ECE 152 / 496
Introduction to Computer Architecture

Main Memory and Virtual Memory

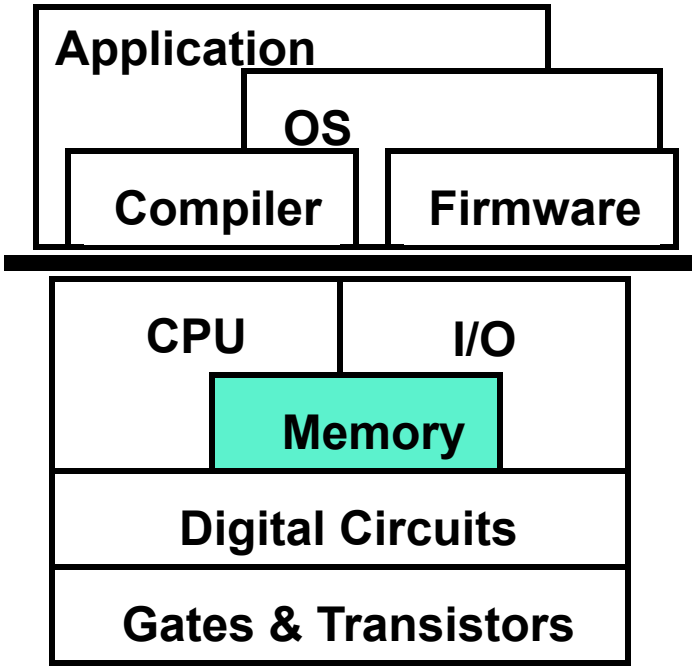
Benjamin C. Lee
Duke University

Slides from Daniel Sorin (Duke)
and are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)

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 can pipeline this processor
  - 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



- Memory hierarchy review
- DRAM technology
  - A few more transistors
  - Organization: two-level addressing
- Building a memory system
  - Bandwidth matching
  - Error correction
- Organizing a memory system
- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

Readings

- Patterson and Hennessy
  - Still in Chapter 5

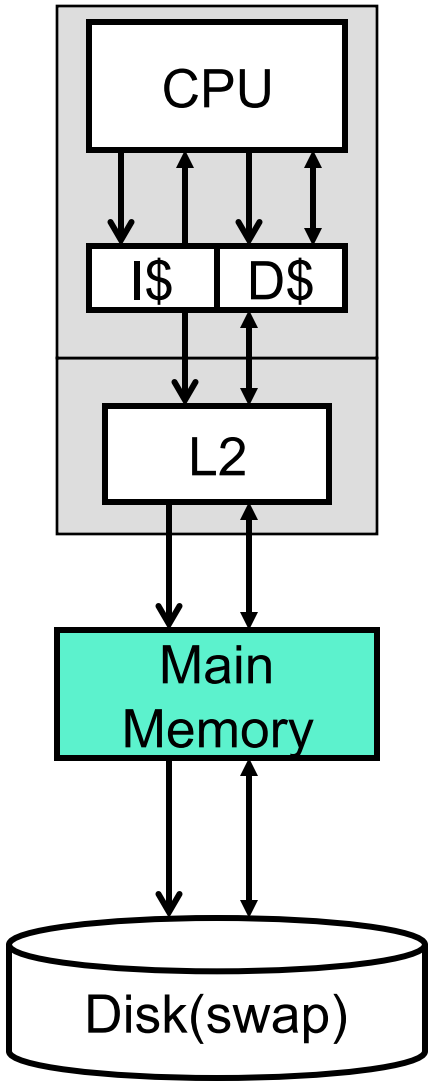
Memory Hierarchy Review

- Storage: registers, **memory**, disk
  - Memory is fundamental element (unlike caches or disk)

- Memory component performance
  - $\mathbf{t_{avg} = t_{hit} + \%_{miss} * t_{miss}}$
  - Can't get both low t_{hit} and $\%_{miss}$ in a single structure

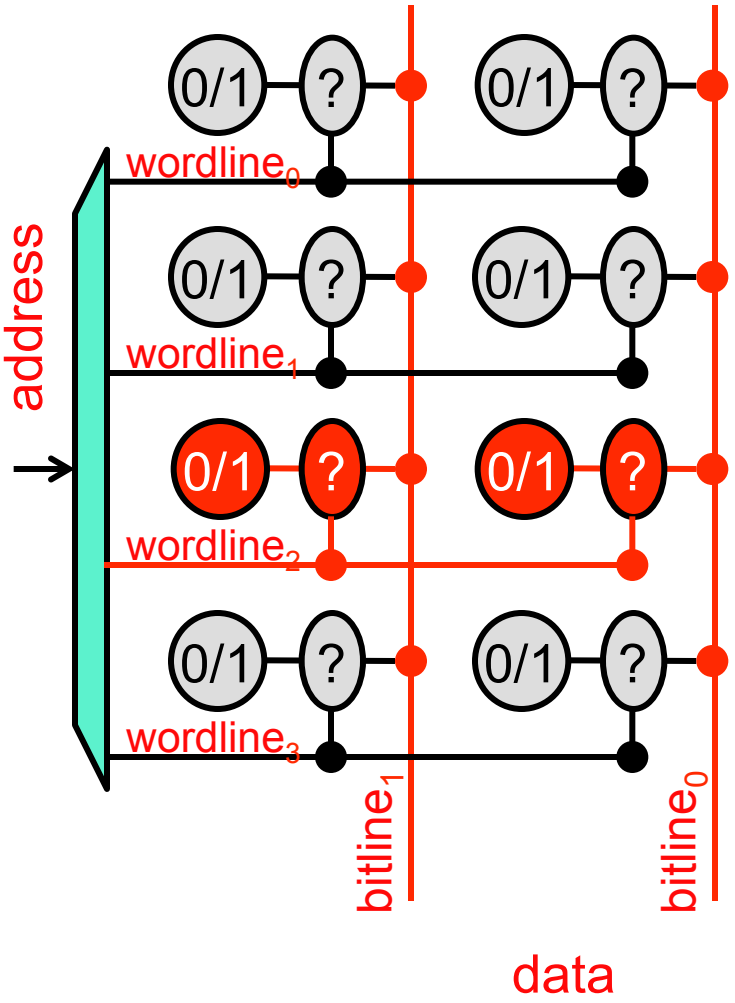
- Memory hierarchy
  - Upper components: small, fast, expensive
  - Lower components: big, slow, cheap
  - t_{avg} of hierarchy is close to t_{hit} of upper (fastest) component
    - 10/90 rule: 90% of stuff found in fastest component
  - **Temporal/spatial locality**: automatic up-down data movement

Concrete Memory Hierarchy



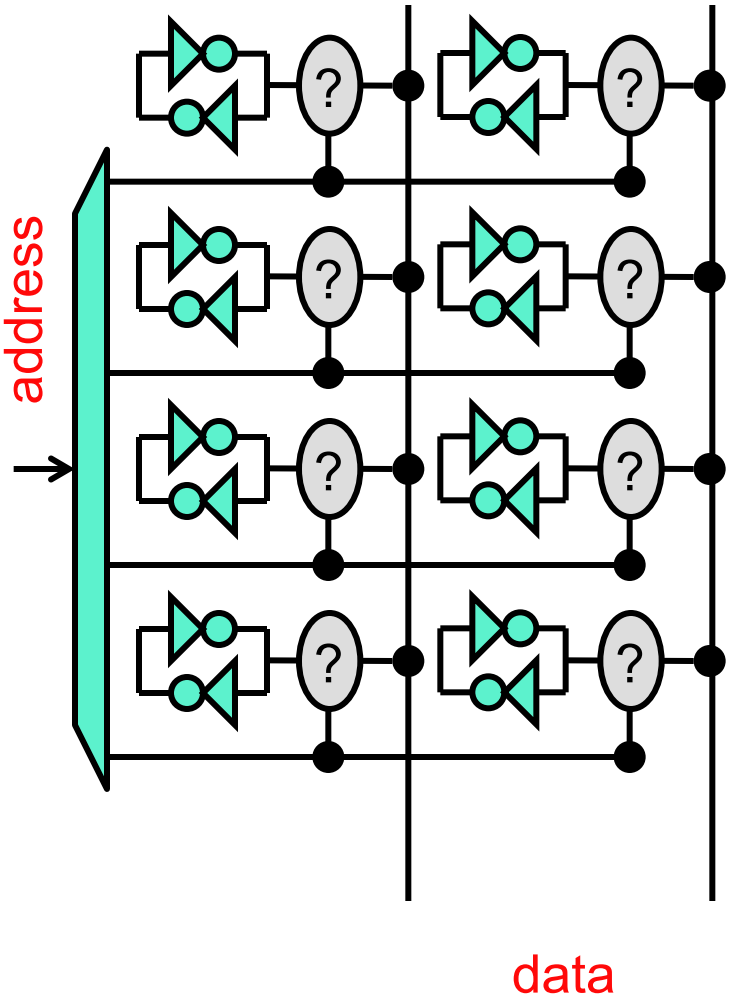
- 1st/2nd(/3rd) levels: caches (L1 I$, L1 D$, L2)
  - Made of SRAM
  - Managed in hardware
  - Previous unit of course
- Below caches level: **main memory**
  - Made of DRAM
  - Managed in software
  - This unit of course
- Below memory: disk (swap space)
  - Made of magnetic iron oxide disks
  - Managed in software
  - Next unit

RAM in General (SRAM and DRAM)



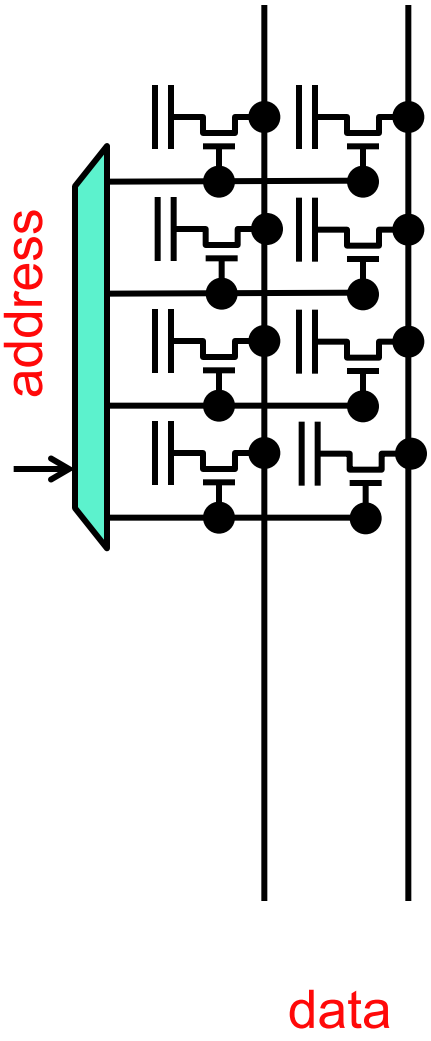
- RAM: large storage arrays
- Basic structure
  - MxN array of bits (M N-bit words)
    - This one is 4x2
  - Bits in word connected by **wordline**
  - Bits in position connected by **bitline**
- Operation
  - Address decodes into M wordlines
  - Assert wordline → word on bitlines
  - Bit/bitline connection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

SRAM



- **SRAM**: static RAM
  - Bits as cross-coupled inverters
  - Four transistors per bit
  - More transistors for ports
- **"Static"** means
  - Inverters connected to pwr/gnd
  - Bits naturally/continuously "refreshed"
  - Bit values never decay
- Designed for speed

DRAM



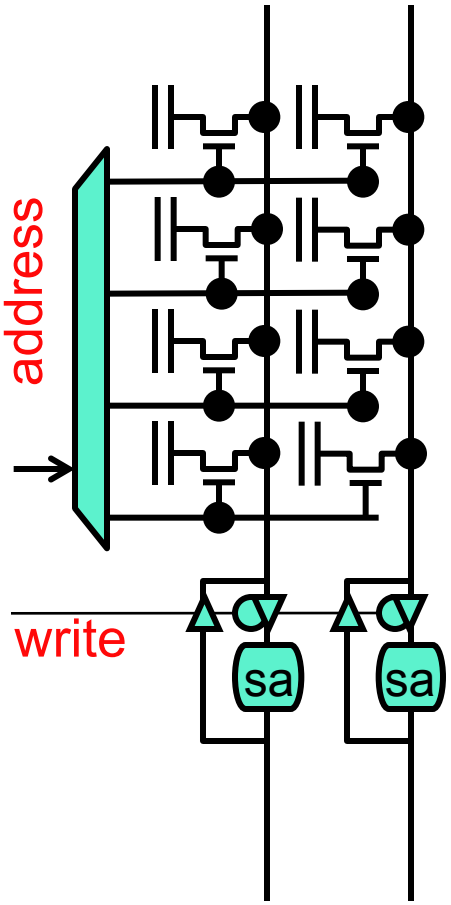
- **DRAM**: dynamic RAM
  - Bits as capacitors (if charge, bit=1)
  - Pass transistors as ports
  - One transistor per bit/port
- **"Dynamic"** means
  - Capacitors not connected to pwr/gnd
  - Stored charge decays over time
  - Must be explicitly refreshed
- Designed for density
  - Moore's Law …

Moore's Law (DRAM capacity)

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

- 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

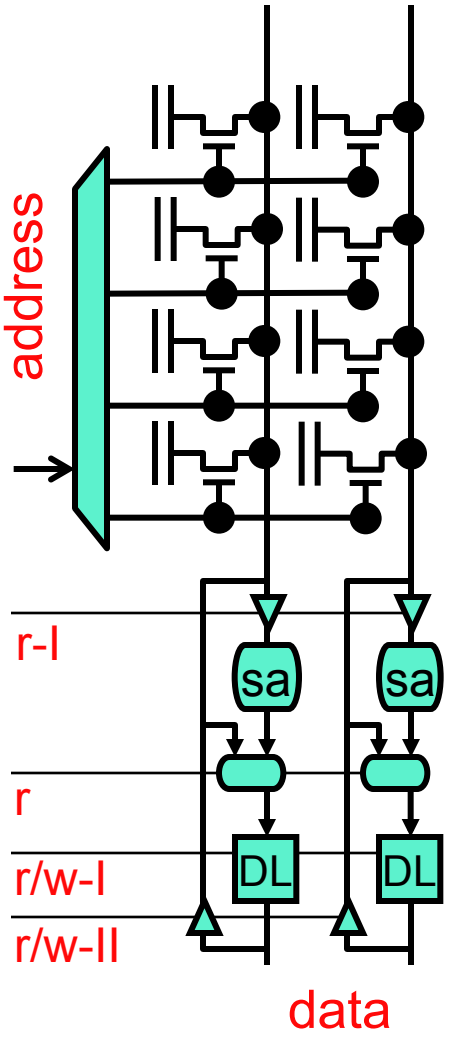
DRAM Operation I



- Read: similar to SRAM read
  - Phase I: pre-charge bitlines (to ~0.5V)
  - 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

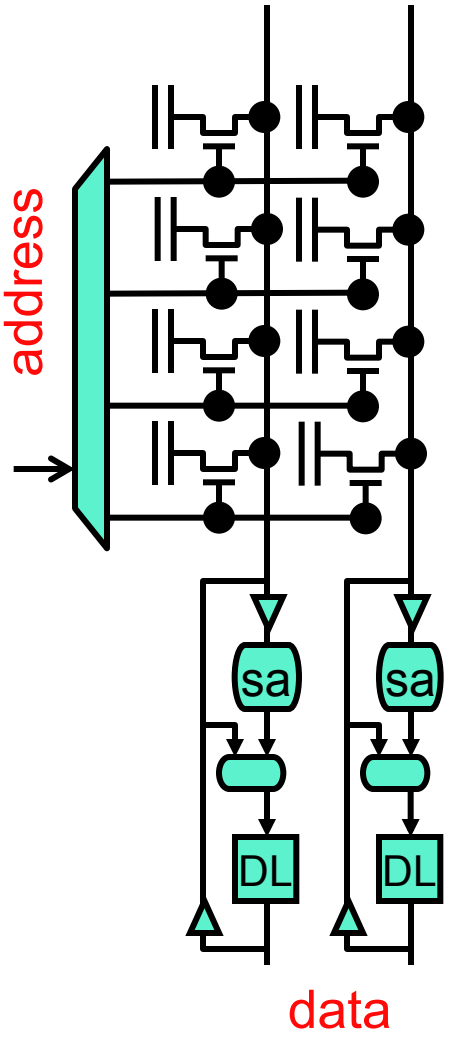
– What about **leakage over time**?

DRAM Operation II



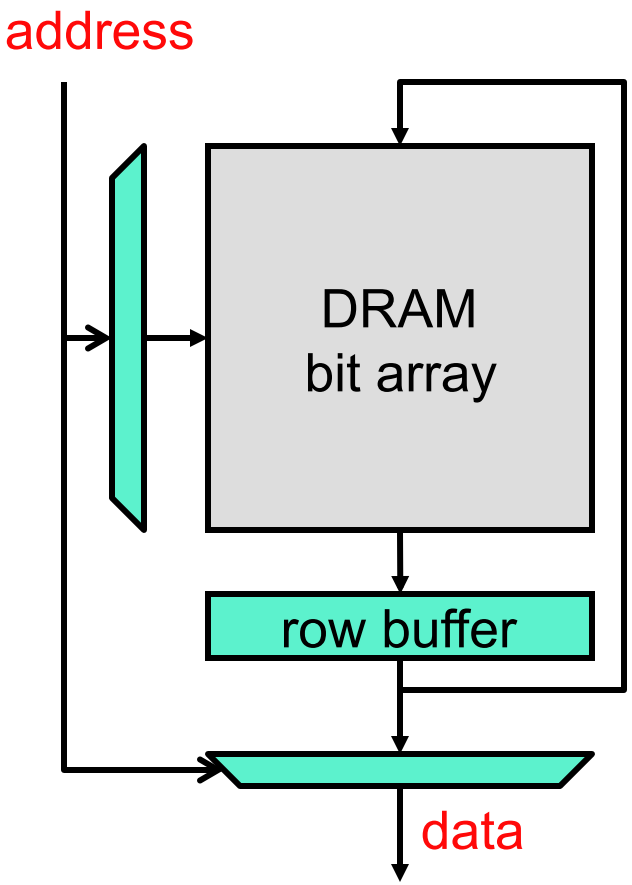
- Solution: add set of D-latches (**row buffer**)
- Read: two steps
  - Step I: read selected word into row buffer
  - Step IIA: read row buffer out to pins
  - Step IIB: write row buffer back to selected word
  - \+ Solves "destructive read" problem
- Write: two steps
  - Step IA: read selected word 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 parameters
  - Large capacity: e.g., 1-4Gb
    - Arranged as square
    - + Minimizes wire length
    - + Maximizes refresh efficiency
  - Narrow data interface: 1–16 bit
    - Cheap packages → few bus pins
    - Pins are expensive
  - Narrow address interface: N/2 bits
    - 16Mb DRAM had a 12-bit address bus
    - How does that work?

Access Time and Cycle Time

- DRAM access slower than SRAM
  - More bits → longer wires
  - Buffered access with two-level addressing
  - 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

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

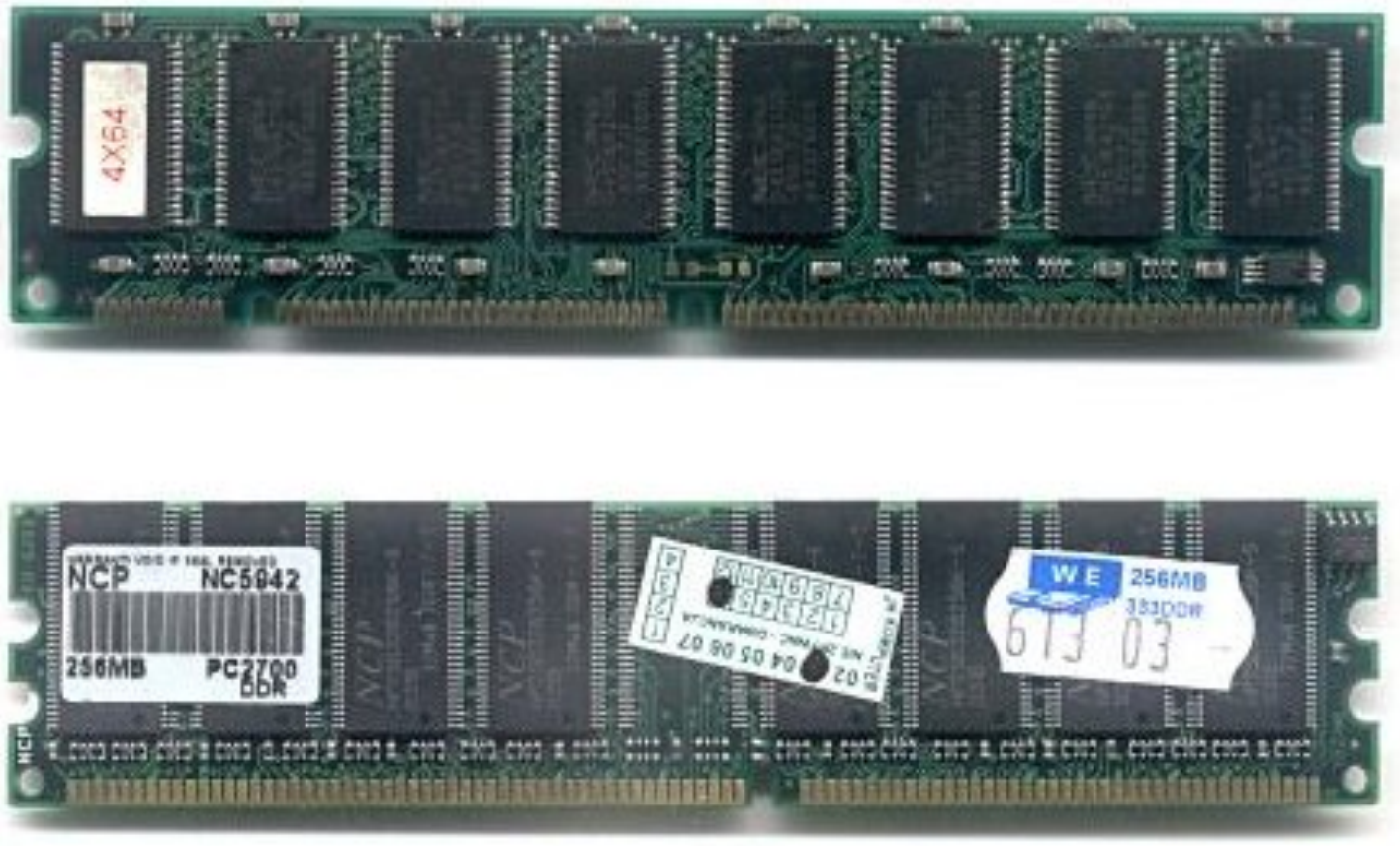
Memoirs of a Computer Programmer, 1985

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
- RDRAM = Rambus DRAM
  - Very fast, expensive DRAM
- GDRAM = graphics DRAM
  - GDDR

DRAM Packaging

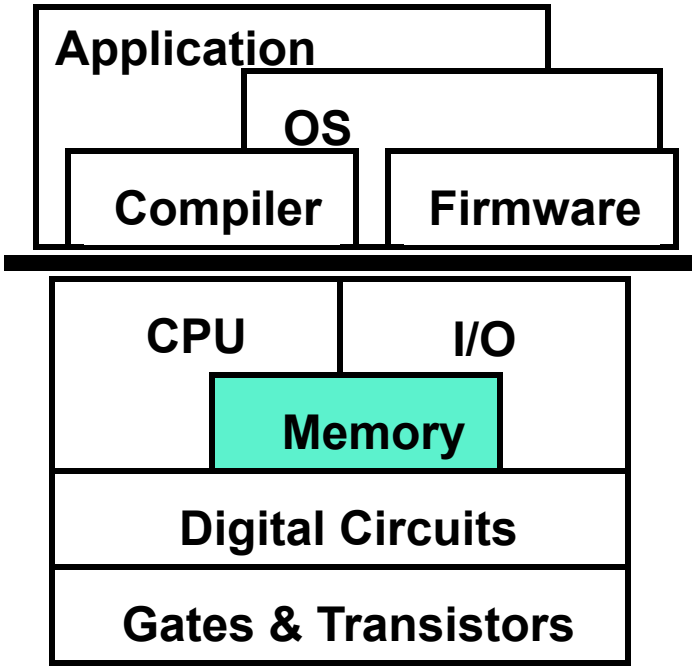
- DIMM = dual inline memory module
  - E.g., 8 DRAM chips, each chip is 4 or 8 bits wide



DRAM: A Vast Topic

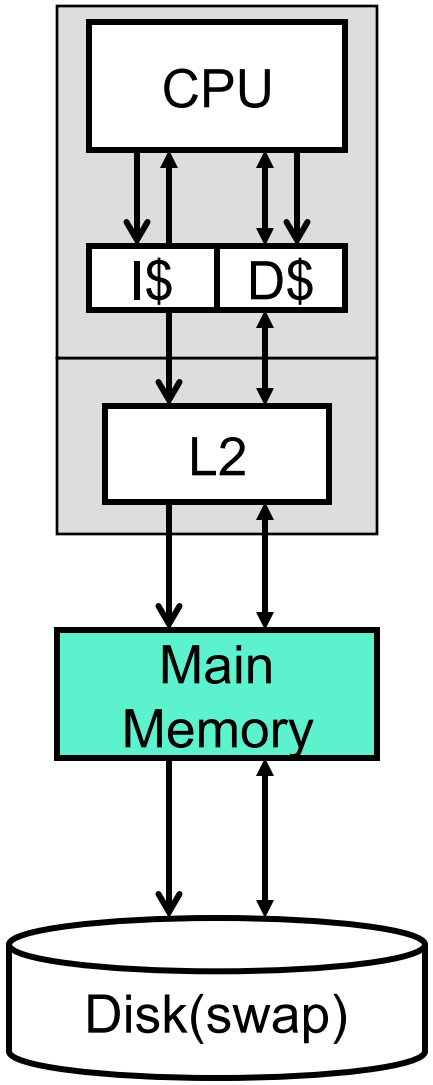
- Many flavors of DRAMs
  - DDR3 SDRAM, RDRAM, GDRAM, etc.
- Many ways to package them
  - SIMM, DIMM, FB-DIMM, etc.
- Many different parameters to characterize their timing
  - t_{RC}, t_{RAC}, t_{RCD}, t_{RAS}, etc.
- Many ways of using row buffer for "caching"
- Etc.
- There's at least one whole textbook on this topic!
  - And it has ~1K pages
- We could, but won't, spend rest of semester on DRAM

This Unit: Main Memory



- Memory hierarchy review
- DRAM technology
  - A few more transistors
  - Organization: two level addressing
- Building a memory system
  - Bandwidth matching
  - Error correction
- Organizing a memory system
- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

Building a Memory System



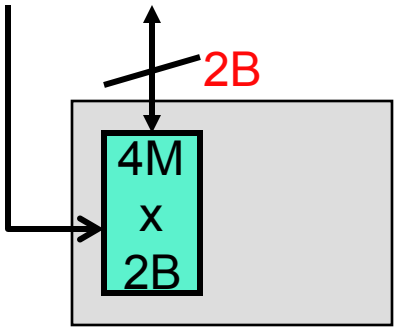
- How do we build an efficient main memory out of standard DRAM chips?
  - How many DRAM chips?
  - What width/speed (data) bus to use?
    - Assume separate address bus

An Example Memory System

- Parameters
  - 32-bit machine
  - L2 with 32B blocks (must pull 32B out of memory at a time)
  - 4Mx16b DRAMs, 20ns access time, 40ns cycle time
    - Each chip is 4Mx2B = 8 MB
  - 100MHz (10ns period) data bus
  - 100MHz, 32-bit address bus

- How many DRAM chips?
- How wide to make the data bus?

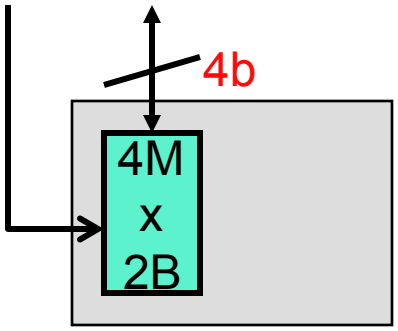
First Memory System Design



- 1 DRAM + 16b (=2B) bus
  - Access time: 630ns
    - Not including address
  - Cycle time: 640ns
    - DRAM ready to handle another miss
  - Observation: data bus idle 75% of time!
    - We have over-designed bus
    - Can we use a cheaper bus?

T (ns)	DRAM	Data Bus
10	**[31:30]**	
20	**[31:30]**	
30	**refresh**	**[31:30]**
40	**refresh**	
50	[29:28]	
60	[29:28]	
70	refresh	[29:28]
80	refresh	
…	…	…
600	refresh	
610	[1:0]	
620	[1:0]	
630	refresh	**[1:0]**
640	refresh	

Second Memory System Design

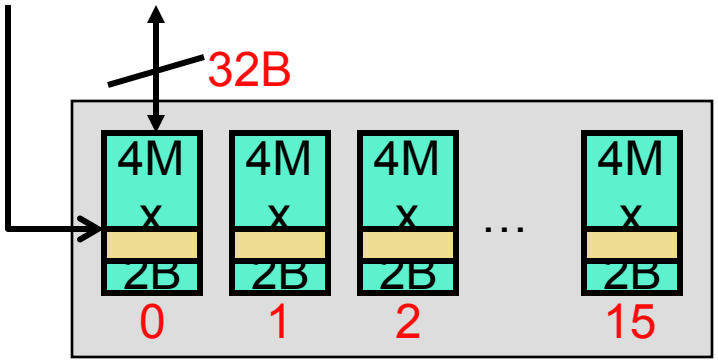


- 1 DRAM + **4b** bus
  - One DRAM chip, don't need 16b bus
  - DRAM: 2B / 40ns → 4b / 10ns
  - Balanced system → match bandwidths
  - Access time: 660ns (30ns longer=+4%)
  - Cycle time: 640ns (same as before)

  \+ Much cheaper!

T (ns)	DRAM	Bus
10	**[31:30]**	
20	**[31:30]**	
30	**refresh**	**[31H]**
40	**refresh**	**[31L]**
50	[29:28]	**[30H]**
60	[29:28]	**[30L]**
70	refresh	[29H]
80	refresh	[29L]
…	…	…
600	[1:0]	[2H]
610	[1:0]	[2L]
620	refresh	[1H]
640	refresh	[1L]
650		[0H]
660		**[0L]**

Third Memory System Design



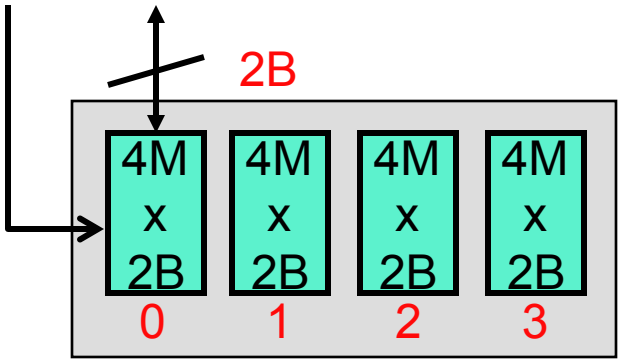
T (ns)	DRAM0	DRAM1	DRAM15	Bus
10	**[31:30]**	**[29:28]**	**[1:0]**	
20	**[31:30]**	**[29:28]**	**[1:0]**	
30	**refresh**	**refresh**	**refresh**	**[31:0]**
40	**refresh**	**refresh**	**refresh**	

- How fast can we go?
- 16 DRAM chips + 32B bus
  - **Stripe data across chips**
  - Byte M in chip (M/2)%16 (e.g., byte 38 is in chip 3)
  - Access time: 30ns
  - Cycle time: 40ns
  - – 32B bus is very expensive

Latency and Bandwidth

- In general, given bus parameters…
  - Find smallest number of chips that minimizes cycle time
  - Approach: match bandwidths between DRAMs and data bus
    - If they don't match, you're paying too much for the one with more bandwidth

Fourth Memory System Design



- 2B bus
  - Bus b/w: 2B/10ns
  - DRAM b/w: 2B/40ns
  - 4 DRAM chips
  - Access time: 180ns
  - Cycle time: 160ns

T (ns)	DRAM0	DRAM1	DRAM2	DRAM3	Bus
10	**[31:30]**	**[29:28]**	**[27:26]**	**[25:24]**	
20	**[31:30]**	**[29:28]**	**[27:26]**	**[25:24]**	
30	**refresh**	**refresh**	**refresh**	**refresh**	**[31:30]**
40	**refresh**	**refresh**	**refresh**	**refresh**	**[29:28]**
50	[23:22]	[21:20]	[19:18]	[17:16]	**[27:26]**
60	[23:22]	[21:20]	[19:18]	[17:16]	**[25:24]**
…	…	…	…	…	…
110	refresh	refresh	refresh	refresh	[15:14]
120	refresh	refresh	refresh	refresh	[13:12]
130	[7:6]	[5:4]	[3:2]	[1:0]	[11:10]
140	[7:6]	[5:4]	[3:2]	[1:0]	[9:8]
150	refresh	refresh	refresh	refresh	[7:6]
160	refresh	refresh	refresh	refresh	[5:4]
170					[3:2]
180					**[1:0]**

Memory Access and Clock Frequency

- Nominal **clock frequency** applies to CPU and caches
  - Memory bus has its own clock, typically much slower
  - SDRAM operates on bus clock

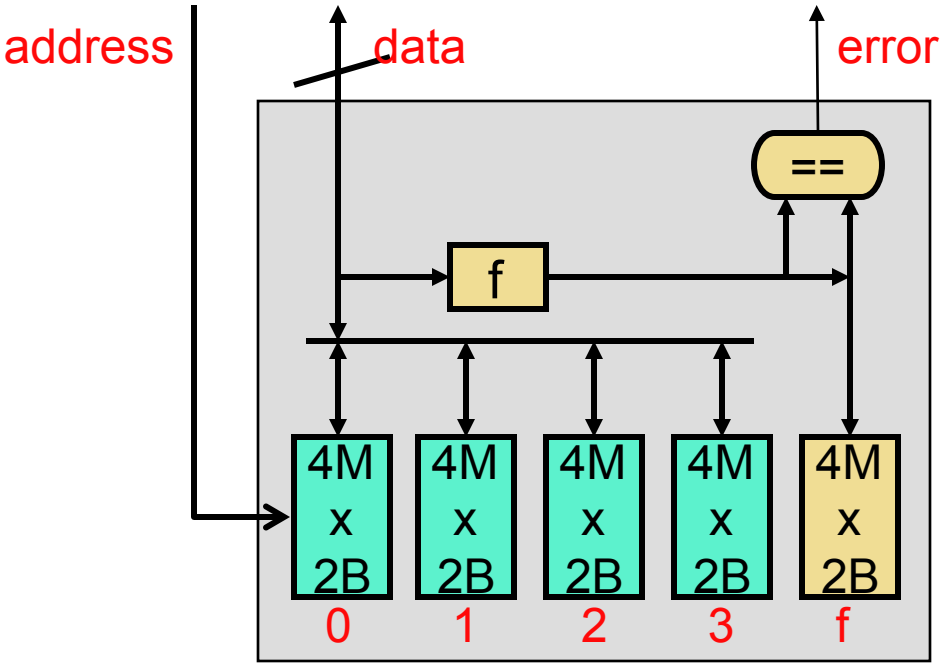
- Another reason why processor clock frequency isn't perfect performance metric
  - Clock frequency increases don't reduce memory or bus latency
  - May make misses come out faster
    - At some point memory bandwidth may become a **bottleneck**
    - Further increases in (core) clock speed won't help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

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($data_{N-1...0}$) = 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
  – Take ECE 254 (Fault Tolerant Computing) for more info

Memory Organization

- So data is striped across DRAM chips
- But how is it organized?
  - Block size?
  - Associativity?
  - Replacement policy?
  - Write-back vs. write-thru?
  - Write-allocate vs. write-non-allocate?
  - Write buffer?
  - Optimizations: victim buffer, prefetching, anything else?

Low $\%_{miss}$ At All Costs

- For a memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → minimal t_{hit} important
  - t_{miss} is not bad → minimal $\%_{miss}$ less important
  - Low capacity/associativity/block-size, write-back or write-through
- Moving down (L2) emphasis turns to $\%_{miss}$
  - Infrequent access → minimal t_{hit} less important
  - t_{miss} is bad → minimal $\%_{miss}$ important
  - High capacity/associativity/block size, write-back
- For memory, emphasis entirely on $\%_{miss}$
  - t_{miss} is disk access time (measured in ms, not ns)

Typical Memory Organization Parameters

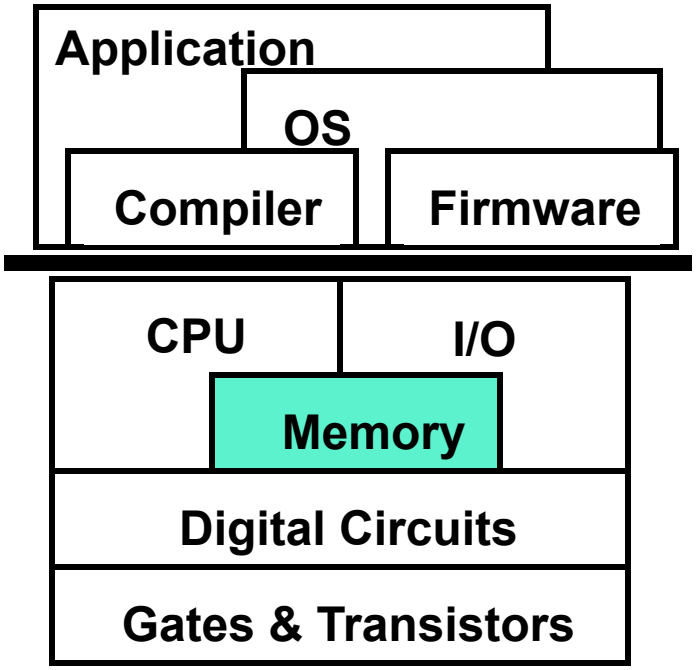
Parameter	I$/D$	L2	Main Memory
t_{hit}	1-2ns	10ns	**30ns**
t_{miss}	**10ns**	**30ns**	**10ms (10M ns)**
Capacity	8–64KB	128KB–2MB	**512MB–8GB**
Block size	16–32B	32–256B	**8–64KB pages**
Associativity	1–4	4–16	**Full**
Replacement Policy	NMRU	NMRU	**working set**
Write-through?	Sometimes	No	**No**
Write-allocate?	Sometimes	Yes	**Yes**
Write buffer?	Yes	Yes	**No**
Victim buffer?	Yes	No	**No**
Prefetching?	Sometimes	Yes	**Sometimes**

One Last Gotcha

- On a 32-bit architecture, there are 2^{32} byte addresses
  - Requires 4 GB of memory
  - But not everyone buys machines with 4 GB of memory
  - And what about 64-bit architectures?

- Let's take a step back…

This Unit: Main Memory



- Memory hierarchy review
- DRAM technology
  - A few more transistors
  - Organization: two level addressing
- Building a memory system
  - Bandwidth matching
  - Error correction
- Organizing a memory system
- Virtual memory
  - Address translation and page tables
  - A virtual memory hierarchy

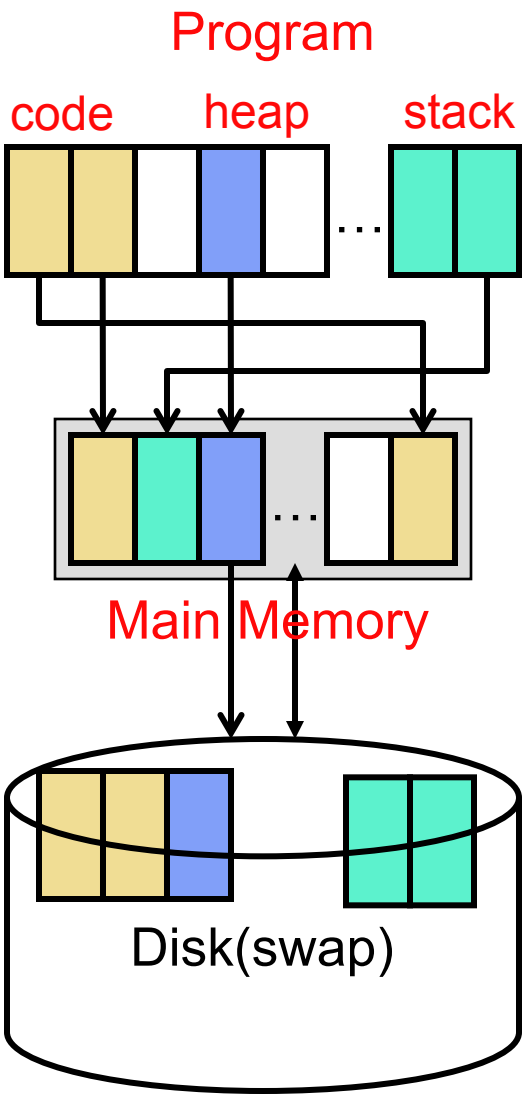
Virtual Memory

- Idea of treating memory like a cache
  - Contents are a dynamic subset of program's address space
  - Dynamic content 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)

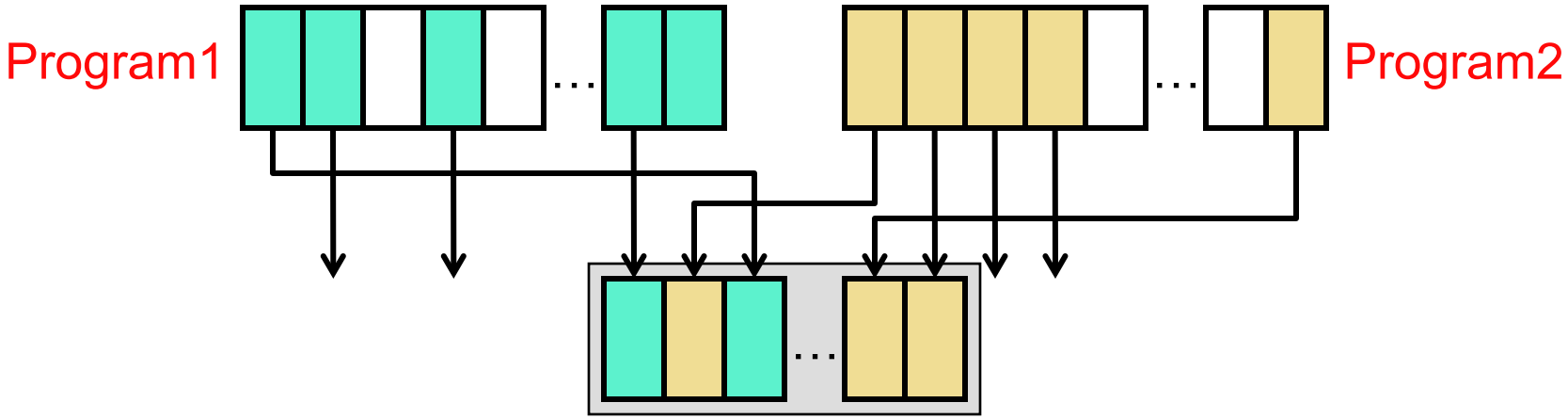
Virtual Memory



- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Itanium is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (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)

Other Uses of Virtual Memory

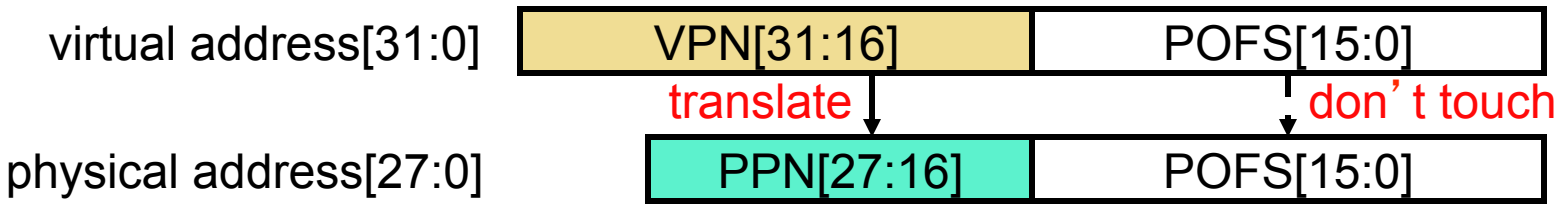
- Virtual memory is quite useful
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - "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

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]

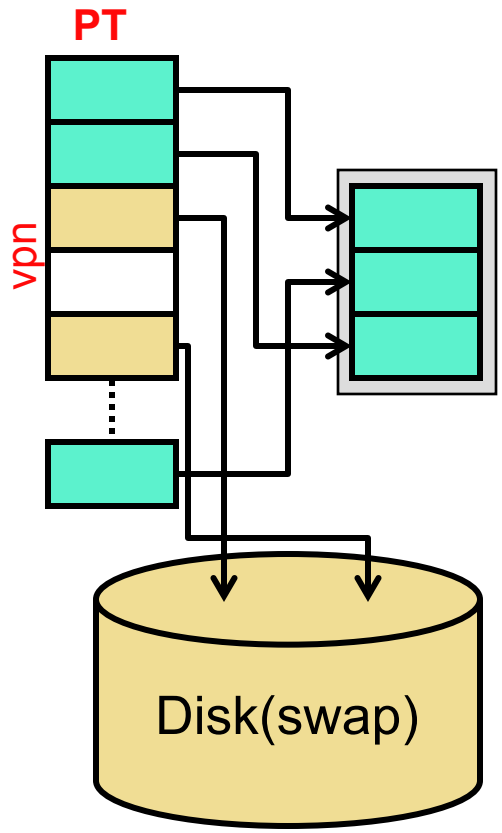
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN (16 = 32 – 16)
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

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)**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (pt[vpn].is_valid)
     return pt[vpn].ppn;
}
```



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

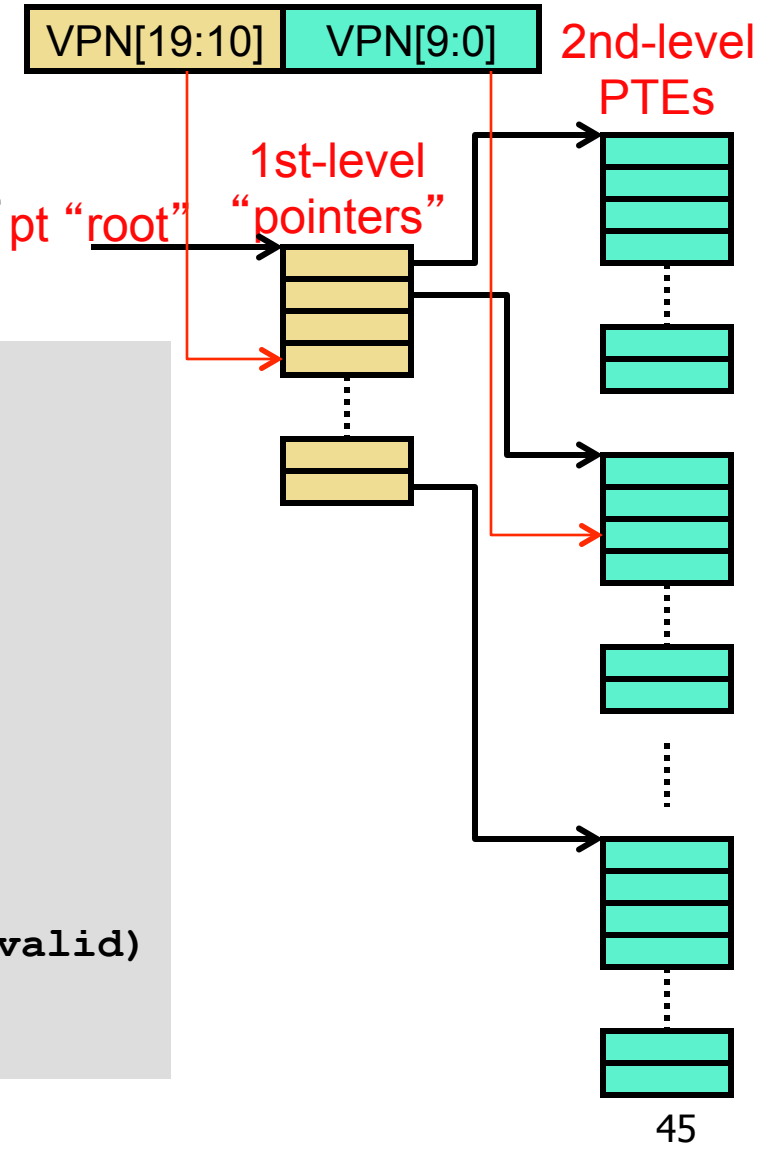
Multi-Level Page Table

- One way: **multi-level page tables**
  - 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)
    - 4KB pages / 4B PTEs → 1K PTEs fit on a single page
    - 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

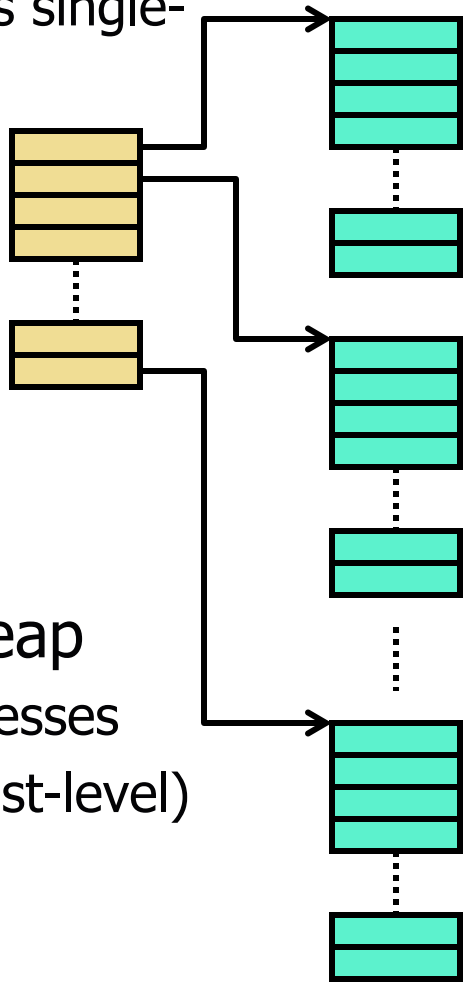



```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```

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)?
  - 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?**

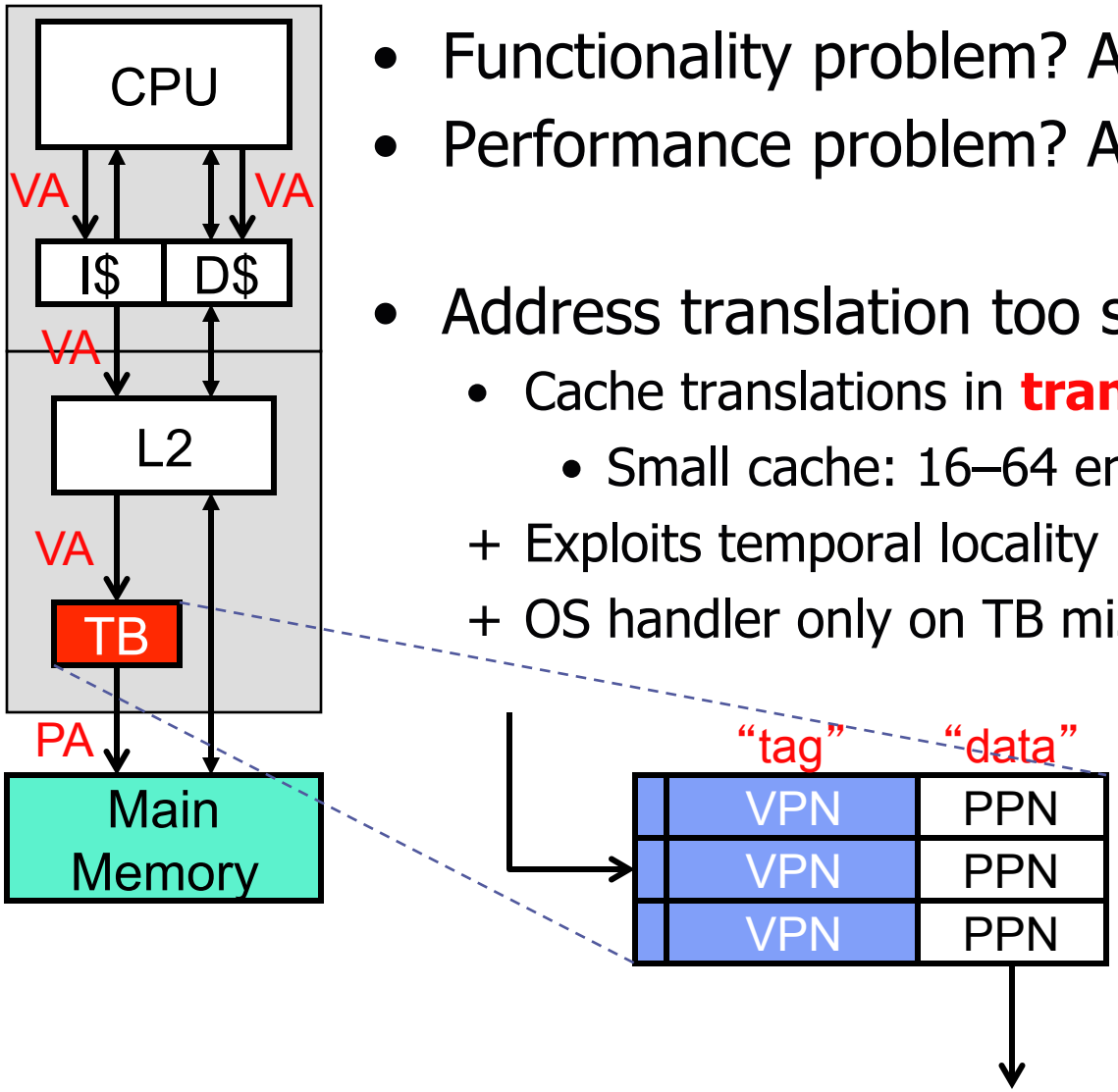
- Option I: process (program) translates its own addresses
  - Page table resides in process visible virtual address space
  - – Bad idea: implies that program (and programmer)…
    - …must know about physical addresses
      - Isn't that what virtual memory is designed to avoid?
    - …can forge physical 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

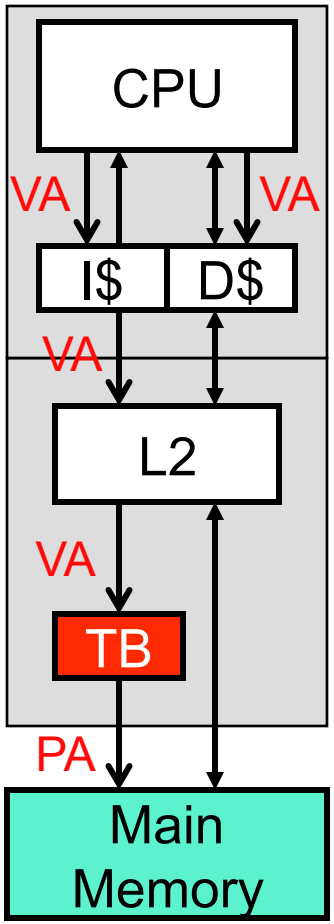
- Handling is tricky for SW handler, but possible
  - First, save current TB miss info before accessing page table
    - So that nested TB miss info doesn't overwrite it
  - Second, **lock nested miss entries into TB**
    - Prevent TB conflicts that result in infinite loop
    - Another good reason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

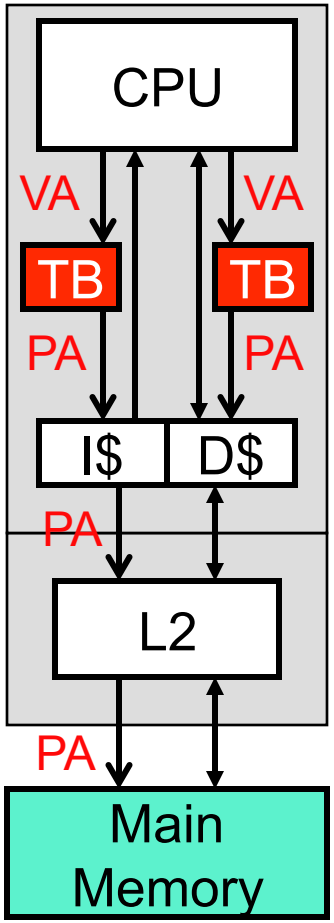
- **OS routine**
  - OS software chooses a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk (like dirty cache block with 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



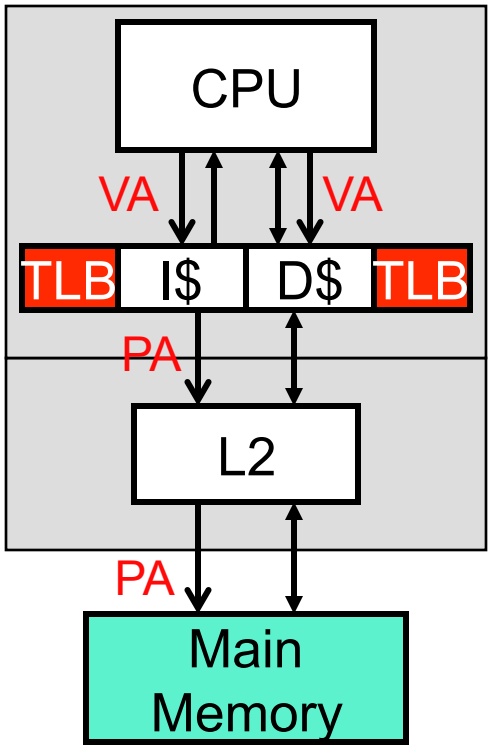
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access memory
  - \+ Fast: avoids translation latency in common case
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **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

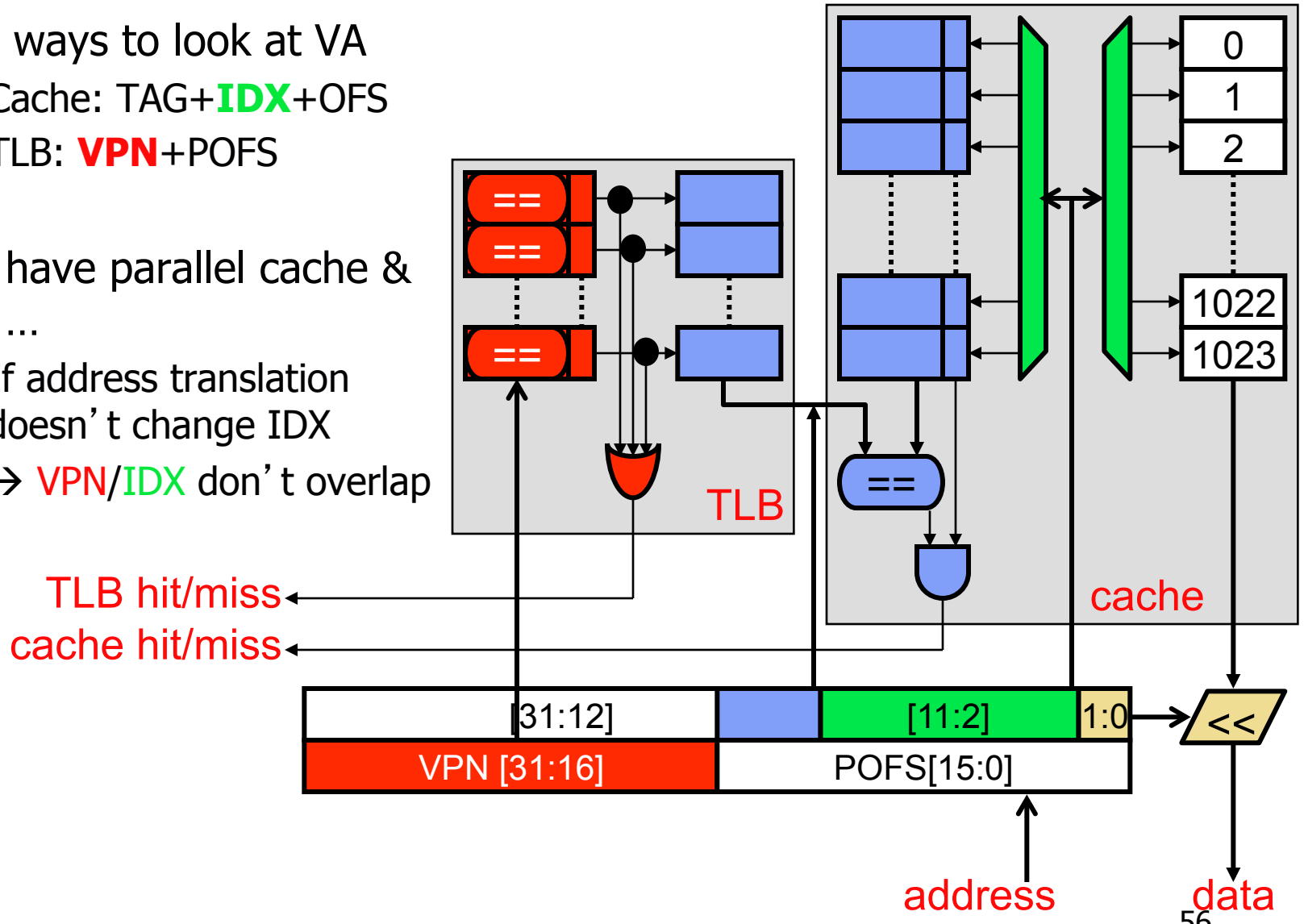
+ No context-switching/aliasing problems
+ Fast: no additional t_{hit} cycles

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

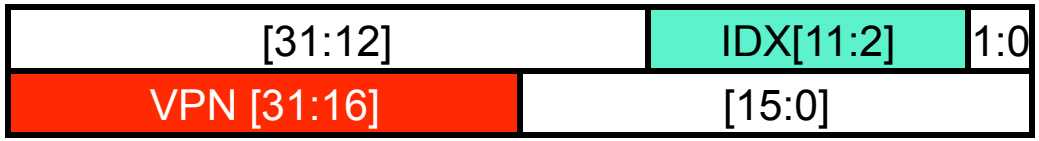
- Common organization in processors today

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



- 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 VPs 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
- Building a memory system
  - DRAM/bus bandwidth matching
- 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