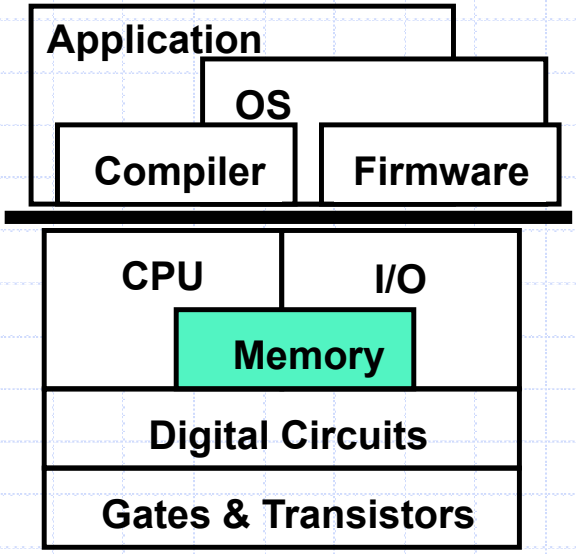
ECE565: Computer Architecture

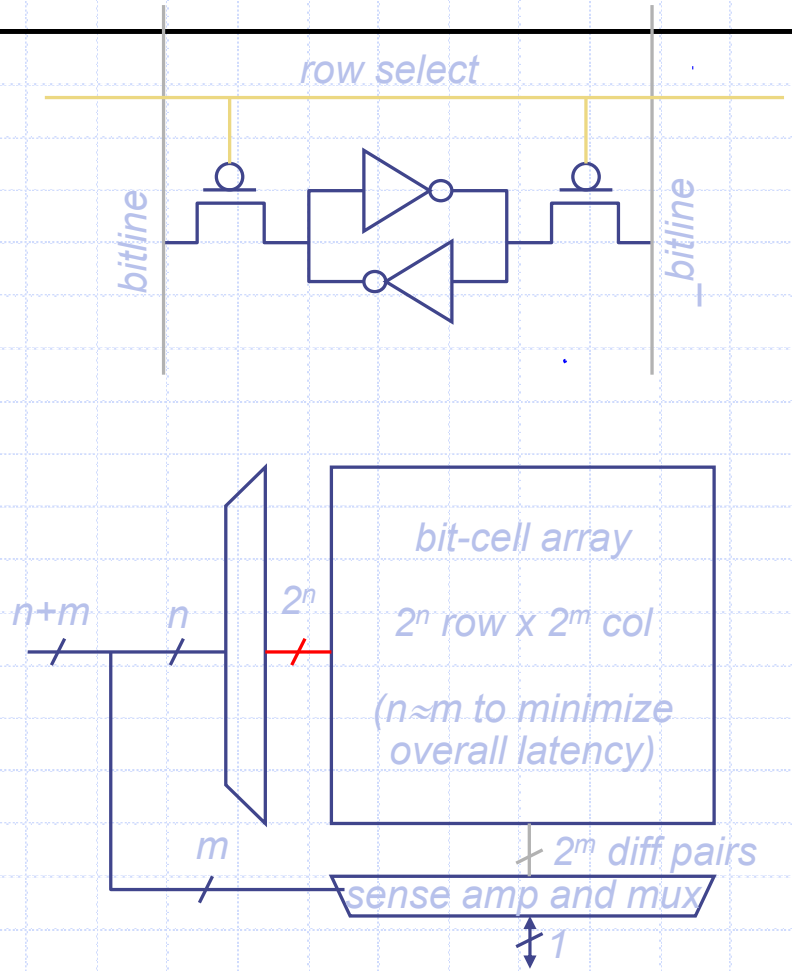
Ch 2b

This Unit: Main Memory



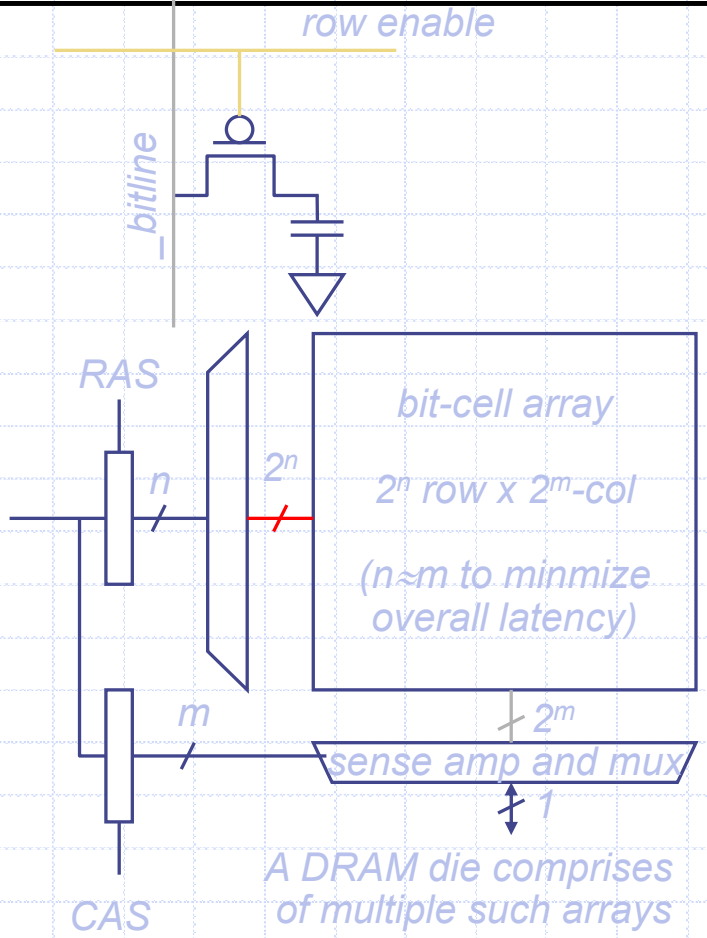
- Memory hierarchy review
- Virtual memory
  - Address translation and page tables
  - Virtual memory's impact on caches
  - Page-based protection
- Organizing a memory system
  - Bandwidth matching
  - Error correction

Static Random Access Memory



- Read Sequence
  1. address decode
  2. drive row select
  3. selected bit-cells drive bitlines
  4. diff. sensing and col. select
  5. precharge all bitlines
- Access latency dominated by steps 2 and 3
- Cycling time dominated by steps 2, 3 and 5
  - step 2 proportional to 2^m
  - step 3 and 5 proportional to 2^n
- usually encapsulated by synchronous (sometime pipelined) interface logic

Dynamic Random Access Memory



- Bits stored as charges on node capacitance (non-restorative)
  - bit cell loses charge when read
  - bit cell loses charge over time
- Read Sequence

  1~3 same as SRAM

  4. the sense amp amplifies and regenerates the bitline, data bit is mux'ed out

  5. precharge all bitlines

Refresh: A DRAM controller must periodically, either distributed or in a burst, read all rows within the allowed refresh time (10s of ms) synchronous interfaces

Brief History of DRAM

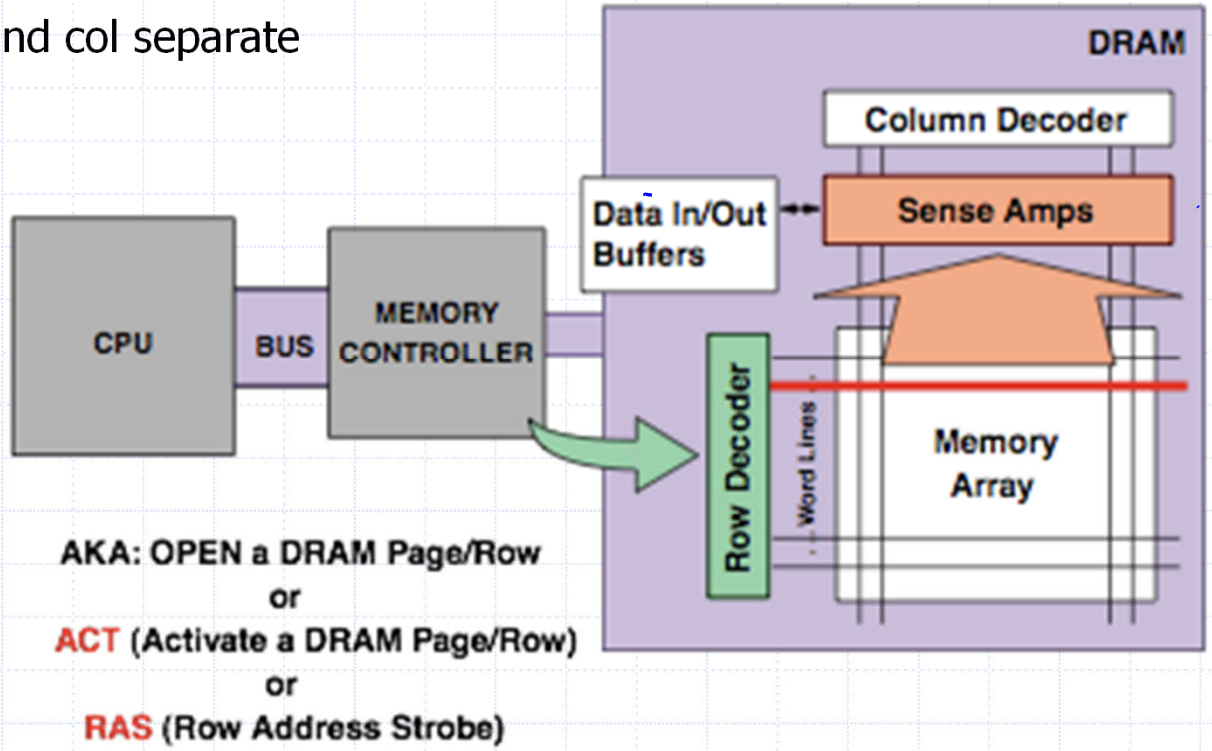
- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - Each cell was a small magnetic "donut"
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"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

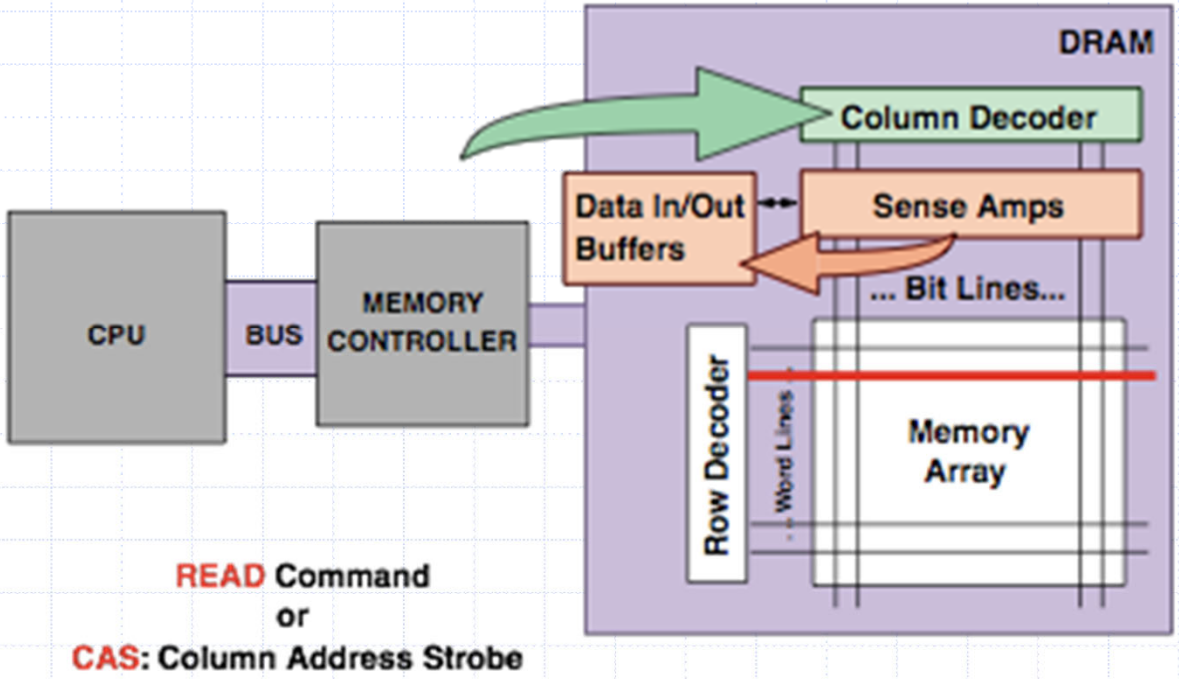
DRAM Basics [Jacob and Wang]

- Precharge and Row Access: internal details exposed unlike SRAM
  - SRAM – one address
  - DRAM – row and col separate



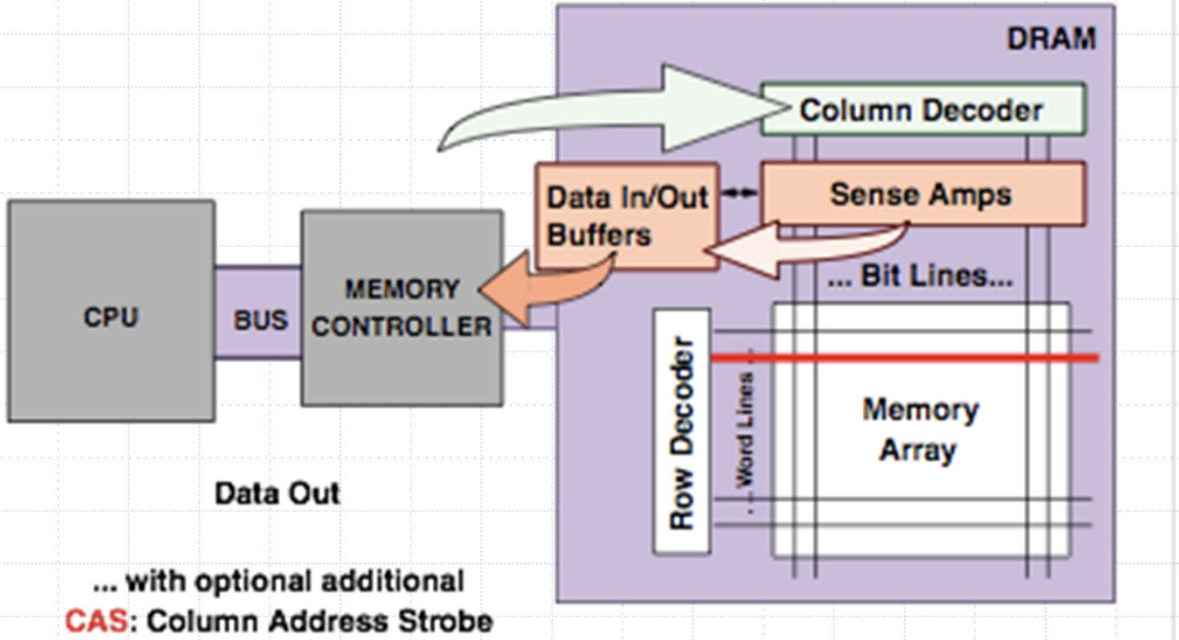
DRAM Basics, cont.

- Column Access



DRAM Basics, cont.

- Data Transfer



Optimization #1: Row buffer (page) locality for latency

- various tricks to allow faster repeated accesses to the same row
- Open Page
  - Row stays active until another row needs to be accessed
  - Acts as memory-level cache to reduce latency
  - Variable access latency complicates memory controller
  - Higher power dissipation (sense amps remain active)
  - Longer latency if few row hits
- Closed Page
  - Immediately deactivate row after access
  - Better if few row hits
  - All accesses become Activate Row, Read/Write, Precharge
- Complex power v. performance trade off

DRAM Bandwidth

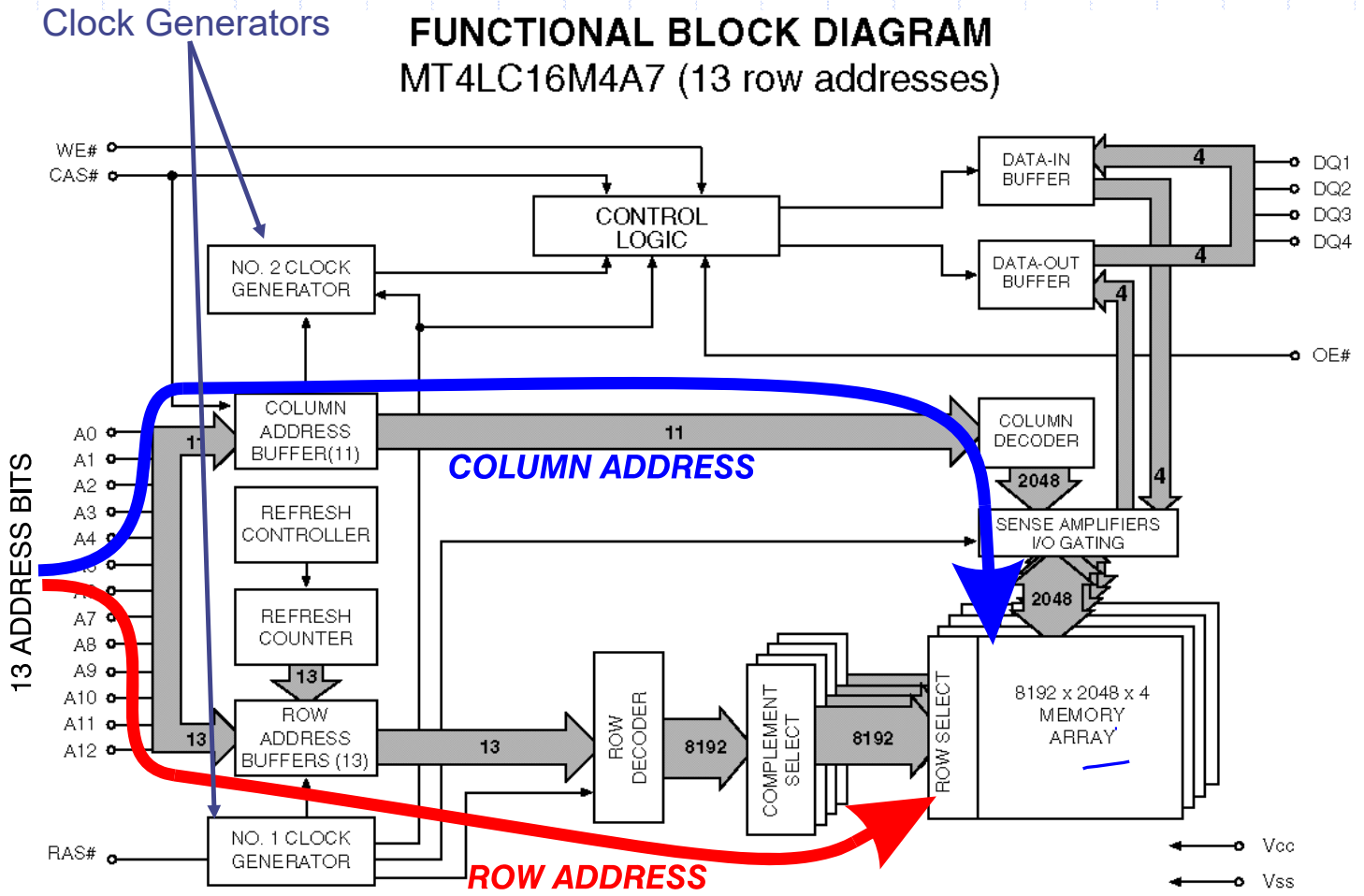
- Use multiple DRAM chips to increase bandwidth
  - Recall, access are the same size as last-level cache
  - Example, 16 2-byte wide chips for 32B access
- DRAM density increasing faster than demand
  - Result: number of memory chips per system decreasing
- Need to increase the **bandwidth per chip**
  - Especially important in game consoles
  - SDRAM ➔ DDR ➔ DDR2 ➔ DDR3

DRAM Evolution for optimization #2

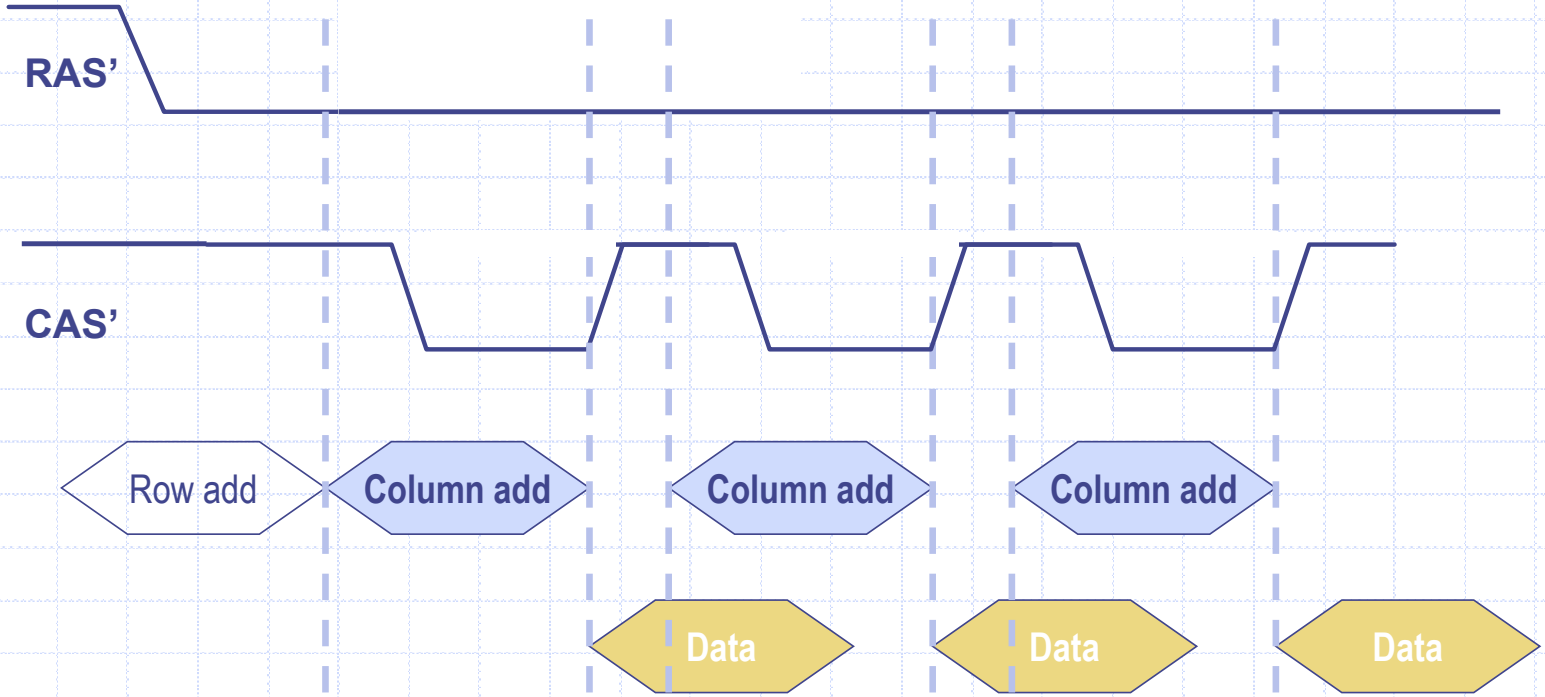
- Survey by Cuppu et al.

1. Early Asynchronous Interface
2. Extended Data Out
3. Synchronous DRAM & Double Data Rate 2/3/4

Old 64MbitDRAM Example from Micron

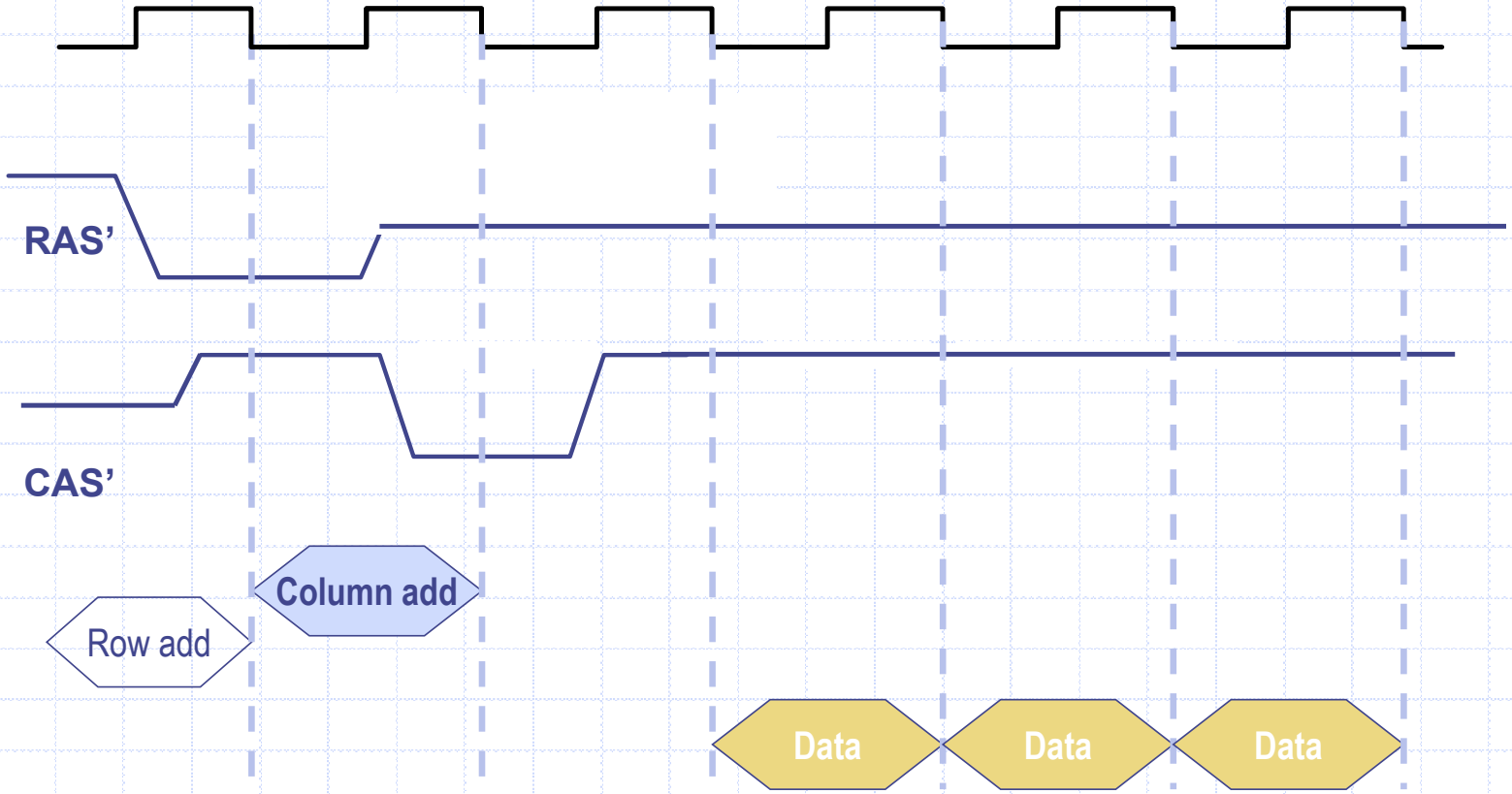


Extended Data Out (EDO)



- Keep row open
- Overlapped Column Address assert with Data Out

Synchronous DRAM (SDRAM)



- Add Clock and Wider data!
- Also multiple transfers per RAS/CAS

Enhanced SDRAM & DDR

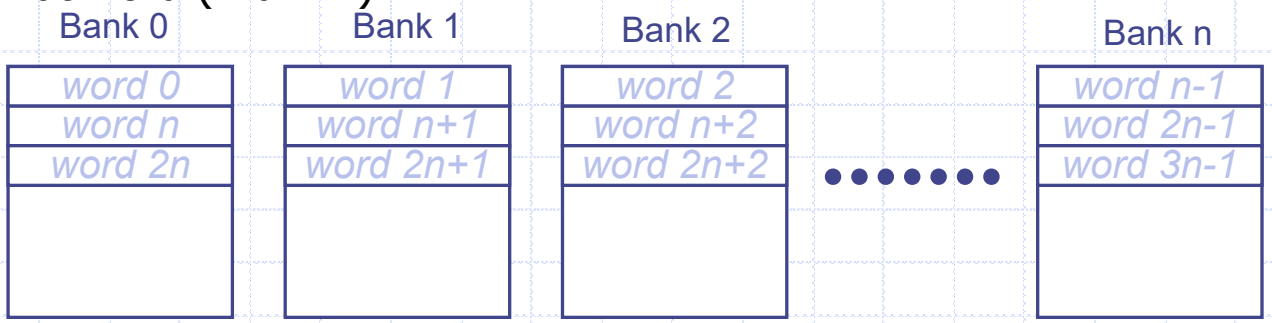
- Evolutionary Enhancements on SDRAM:

1. ESDRAM (Enhanced): Overlap row buffer access with refresh

2. DDR (Double Data Rate): Transfer on both clock edges
3. DDR2's small improvements
   lower voltage, on-chip termination, driver calibration
   prefetching, conflict buffering
4. DDR3, more small improvements
   lower voltage, 2X speed, 2X prefetching,
   2X banks, automatic calibration

Optimization #3: Interleaved Main Memory (old)

- Divide memory into M banks and "interleave" addresses across them, so word A is
  - in bank (A mod M)
  - at word (A div M)

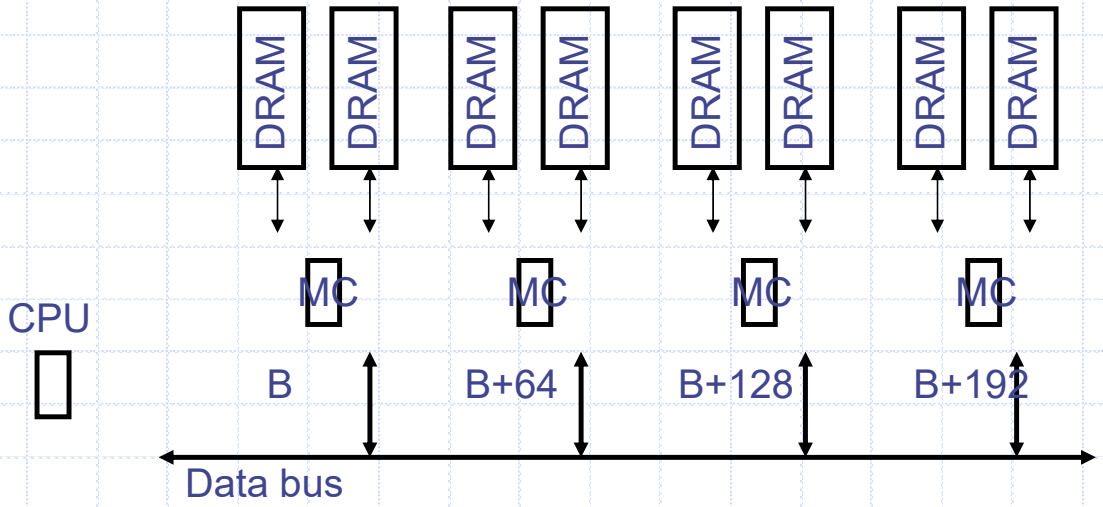


Interleaved memory increases memory BW without wider bus

- *Use parallelism in memory banks to hide memory latency*

Block interleaved memory systems (modern)

- Cache blocks map to separate memory controllers
  - Interleave across DRAMs within a MC
  - Interleave across DRAM banks within a DRAM
    - Modern DRAMs are INTERNALLY highly banked (16-32 banks) for bandwidth



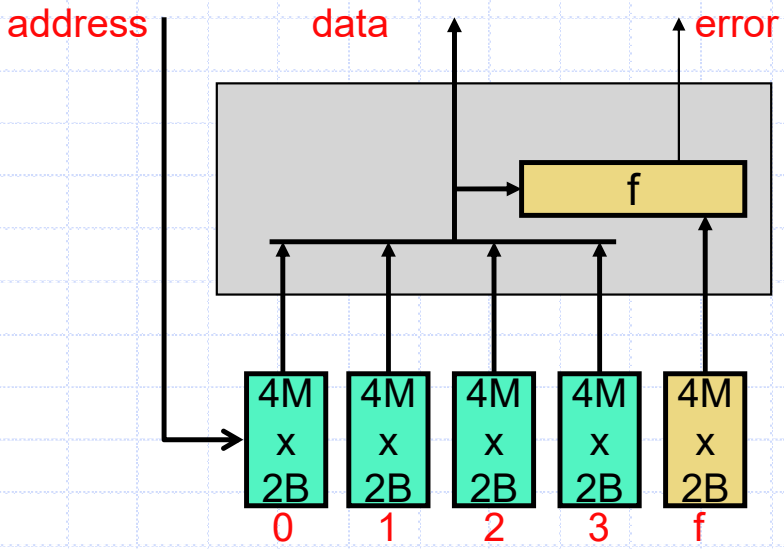
Block interleaved memory systems (modern)

- Within and across MC parallelism – lock-up free!
- In the presence of such highly interleaved memory, each memory controller manages parallelism across banks within the DRAM
  - Schedules accesses to different banks in parallel
  - Reads and writes drive bus in opposite directions so read-write change is slow → MCs batch reads and writes as much as possible
  - Schedules refreshes
  - Almost as complicated as an issue queue!

DRAM Reliability

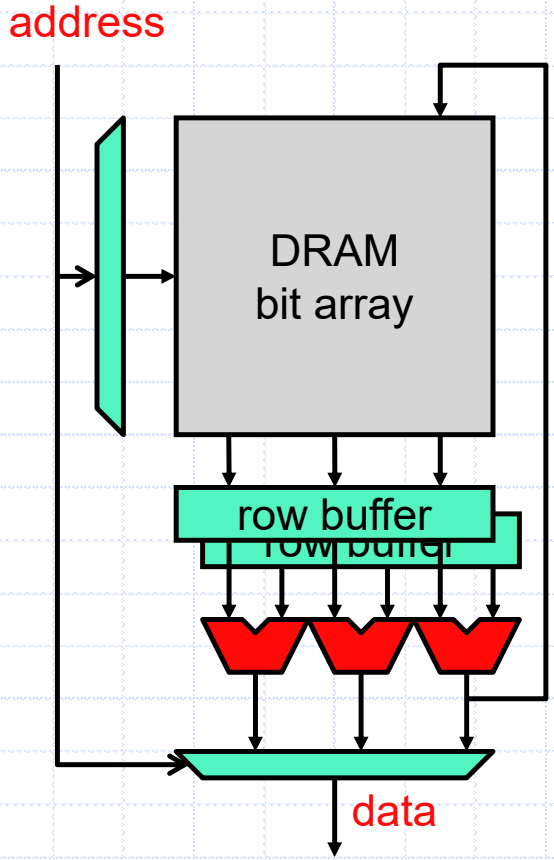
- One last thing about DRAM technology… **errors**
  - DRAM bits can flip from 0➔1 or 1➔0
    - Small charge stored per bit
    - Energetic α-particle strikes disrupt stored charge
    - Many more bits
  - Modern DRAM systems: built-in error detection/correction
    - Today all servers; most new desktop and laptops
- **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

DRAM Error Detection and Correction



- Performed by memory controller (not the DRAM chip)
- Error detection/correction schemes distinguished by…
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Research: Processing in Memory

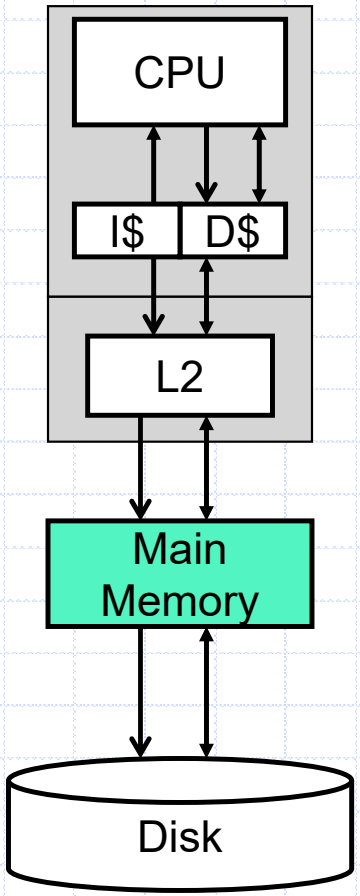


- **Processing in memory**
  - Embed some ALUs in DRAM
    - Picture is logical, not physical
  - Do computation in DRAM rather than…
    - Move data to from DRAM to CPU
    - Compute on CPU
    - Move data from CPU to DRAM
  - E.g.,: IRAM: intelligent RAM
    - Berkeley research project
    - [Patterson+,ISCA'97]

Memory Hierarchy Review

- Storage: registers, **memory**, disk
  - Memory is the fundamental element
- Memory component performance
  - **$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$**
  - 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 levels: caches (I$, D$, L2)
  - Made of SRAM
  - Last unit
- 3rd level: **main memory**
  - Made of DRAM
  - Managed in software
- 4th level: disk (swap space)
  - Made of magnetic iron oxide discs
  - Manage in software

Memory Organization

- Paged "virtual" memory
  - Programs want a conceptual view of a memory of unlimited size
  - Use disk as a *backing store* when physical memory is exhausted
  - Memory acts like a cache, managed (mostly) by software

  - BUT cache miss is millions of cycles so engineered differently than hardware caches

- How is the "memory as a cache" organized?

Virtual memory – basic 4 Q's

- How is the "memory as a cache" organized?
  - Block size? Blocks called Pages - typically 4-8KB or larger
- Architecture presents programs with a simple view
    - memory addressed with 32-bit addresses
    - lw $1, 0x100028 => 0x100028 is the "virtual address"
    - system maps VA to physical address (PA)
    - 0x100028 -> 0xF028 (page 15, offset 28 for 4K page)
- Basic 4 Qs for any cache (in 5 slides)
  - Q1: Placement
  - Q2: Identification
  - Q3: Replacement policy
  - Q4: Write policy?

Virtual memory

- Someone else and I run unrelated programs each
  - lw $r1, 0x100028
  - VA must map to different PA
- Thus, VA allows programs to:
  - Address more memory than system has [old reason]
  - think it is the only program running in memory
  - think it always starts at address 0x0: Compatibility
  - be protected from rogue programs
  - start running when most of the program is still on disk

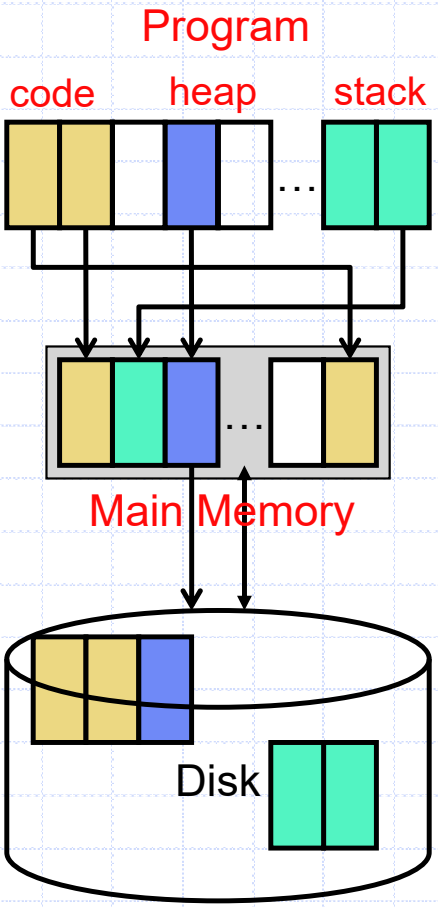
Virtual memory

- A VA miss is called a page fault
  - an exception that saves the PC
  - OS gains control and initiates disk access
  - OS usually runs someone else in the meantime
  - interrupt when disk access is complete
  - original instruction restarts
- Unlike cache misses, why is OS used to handle a page fault?

Virtual Memory

- Idea of treating memory like a cache…
  - Contents are a dynamic subset of program's address space
  - Dynamic content management transparent to program
- Original motivation: **capacity**
  - Atlas (1962): Fully-associative cache of pages, called *one-level store*
  - 16K words of core memory; 96K words of drum storage
- Successful 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 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)

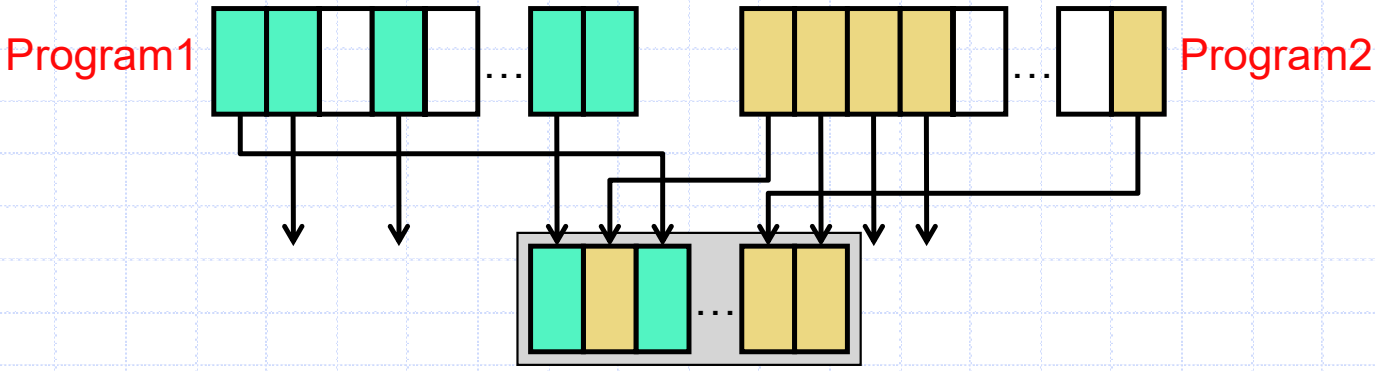
Virtual Memory



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

Modern uses of Virtual Memory

- Virtual memory is quite a useful feature
  - Automatic, transparent memory management is just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **program isolation and 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



More Uses of Virtual Memory

- **Isolation and Protection**
  - Piggy-back mechanism to implement page-level protection
  - Map virtual page to physical page
    - … and to Read/Write/Execute protection bits in page table
  - In multi-user systems
    - Prevent user from accessing another's memory
    - Only the operating system can see all system memory
  - Attempt to illegal access, to execute data, to write read-only data?
    - Exception → OS terminates program
- Inter-process communication
  - Map virtual pages in different processes to same physical page
  - Share files via the UNIX mmap() call

Back to the 4 questions

- Recall the 4 questions in caches
- Q1. Where does a block go?
  - Fully associative
    - Block offset and tag
    - NO INDEX
  - Why?
    - AMAT = hit time + miss-rate*miss-penalty
    - Miss-penalty = ~1 million cycles
    - Have to minimize miss-rate
    - full- associativity minimizes miss rate

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

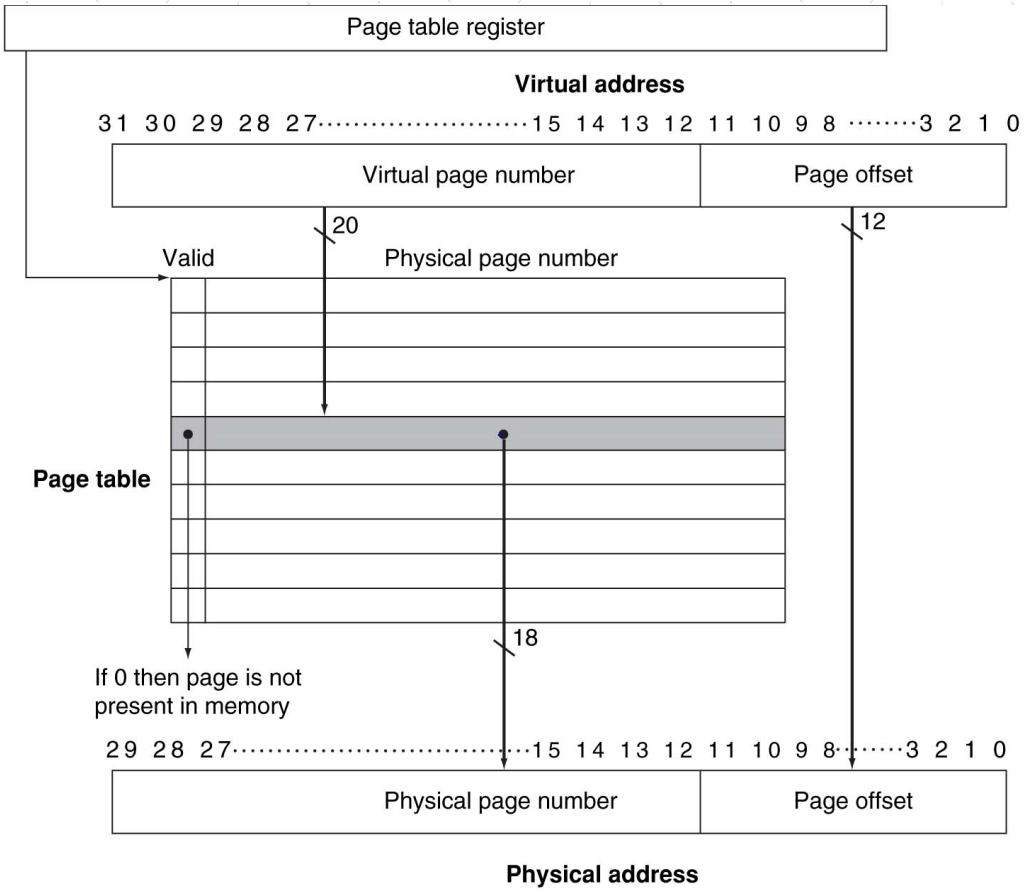
- For a memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access $\rightarrow$ minimal t_{hit} important
  - t_{miss} is not bad $\rightarrow$ minimal $\%_{miss}$ less important
  - Low capacity/associativity/block-size, write-back or write-thru
- Moving down (L2) emphasis turns to $\%_{miss}$
  - Infrequent access $\rightarrow$ minimal t_{hit} less important
  - t_{miss} is bad $\rightarrow$ 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)

Q2. Block Identification

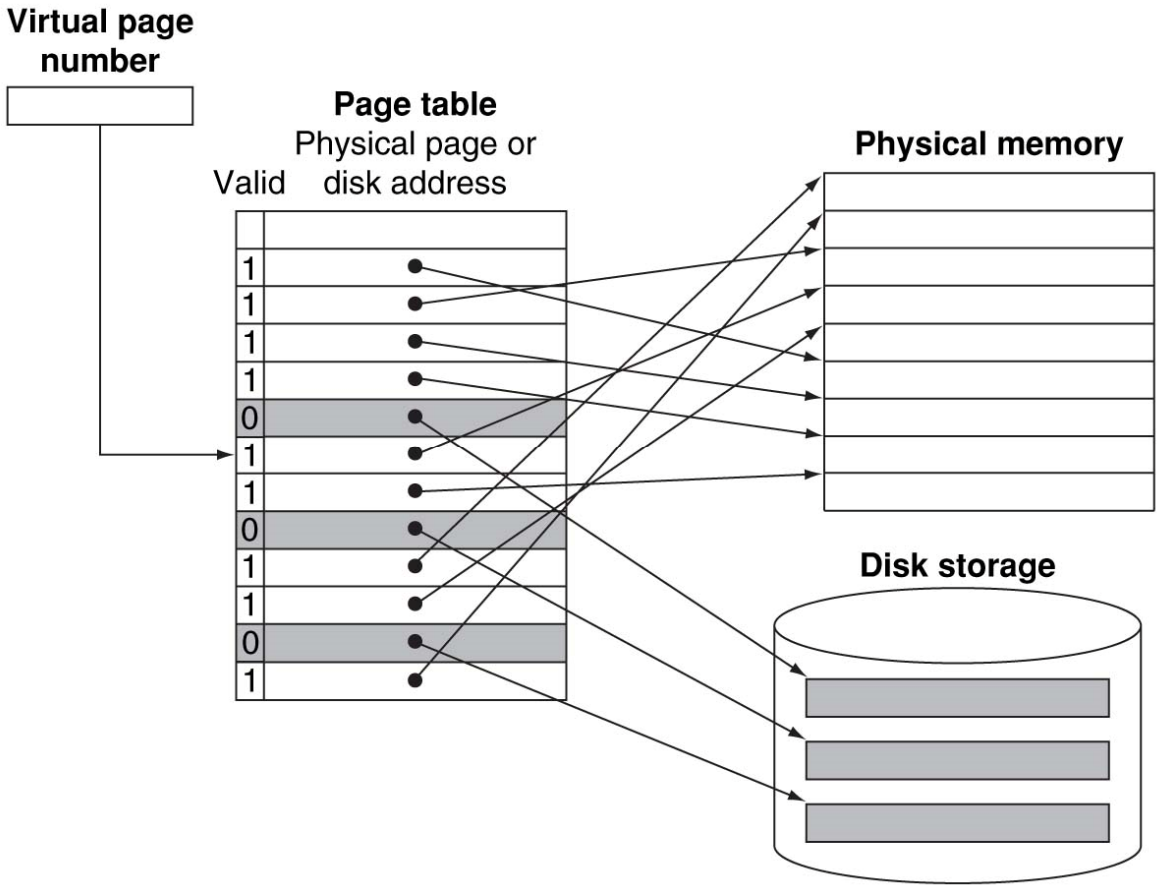
- Q2. Block identification
  - Fully associative search??
    - 30-bit physical address (1GB)
    - 4 KB pages
    - Number of frames = $2^{30}/2^{12} = 2^{18} = 256K$
    - Compare 256 K frames in parallel? Whoa!
  - Reframe question:
    - Cache (previously): Is this VA in any given frame? => Parallel search
    - VM (now): Where is this VA? => Table lookup
      - Page table
      - Fully associativity via table lookup
        - each table entry can point to ANY frame

Page Table

- Virtual page number
  - Tag in caches
- Physical page number
  - Frame-number in caches
- PTBR (ONE per HW thread)
  - Change value on context switch
  - Per-process page table



Page Table


Virtual page number
Page table
Physical page or disk address
Valid
1
1
1
1
0
1
1
0
1
1
0
1
Physical memory
Disk storage

Replacement

- Q3. Block replacement
  - LRU and/or LRU approximation (NRU with reference/use bits)
  - Sophisticated mechanisms possible (handling in software)
- Page-fault : Exception
  - Save instruction that causes fault
  - OS service the fault, i.e., brings in the relevant page from disk (VA-> disk address??)
  - OS knows service is slow; schedule other program
  - When disk access is complete, restart at offending instruction

Write handling

- Q4. What happens on a write
  - Write-through or write-back?
  - Write-allocate vs. write-non-allocate?

Page Table

- Managed by OS
  - Because of protection/sharing among processes
  - Also because placement and replacement all done by OS

  - Address translation via page table lookup
    - Translation is a function and a table can implement a function

Memory Organization Parameters

Parameter	I$/D$	L2	Main Memory
t_{hit}	1-2ns	5-15ns	**100ns**
t_{miss}	**5-15ns**	**100ns**	**10ms (10M ns)**
Capacity	8–64KB	256KB–8MB	**256MB–1TB**
Block size	16–32B	32–256B	**8–64KB pages**
Associativity	1–4	4–16	**Full**
Replacement Policy	LRU/NMRU	LRU/NMRU	**working set**
Write-through?	Either	No	**No**
Write-allocate?	Either	Yes	**Yes**
Write buffer?	Yes	Yes	**No**
Victim buffer?	Yes	Maybe	**No**
Prefetching?	Either	Yes	**Software**

Virtual Memory

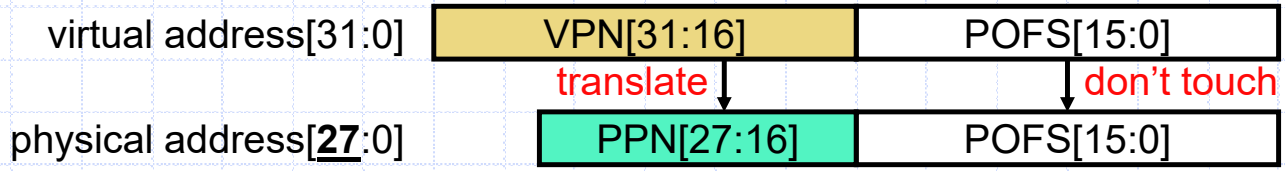
- CPU and memory are shared by processes and VM ensures protection but the sharing is slightly different
- CPU is time shared – processes run for a while before being switched out by OS
  - OS saves their registers and PC in its data structures
- Memory is space shared – processes occupy pages and their page tables sit in memory
  - This occurs even when a process is switched out of the CPU

Exercise: Virtual Memory Design

- Each process has its own page table for its virtual address space
- Page size = 8KB
- Virtual Memory
  - 64b → $2^{64} = 2^{34}$ GB!
- Physical memory = 8GB
- What is the size of the per-process page table?
  - Where does it exist?

- What happens on a context switch?

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
  - VA→PA = [VPN, POFS] → [PPN, POFS]

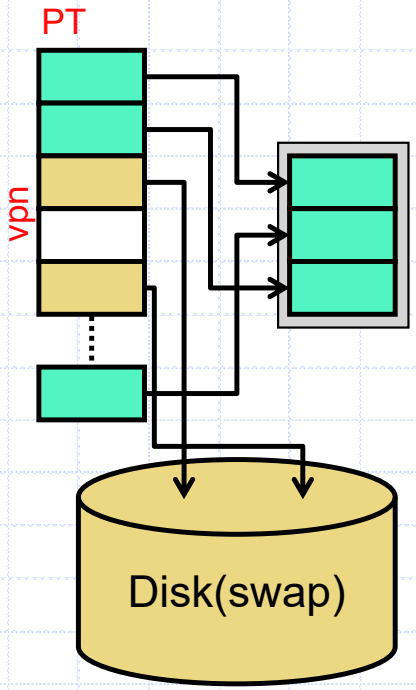
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Mechanics of Address Translation

- How are addresses translated?
  - In software (now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
  - **Managed by the operating system**
  - 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;
}
```



Address Translation Mechanics

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

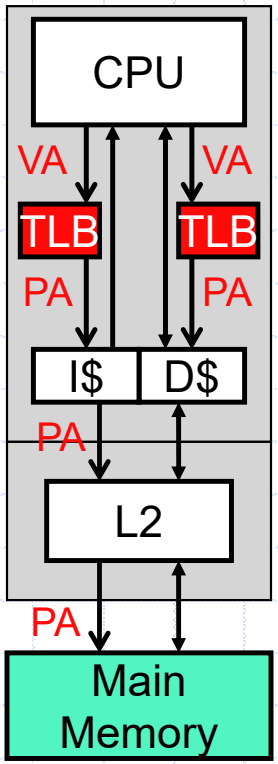
- Conceptual view:
  - Translate virtual address before every cache access
  - Walk the page table for every load/store/instruction-fetch
  - Disallow program from modifying its own page table entries

Translation

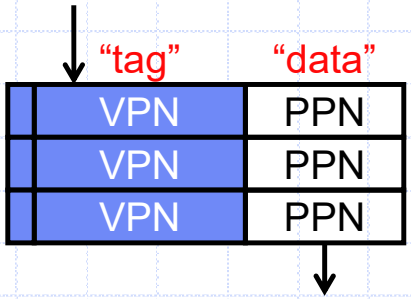
- Whoa! Wait a minute
- Where is the page table?
- In memory?
  - Then two accesses for every ld/st
  - Worse- memory access before cache access? Nonsense!

- What can we do?

Translation Lookaside Buffer



- Functionality problem? add indirection
- Performance problem? add cache
- Address translation too slow?
  - Cache translations in **translation lookaside buffer (TLB)**
    - Small cache: 16–512 entries
    - Small TLBs often fully associative (<64)
  - \+ Exploits temporal locality in page table (PT)
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler



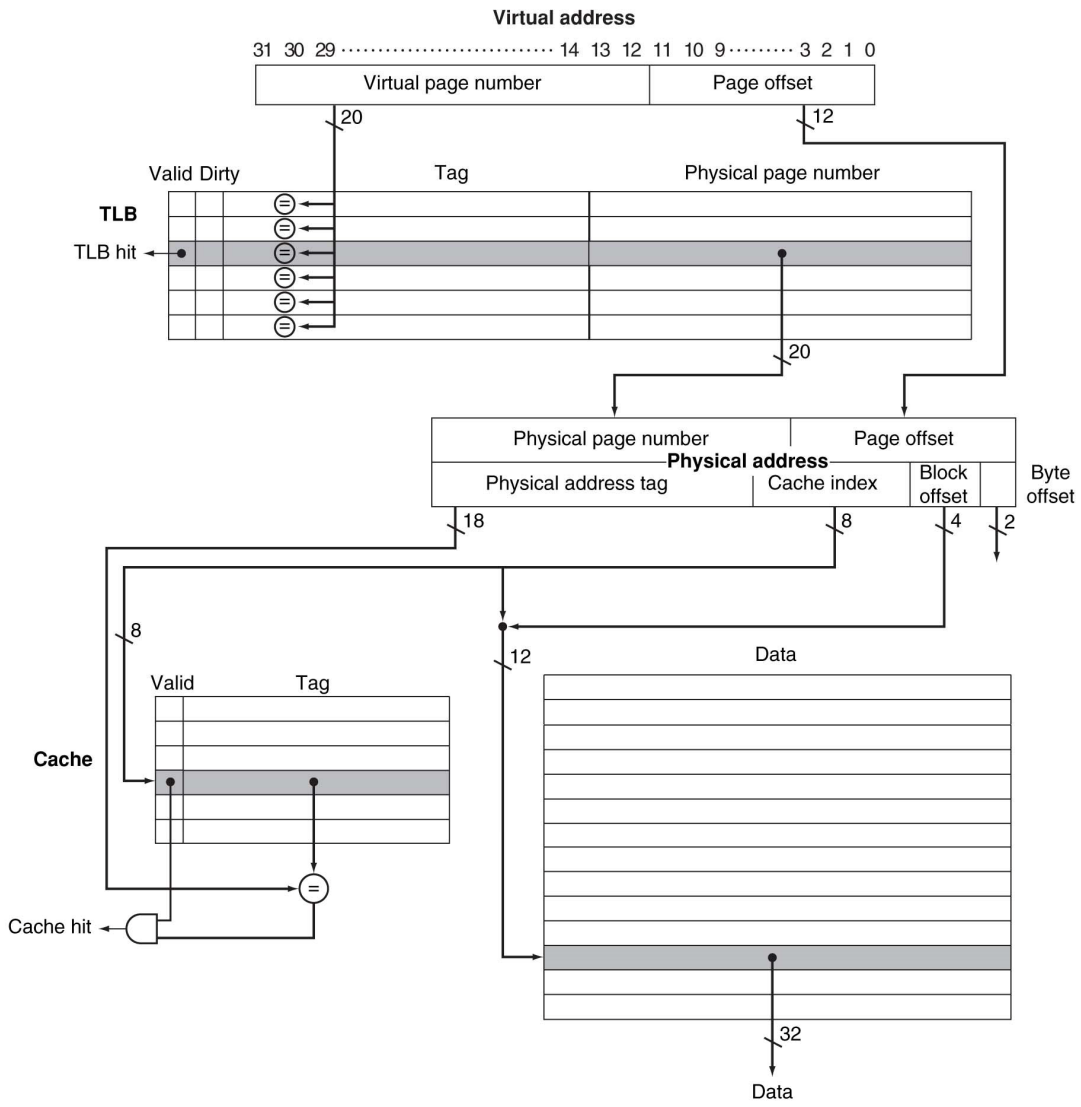
TLB

- A cache for page table
- Holds PTE for data

- What are the tag, index, block offset?

TLB+Cache

- Cache operation
  - With physical addresses
  - Translation on critical path



TLB design

- TLB's input address is virtual page number
- TLB is a cache for page table entries (PTEs) which already exploits spatial locality thru pages so TLB's block size = 1 PTE → TLB block offset is 0 bits
- TLB could be set assoc. or fully assoc.
  - VM is fully assoc. (VM miss penalty is huge) but TLB miss penalty is memory access (not huge) so TLB need not be fully assoc.
- So depending on #TLB entries and associativity, compute index & tag

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
- **Like caches**: there can be L2 TLBs
  - Why? Think about this…
- **Rule of thumb**: TLB should "cover" L2/LLC contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Think about relative miss latency in each…

Design, Design, Design

- 128-entry, 4-way set associative TLB
- 32bit VA, 40 bit PA, 8KB pages

- Practice.