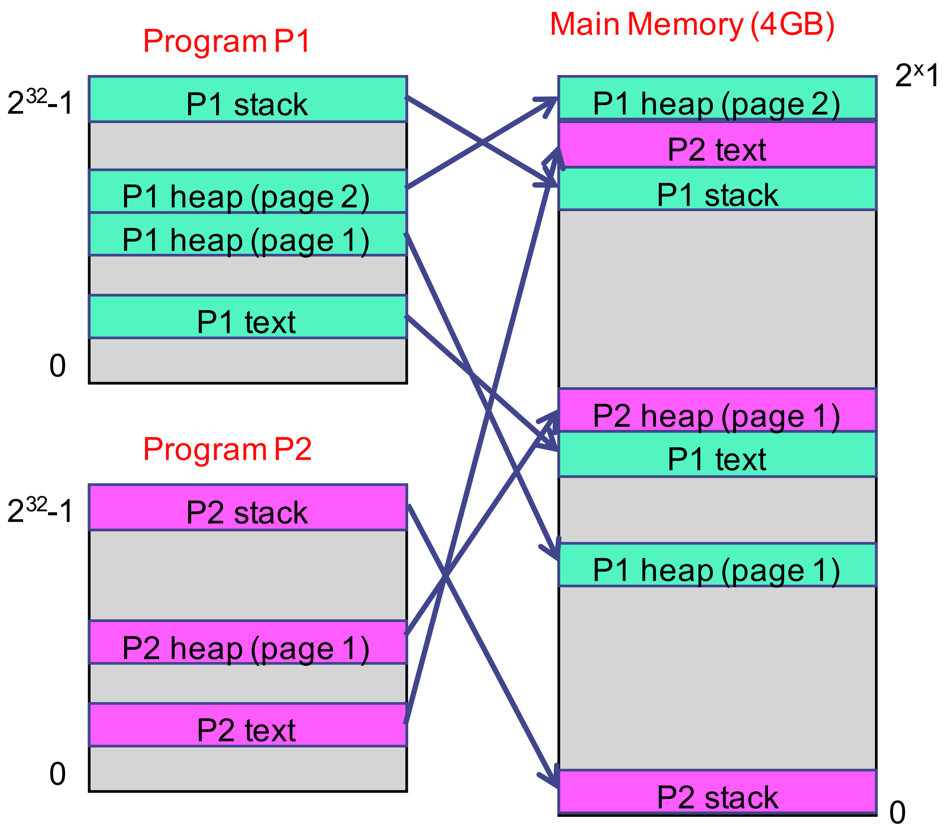
Map From Virtual to Physical Address



2^X-1 = size of phys memory (i.e., how much DRAM)

- Programs use **virtual addresses (VA)**
- Hardware and OS map virtual addresses to **physical addresses (PA)**

Map From Virtual to Physical Address

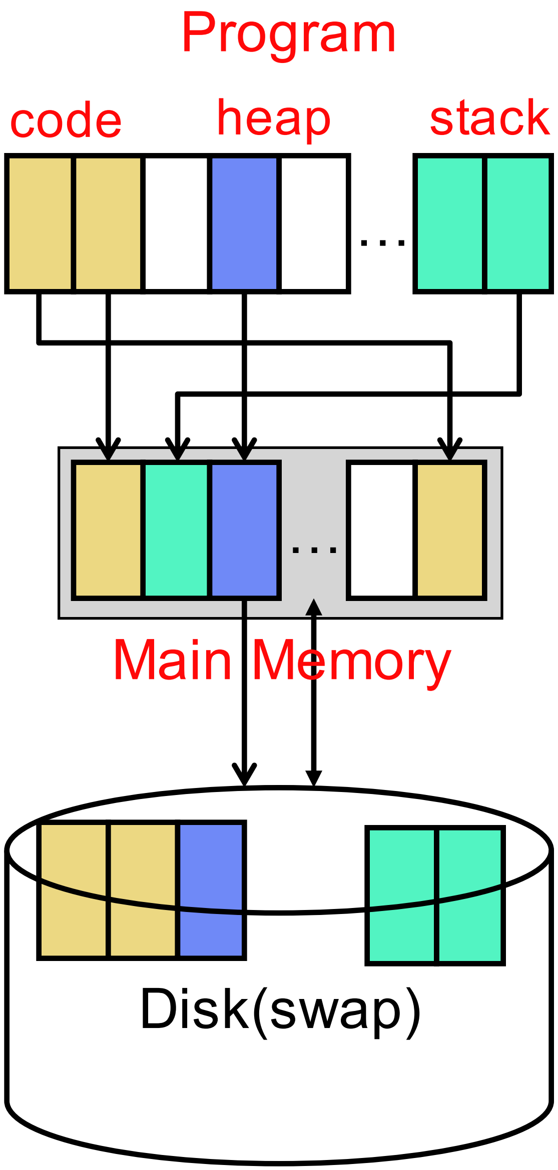


- Programs issue loads, stores, and instr fetches to virtual addresses
- Under the hood, hardware/OS translate VAs to PAs
- Translation maps {VA, Process ID}→ PA
  - Programs P1 and P2 can each have own address 217 → different physical addresses
- Translation done at granularity of a **page** (typically 8KB-64KB)
  - From virtual page number (VPN) to physical page number (PPN) {VPN, PID} → PPN

Physical << Virtual Memory

- What if programs use a total amount of virtual memory that is more than the amount of physical memory?
- Physical memory is cache
  - Subset of all virtual memory used by all running programs
- Miss in physical memory is a **page fault**
- Last level of memory hierarchy is disk
  - Everything fits on disk – no misses
  - OS manages what data lives in physical memory and what data lives on disk (reminder: take CS 330)
  - If page fault, OS brings in desired page from disk and makes room by replacing a page from memory and moving it to disk

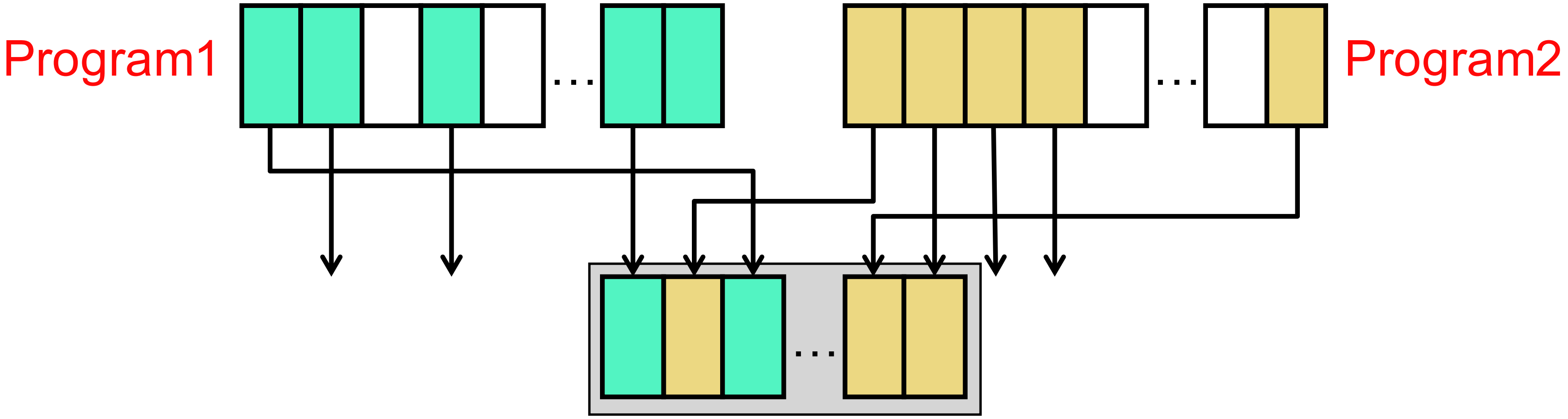
Virtual Memory



- Programs use **virtual addresses (VA)**
  - $0 \ldots 2^N - 1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Itanium is 64-bit
- Memory uses **physical addresses (PA)**
  - $0 \ldots 2^M - 1$ ($M < N$, especially if $N = 64$)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

Virtual Memory

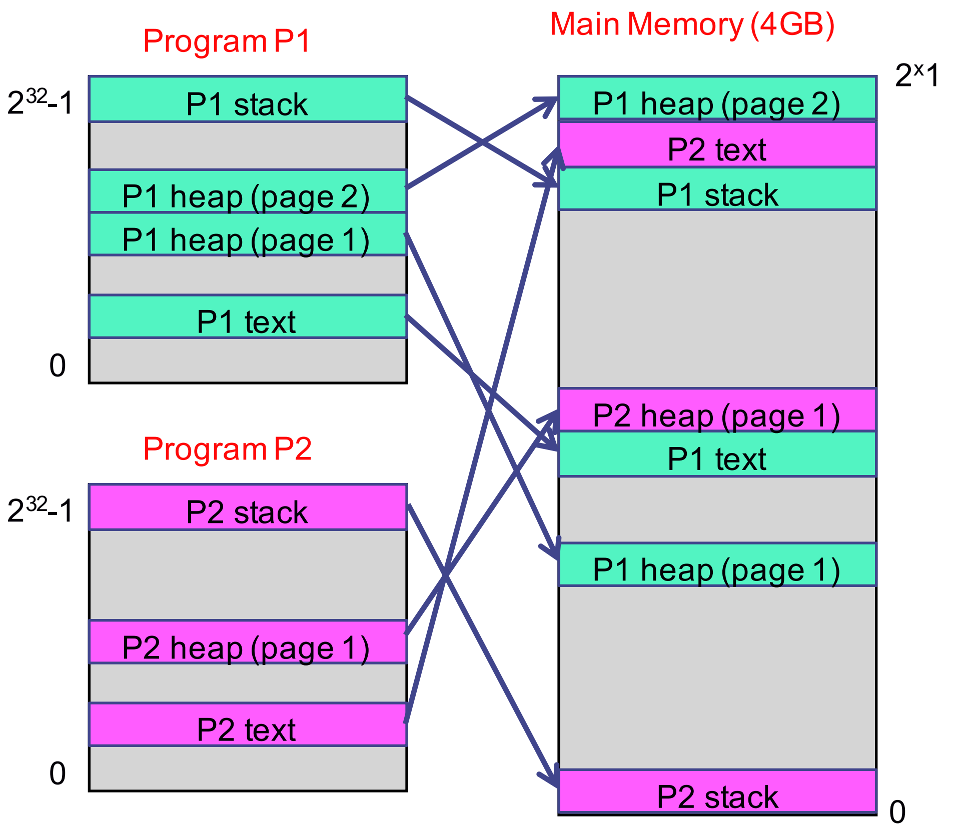
- Virtual memory is quite useful
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - "System" maps VPs from different processes to different PPs
    - + Prevents processes from reading/writing each other's memory



Still More Uses of Virtual Memory

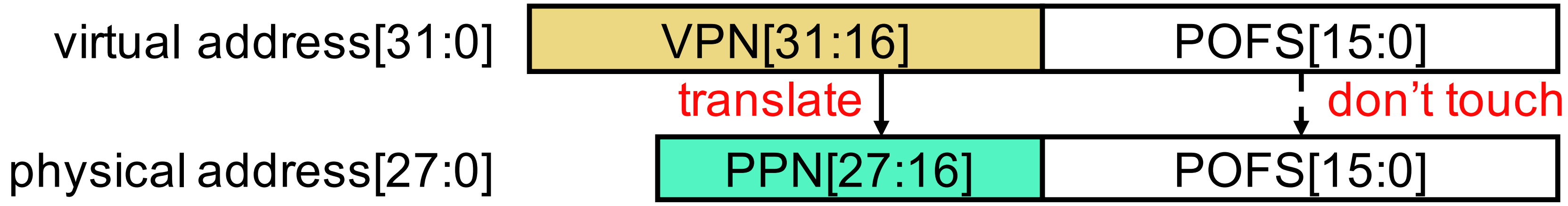
- Inter-process communication
  - Map VPs in different processes to same PPs
- Direct memory access I/O
  - Think of I/O device as another process
  - Will talk more about I/O in a few lectures
- **Protection**
  - Piggy-back mechanism to implement page-level protection
  - Map VP to PP … and RWX protection bits
  - Attempt to execute data, or attempt to write insn/read-only data?
    - Exception → OS terminates program

Map From Virtual to Physical Address



- Hardware/OS translate VAs to PAs
- Translation maps {VA, Process ID}→ PA
  - Programs P1 and P2 can each have own address (e.g., 0x217) → different physical addresses
- Translation done at granularity of a **page** (typically 8KB-64KB)
  - From virtual page number (VPN) to physical page number (PPN) {VPN, PID} → PPN

Address Translation

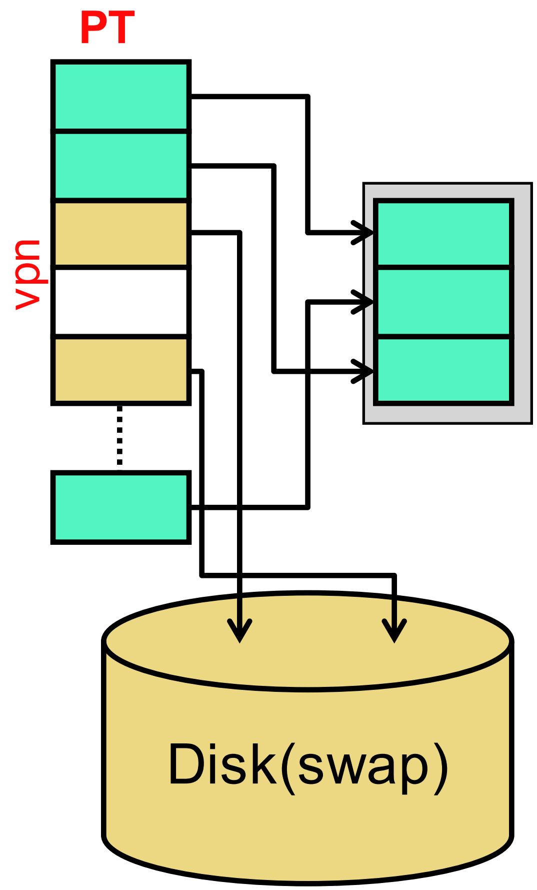


- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** and page offset (POFS)
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated – why not?
  - VA→PA = [VPN, POFS] → [PPN, POFS]

- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN (16 = 32 – 16)
  - Maximum 256MB physical memory → 28-bit PA → 12-bit PPN

Mechanics of Address Translation

- How are addresses translated?
  - In software (now) but with hardware acceleration (a little later)
- Each process is allocated a **page table (PT)**
  - PT is data structure managed by OS
  - Maps VPNs to PPNs or disk addresses
    - VPN entries empty if page never referenced
  - Translation accesses data structure



- Simplest PT design
  - Table with one entry for every VPN

Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs), 32-bit machine, 4KB pages
- Solution
  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB page table

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?

- Page tables can get enormous
- There are ways of making them smaller

Multi-Level Page Table

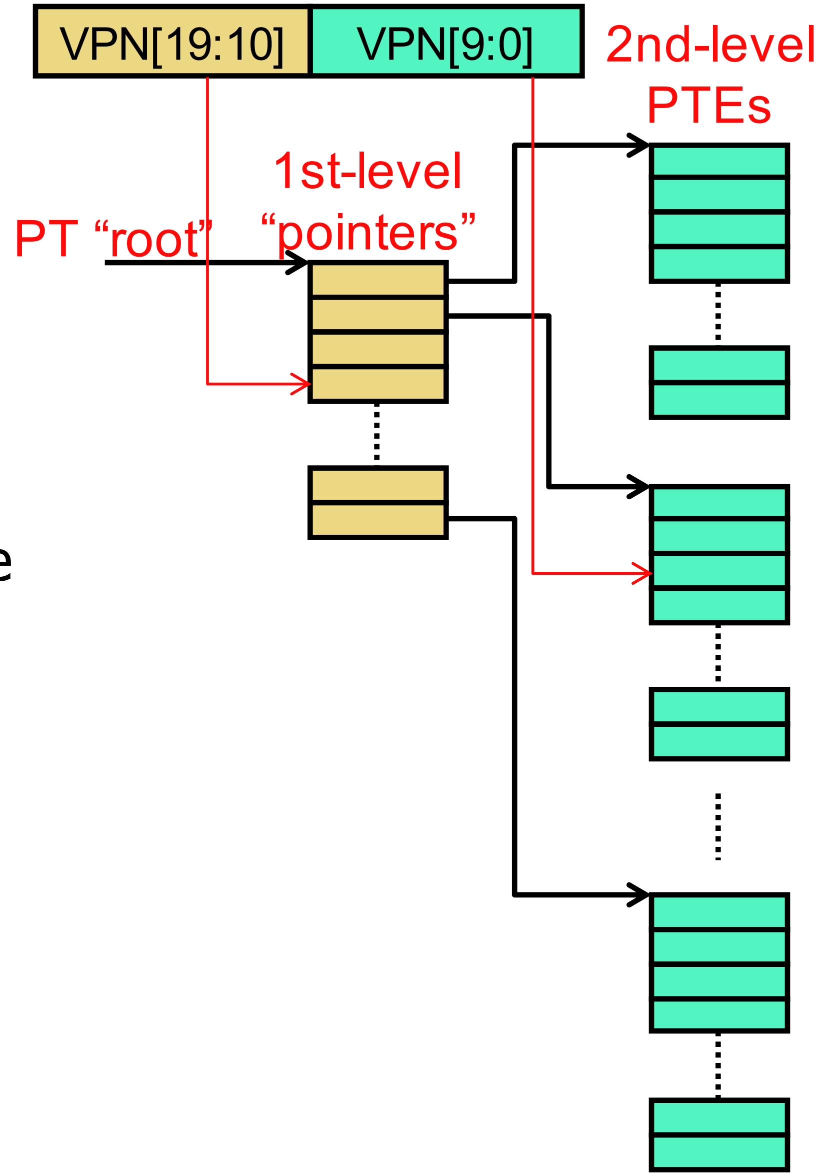
- Tree of Page Tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels

- Example: two-level page table for machine on last slide
  - Compute number of pages needed for lowest-level (PTEs)
  - PTEs reside in memory pages
    - 4KB pages / 4B PTEs → 1K PTEs fit on a single page
  - Each VP requires a PTE
    - 1M PTEs / (1K PTEs/page) → 1K pages to hold PTEs

  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers * 32-bit VA → 4KB → 1 upper level page

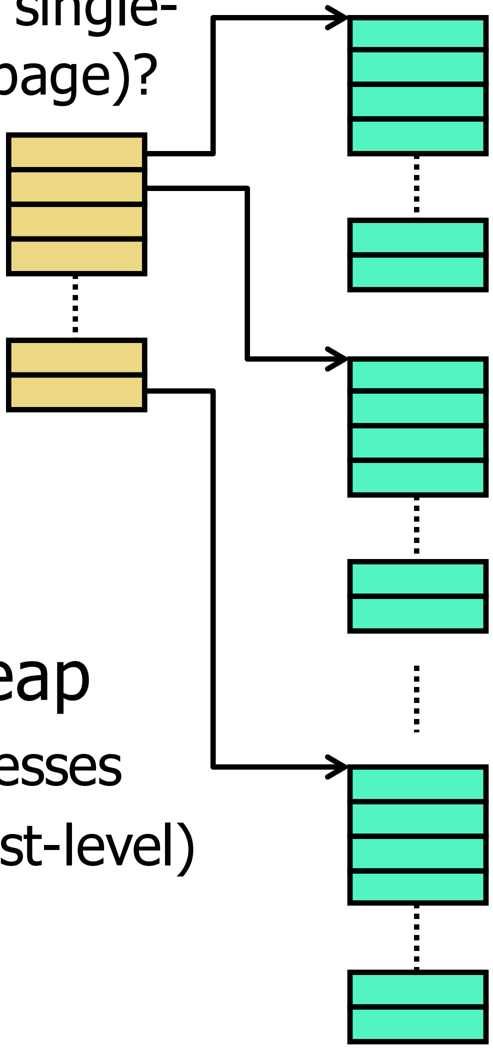
Multi-Level Page Table

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table



Multi-Level Page Table

- Have we saved any space?
  - Isn’t total size of 2nd level PTE pages same as single-level table (i.e., 4MB = 1K pointers* 4KB PTE page)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level pages need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level page maps 4MB of virtual addresses
  - 1 page for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages for PT = 28KB (<< 4MB)



Address Translation Mechanics

- The six questions
  - What? address translation
  - Why? compatibility, multi-programming, protection
  - How? page table
  - **Who performs it?**
  - **When?**
  - **Where does page table reside?**

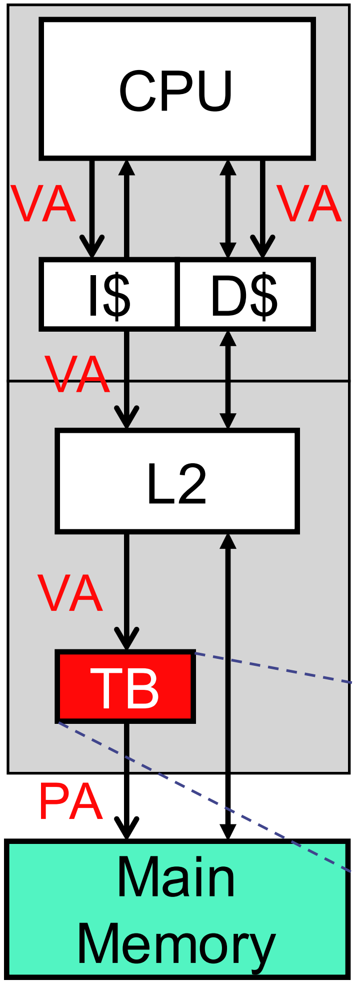
- Option I: process (program) translates its own addresses
  - Page table resides in process visible virtual address space
  - Bad idea: implies that program (and programmer)…
    - …must know about physical addresses
      - Isn't that what virtual memory is designed to avoid?
    - …can forge physical addresses and mess with other programs
  - Translation on L2 miss or always? How would program know?

Who? Where? When? Take II

- Option II: **operating system (OS)** translates for process
  - Page table resides in OS virtual address space
  - + User-level processes cannot view/modify their own tables
  - + User-level processes need not know about physical addresses
  - Translation on L2 miss
    - – Otherwise, OS SYSCALL before any fetch, load, or store

- L2 miss: interrupt transfers control to OS handler
  - Handler translates VA by accessing process's page table
  - Accesses memory using PA
  - Returns to user process when L2 fill completes
  - – Still slow: added interrupt handler and PT lookup to memory access
  - – What if PT lookup itself requires memory access? Head spinning…

Translation Buffer

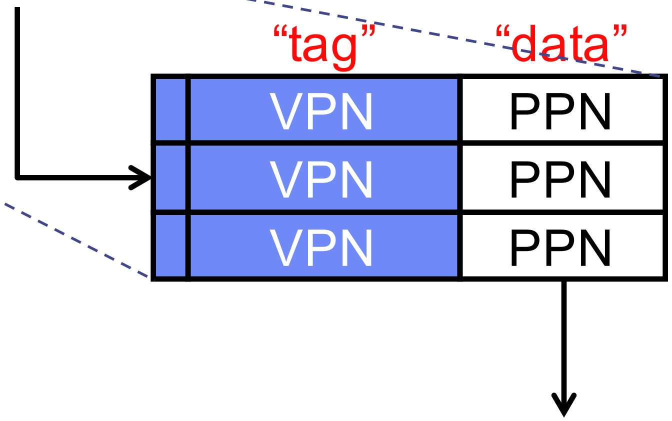


- Functionality problem? Add indirection!
- Performance problem? Add cache!

- Address translation too slow?
  - Cache translations in **translation buffer (TB)**
    - Small cache: 16–64 entries, often fully assoc

  + Exploits temporal locality in PT accesses

  + OS handler only on TB miss



TB Misses

- **TB miss:** requested PTE not in TB, but in PT
  - Two ways of handling

- **1) OS routine**: reads PT, loads entry into TB (e.g., Alpha)
  - Privileged instructions in ISA for accessing TB directly
  - Latency: one or two memory accesses + OS call

- **2) Hardware FSM**: does same thing (e.g., IA-32)
  - Store PT root pointer in hardware register
  - Make PT root and 1st-level table pointers physical addresses
    - So FSM doesn't have to translate them

  + Latency: saves cost of OS call

Nested TB Misses

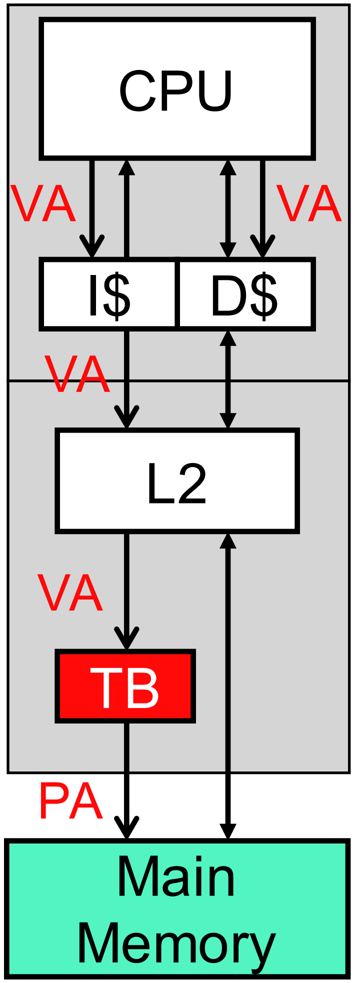
- **Nested TB miss**: when OS handler itself has a TB miss
  - TB miss on handler instructions
  - TB miss on page table VAs
  - Not a problem for hardware FSM: no instructions, PAs in page table

- Handling is tricky for SW handler, but possible
  - First, save current TB miss info before accessing page table
    - So that nested TB miss info doesn't overwrite it
  - Second, **lock nested miss entries into TB**
    - Prevent TB conflicts that result in infinite loop
    - Another good reason to have a highly-associative TB

Page Faults

- **Page fault**: PTE not in TB or in PT
  - Page is simply not in memory
  - Starts out as a TB miss, detected by OS handler/hardware FSM
- **OS routine**
  - OS software chooses a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk (like dirty cache block with writeback $)
  - Read missing page from disk (done by OS)
    - Takes so long (10ms), OS schedules another task
  - Treat like a normal TB miss from here

Virtual Caches

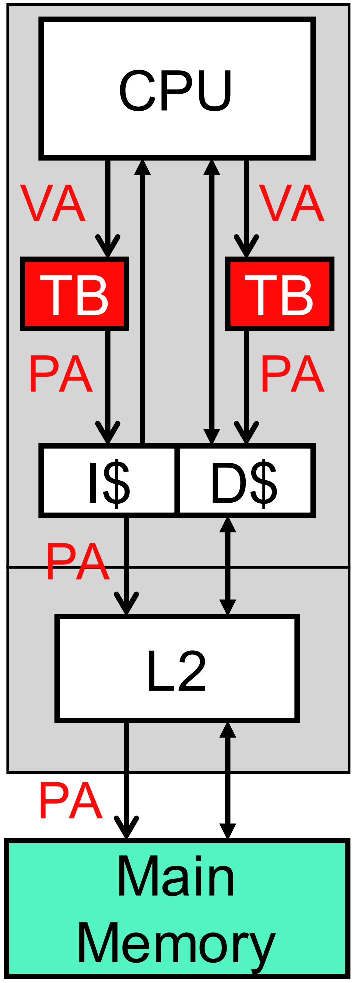


- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - + Fast: avoids translation latency in common case

- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags

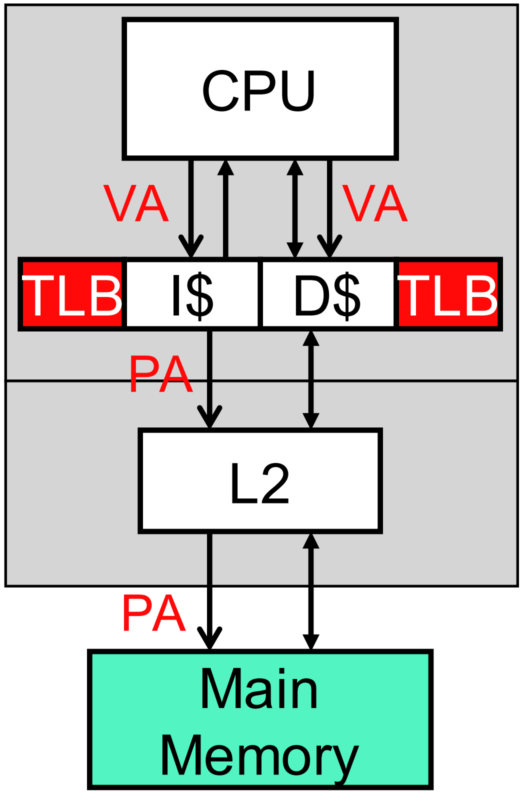
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - How are multiple cache copies kept in sync?
    - Also a problem for I/O (later in course)
  - Disallow caching of shared memory? Slow

Physical Caches



- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  - + No need to flush caches on process switches
    - Processes do not share PAs
  - + Cached inter-process communication works
    - Single copy indexed by PA
  - – Slow: adds 1 cycle to t_{hit}

Virtual Physical Caches



Compromise: **virtual-physical caches**

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel

+ No context-switching/aliasing problems

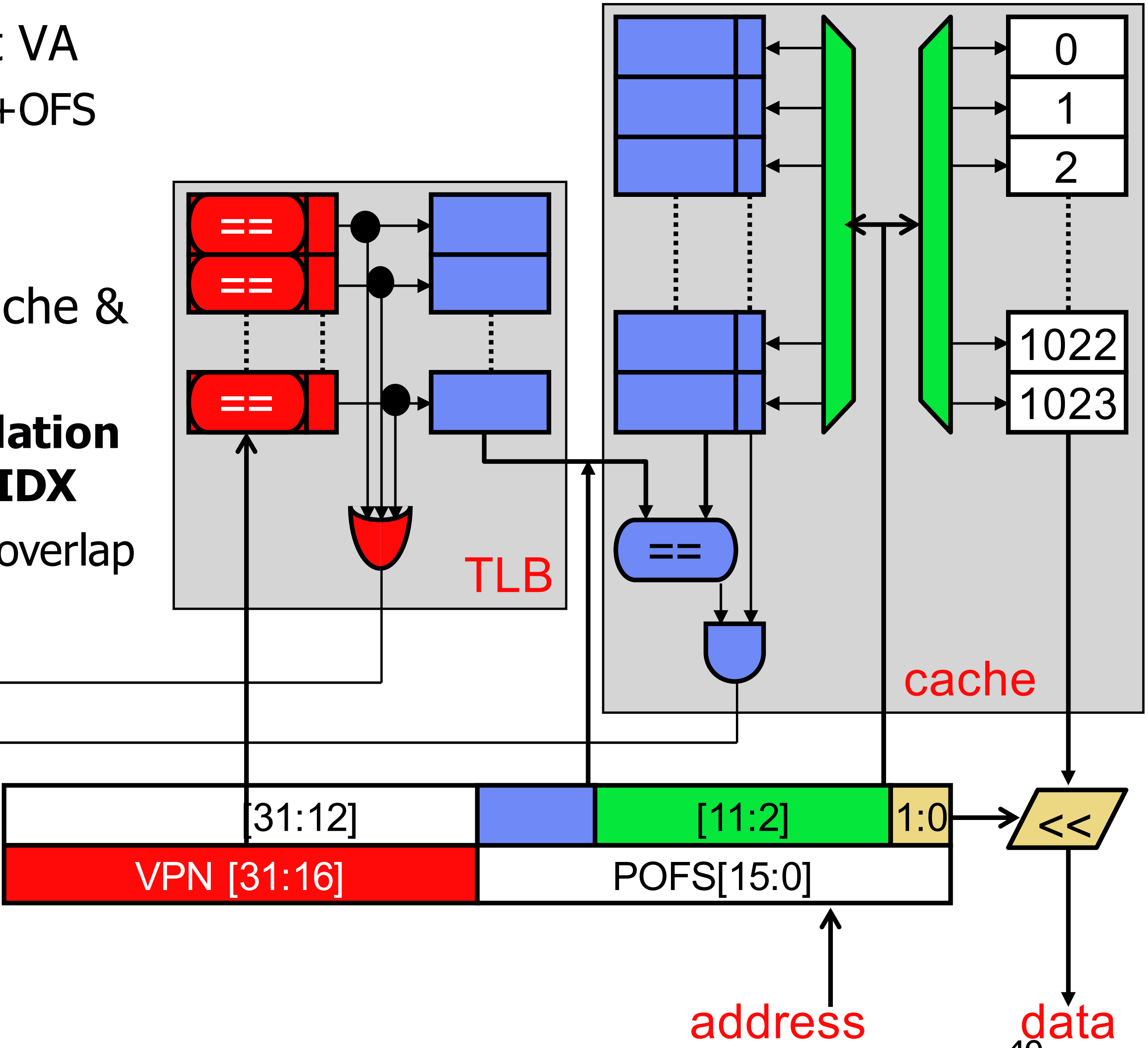
+ Fast: no additional t_{hit} cycles

- A TB that acts in parallel with a cache is a **TLB**
  - **Translation Lookaside Buffer**

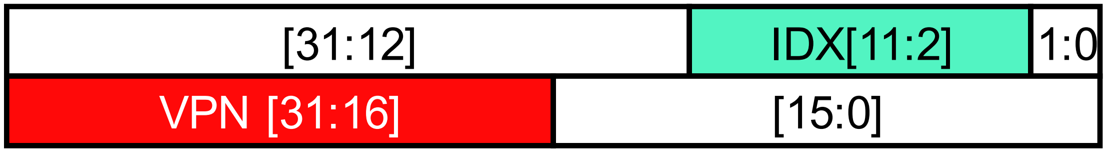
- Common organization in processors today

Cache/TLB Access

- Two ways to look at VA
  - Cache: TAG+**IDX**+OFS
  - TLB: **VPN**+POFS
- Can have parallel cache & TLB …
  - **If address translation doesn't change IDX**
  - → VPN/IDX don't overlap



Cache Size And Page Size



[31:12] IDX[11:2] 1:0

VPN [31:16] [15:0]

- Relationship between page size and L1 I$(D$) size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - I$(D$) size / **associativity** ≤ page size
  - Big caches must be set associative
    - Big cache → more index bits (fewer tag bits)
    - More set associative → fewer index bits (more tag bits)
  - Systems are moving towards bigger (64KB) pages
    - To amortize disk latency
    - To accommodate bigger caches

TLB Organization

- **Like caches**: TLBs also have ABCs
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPNs share a single tag
- **Rule of thumb**: TLB should "cover" L2 contents
  - In other words: #PTEs * page size ≥ L2 size
  - Why? Think about this …

Flavors of Virtual Memory

- Virtual memory almost ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even some embedded (in non-computer) processors support it

- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - IA32 uses this
    - Makes life very difficult
  - **Paged segments**: don't ask

Summary

- DRAM
  - Two-level addressing
  - Refresh, access time, cycle time
- Error correction
- Memory organization
- Virtual memory
  - Page tables and address translation
  - Page faults and handling
  - Virtual, physical, and virtual-physical caches and TLBs

Next part of course: I/O