U. Wisconsin CS/ECE 752 Advanced Computer Architecture I

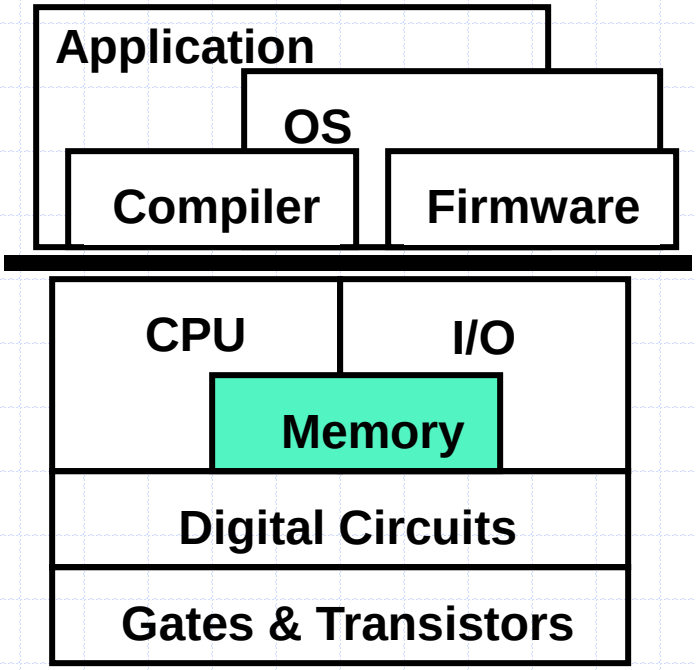
Prof. Guri Sohi

Unit 9: Memory Hierarchy II: Main Memory

Slides developed by Amir Roth of University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood.

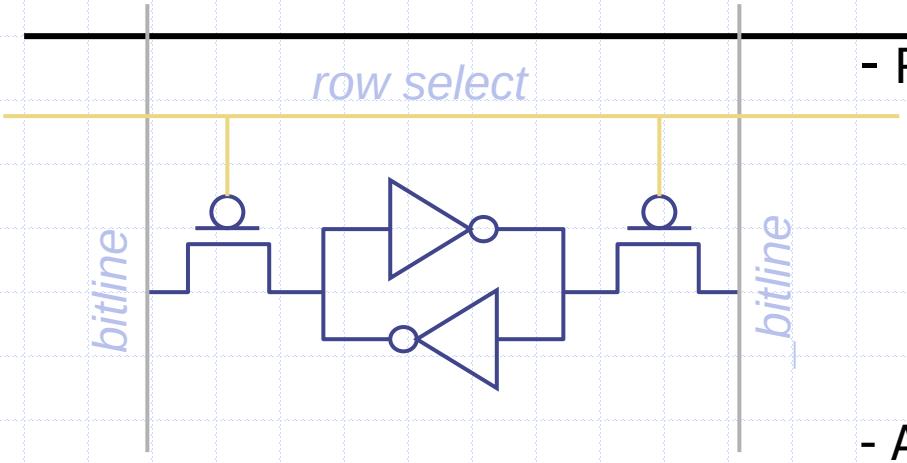
Slides enhanced by Milo Martin, Mark Hill, and David Wood with sources that included Profs. Asanovic, Falsafi, Hoe, Lipasti, Shen, Smith, Sohi, Vijaykumar, and Wood

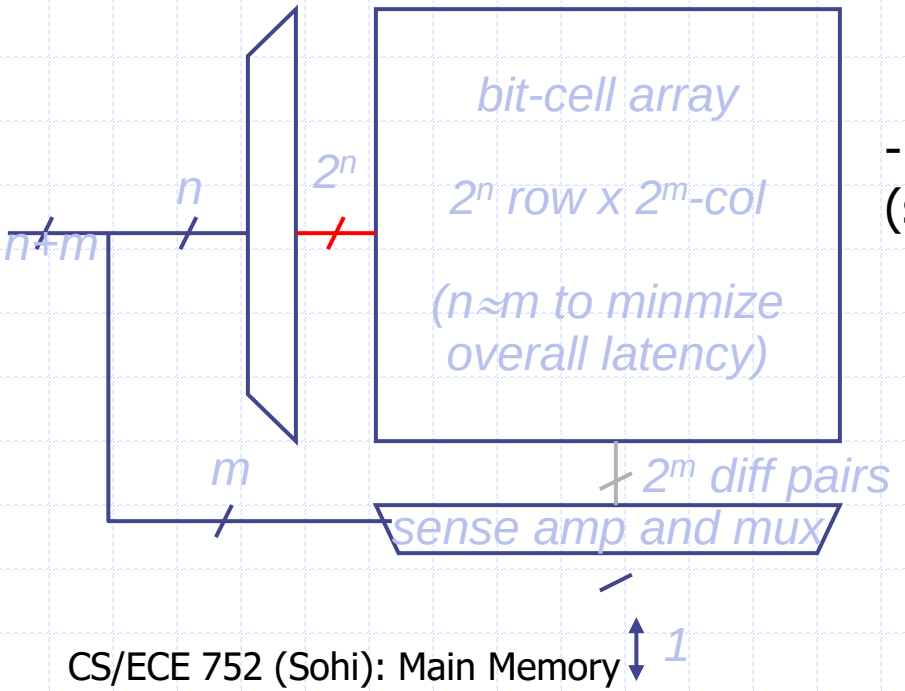
This Unit: Main Memory



- 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

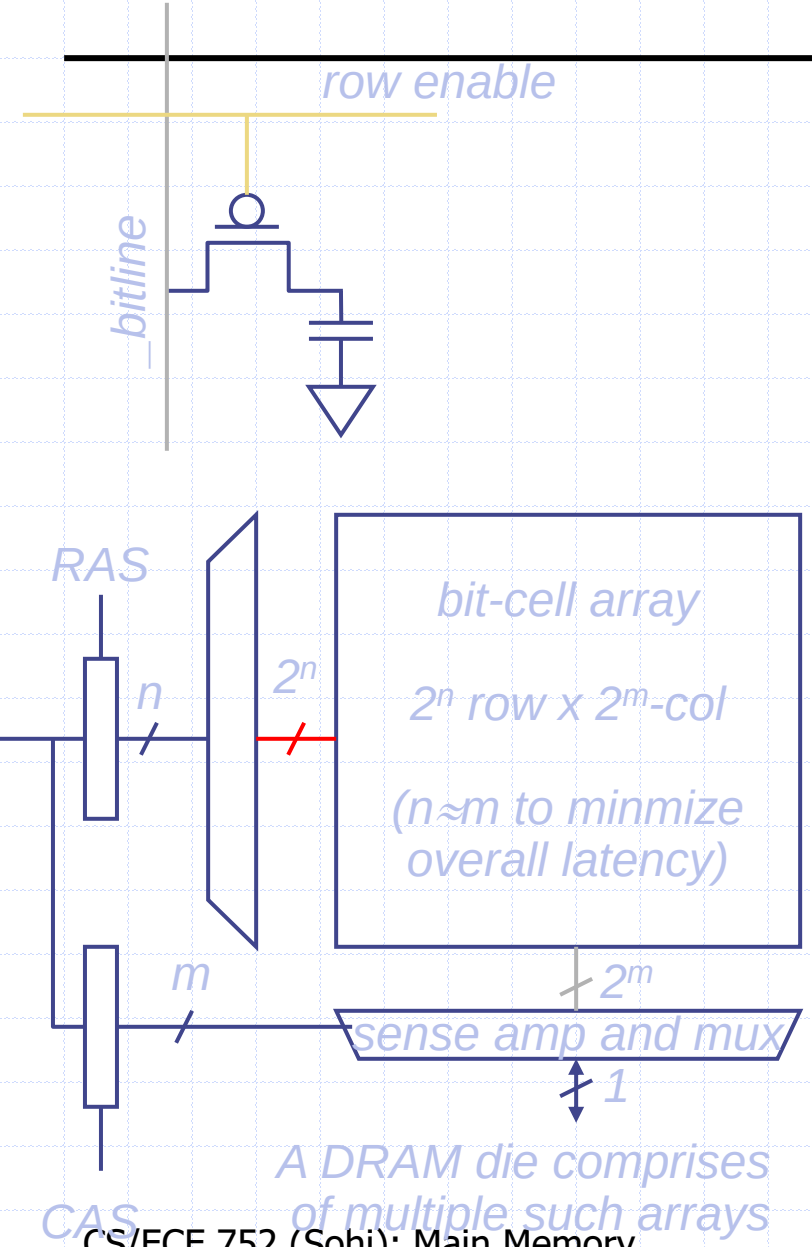
Static Random Access Memory





- 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. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out

  5. precharge all bitlines
- A DRAM controller must periodically, either distributed or in a burst, read all rows within the allowed refresh time (10s of ms) synchronous interfaces
- various hacks to allow faster repeated reads to the same 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)
    - Each cell was a small magnetic "donut"
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

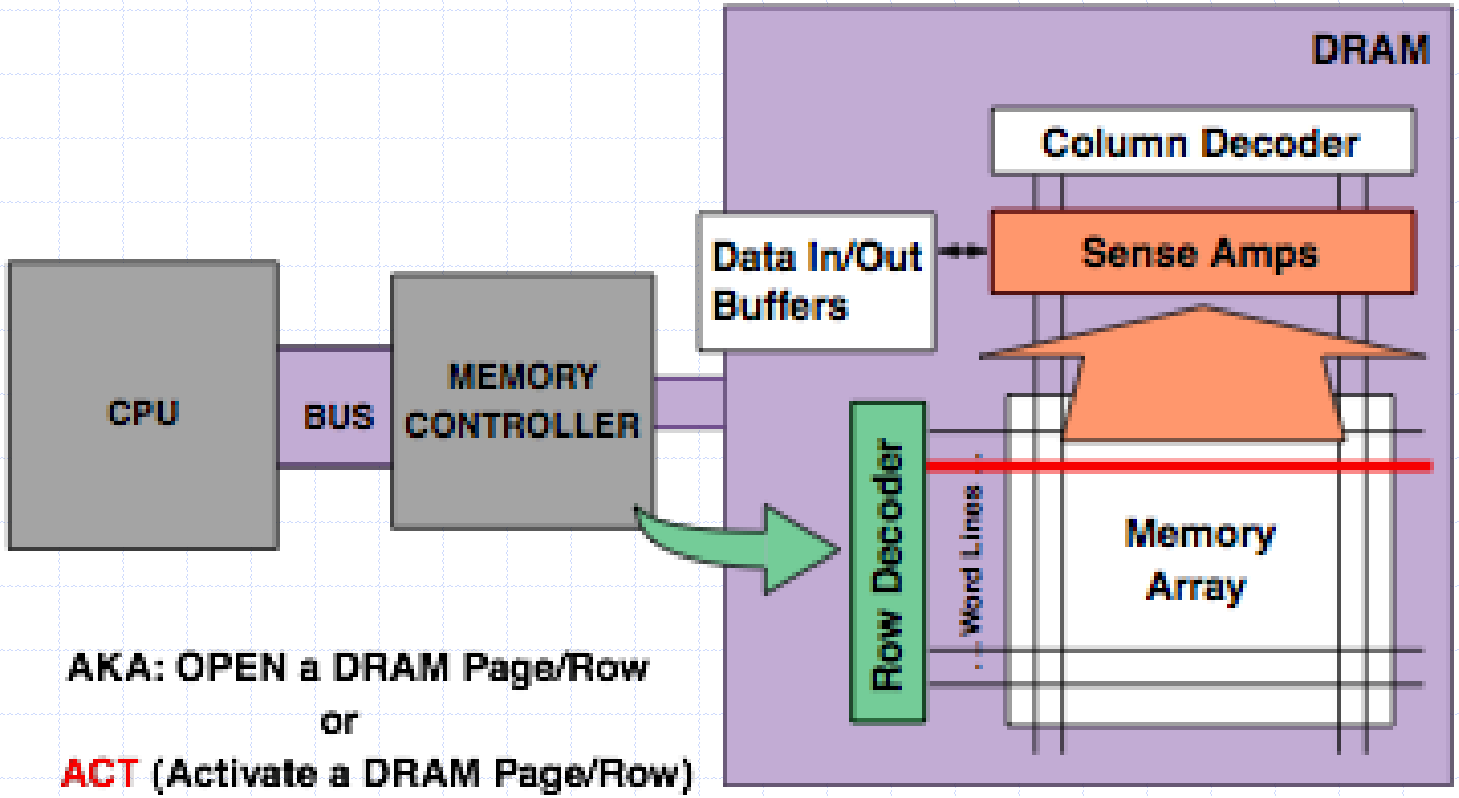
"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

Memoirs of a Computer Programmer, 1985

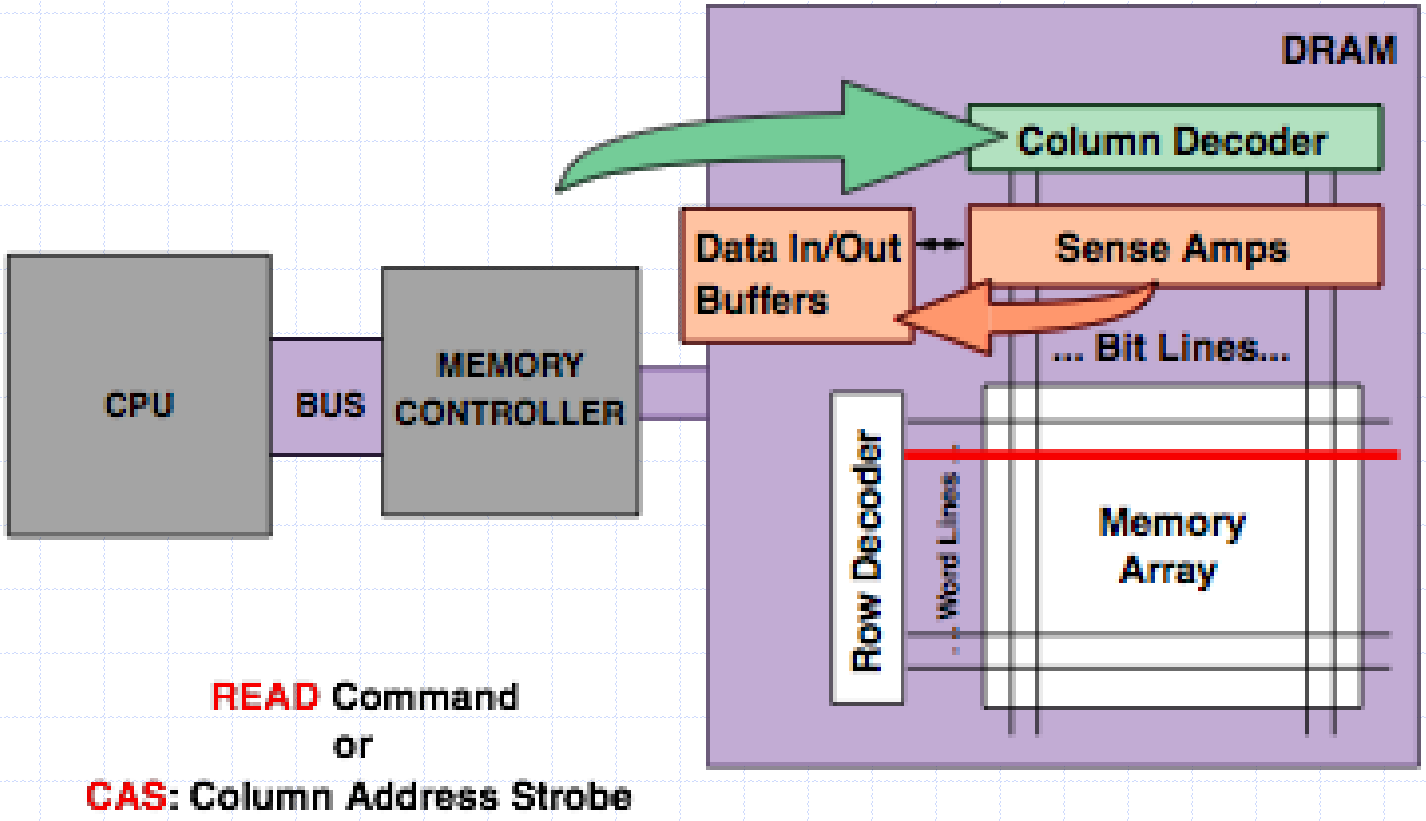
DRAM Basics [Jacob and Wang]

- Precharge and Row Access



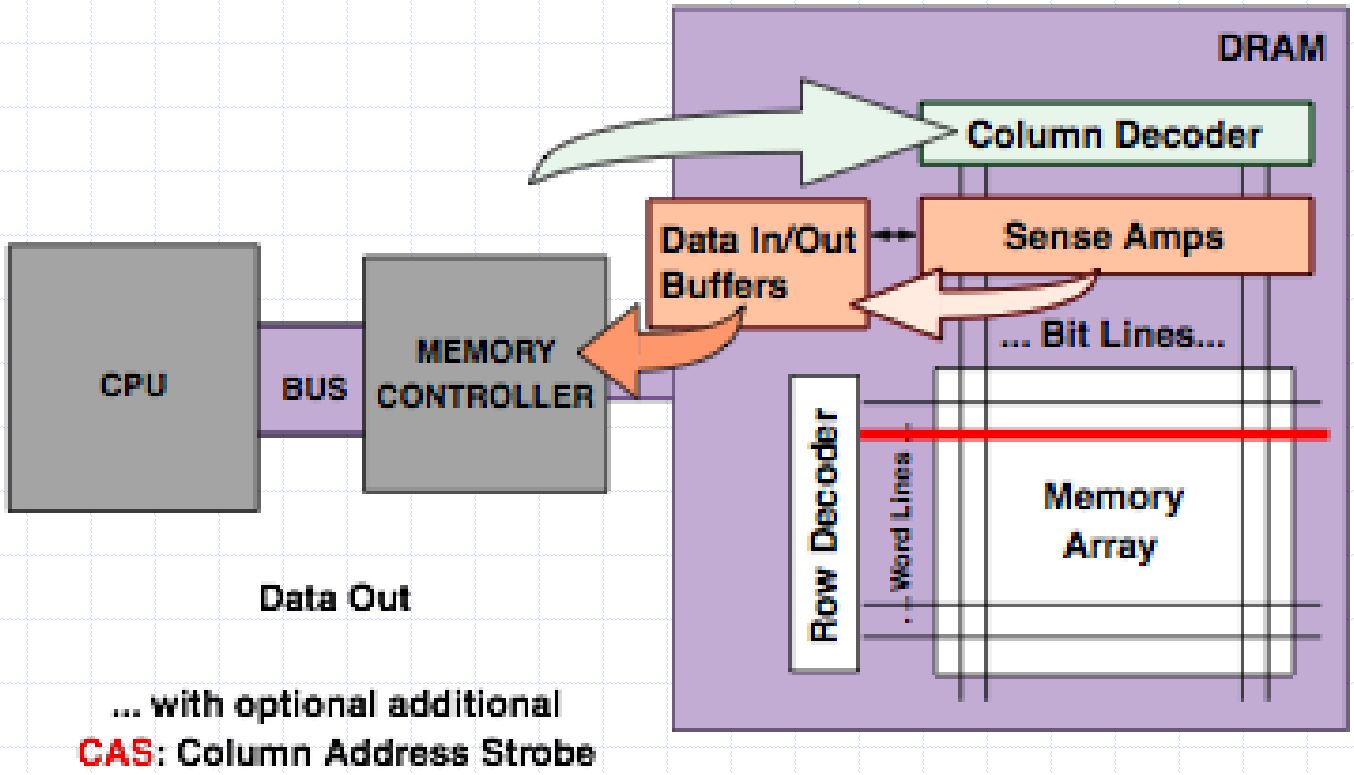
DRAM Basics, cont.

- Column Access



DRAM Basics, cont.

- Data Transfer



Open v. Closed Pages

- 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)
- Closed Page
  - Immediately deactivate row after access
  - All accesses become Activate Row, Read/Write, Precharge
- Complex power v. performance trade off

DRAM Bandwidth

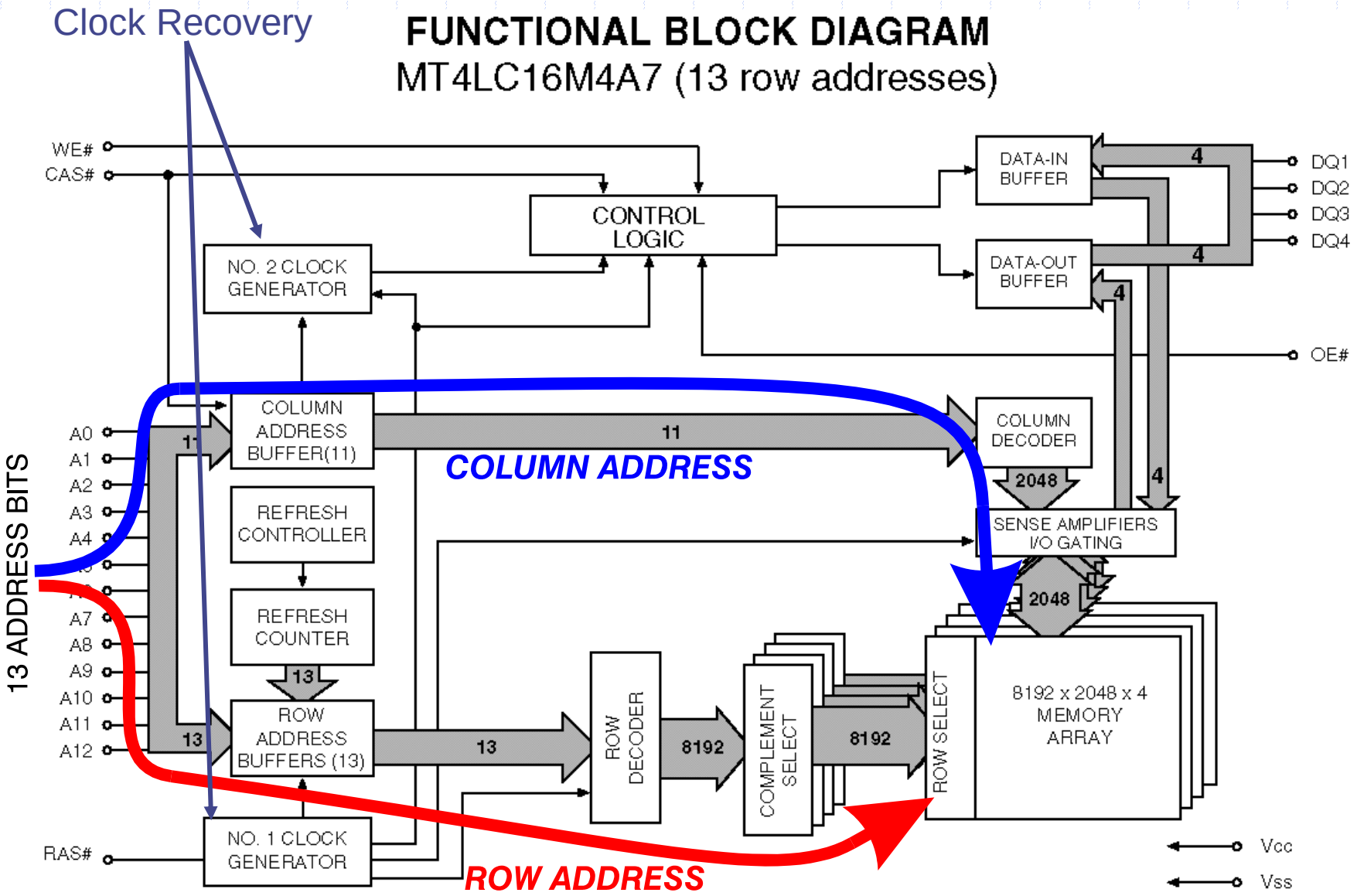
- Use multiple DRAM chips to increase bandwidth
  - Recall, access are the same size as second-level cache
  - Example, 16 2-byte wide chips for 32B access
- DRAM density increasing faster than demand
  - Result: number of memory chips per system decreasing
- Need to increase the **bandwidth per chip**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

DRAM Evolution

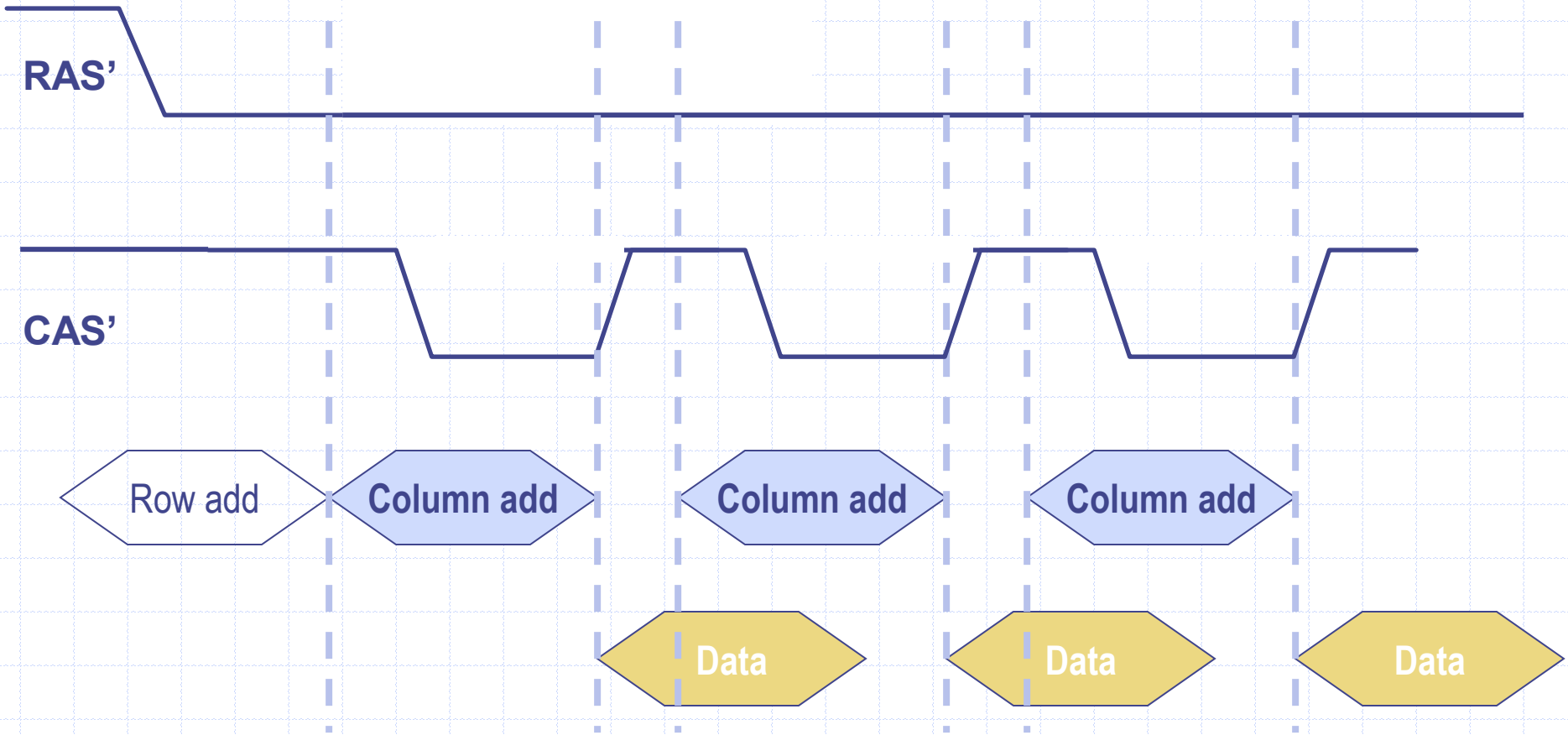
- Survey by Cuppu et al.

1. Early Asynchronous Interface
2. Fast Page Mode/Nibble Mode/Static Column (skip)
3. Extended Data Out
4. Synchronous DRAM & Double Data Rate
5. Rambus & Direct Rambus
6. FB-DIMM

Old 64MbitDRAM Example from Micron

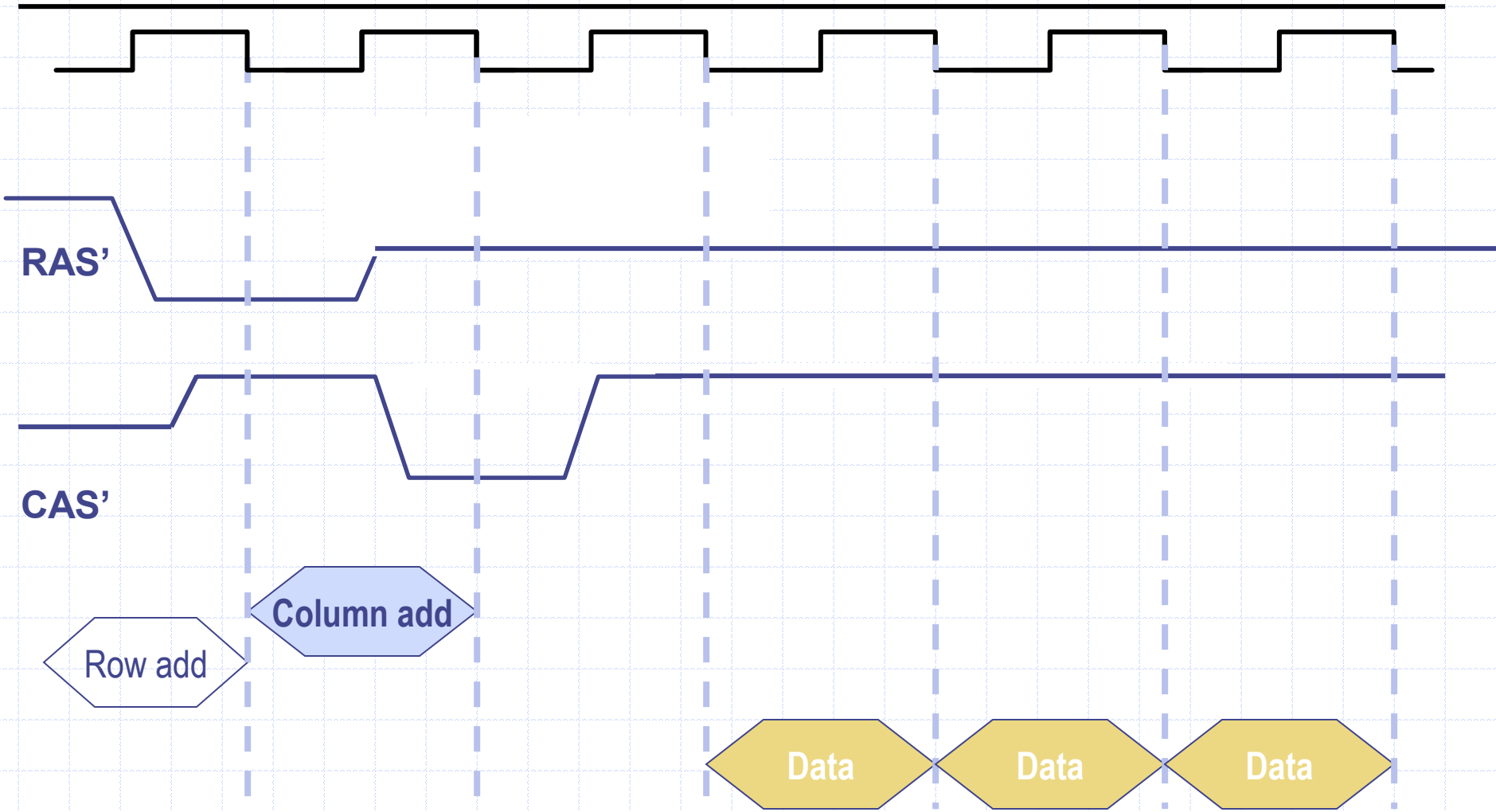


Extended Data Out (EDO)



- Similar to Fast Page Mode
- But 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:

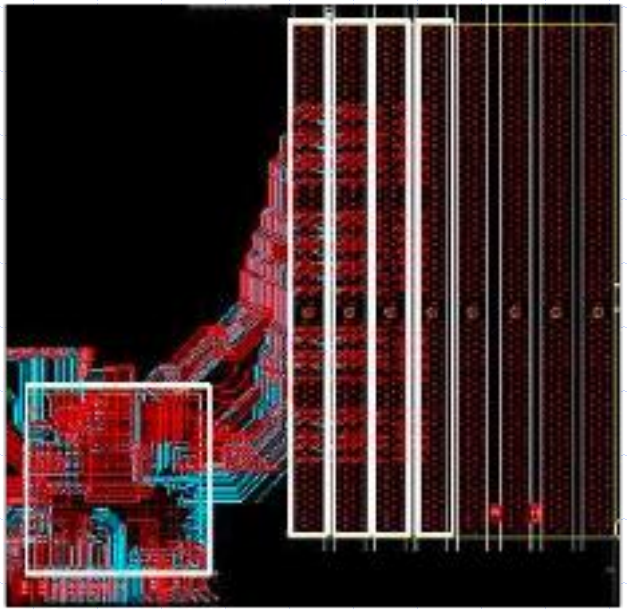
1. ESDRAM (Enhanced): Overlap row buffer access with refresh

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, "fly-by topology", automatic calibration

Wide v. Narrow Interfaces

- High frequency ➔ short wavelength ➔ data skew issues
  - Balance wire lengths

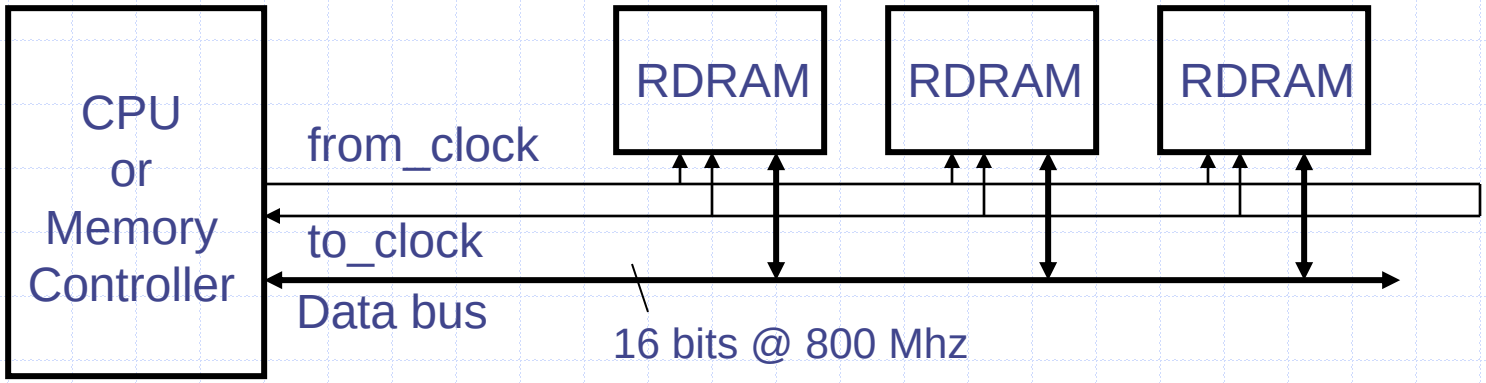
DDR-2 serpentine board routing



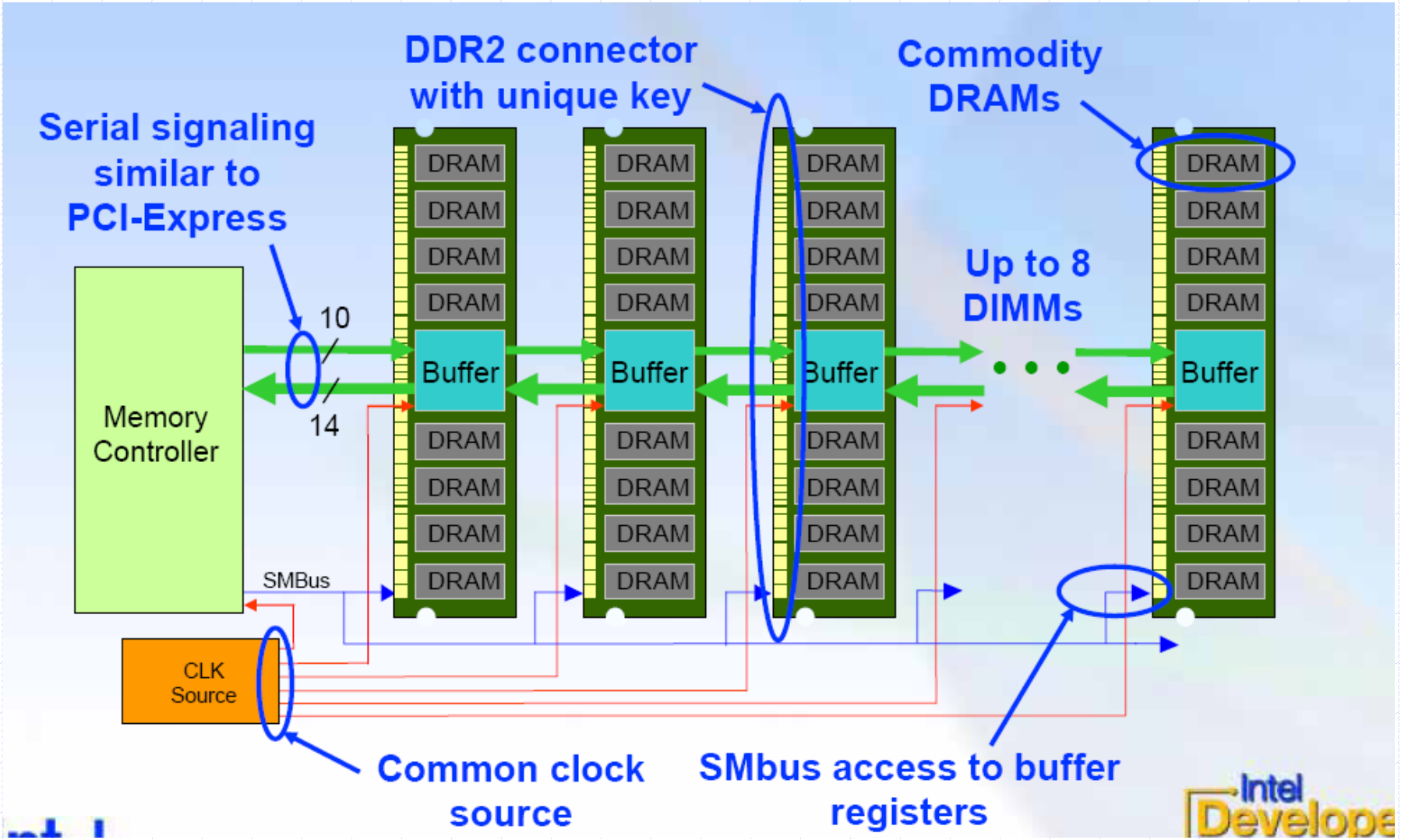
FB-DIMM board routing

Rambus RDRAM

- High-frequency, narrow channel
  - Time multiplexed "bus" ➔ dynamic point-to-point channels
  - ~40 pins ➔ 1.6GB/s
- Proprietary solution
  - Never gained industry-wide acceptance (cost and power)
  - Used in some game consoles (e.g., PS2)



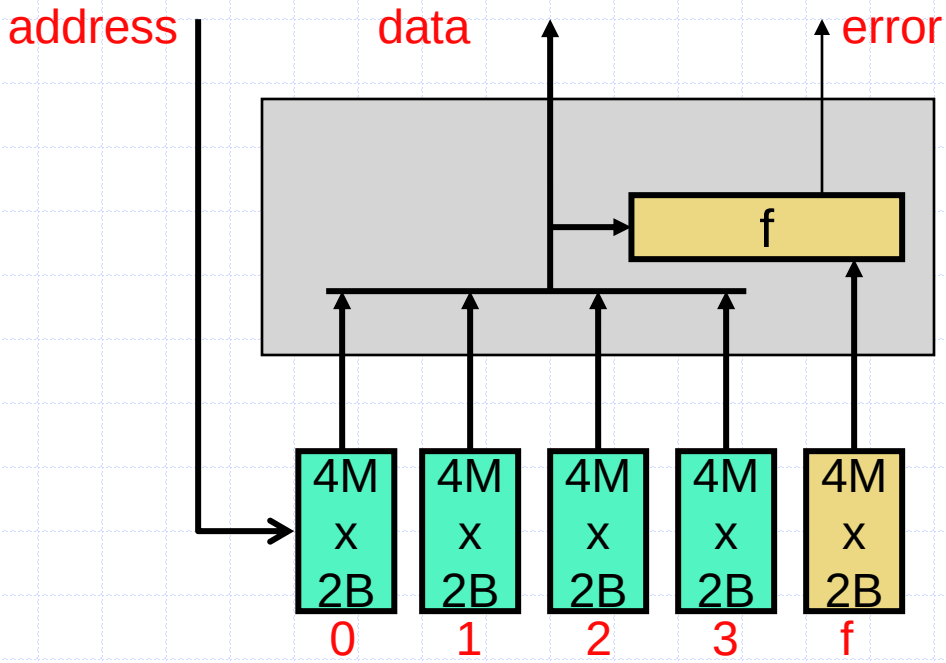
FB-DIMM


DDR2 connector with unique key
Commodity DRAMs
Serial signaling similar to PCI-Express
Up to 8 DIMMs
Memory Controller
10
14
Buffer
DRAM
SMBus
CLK Source
Common clock source
SMbus access to buffer registers

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

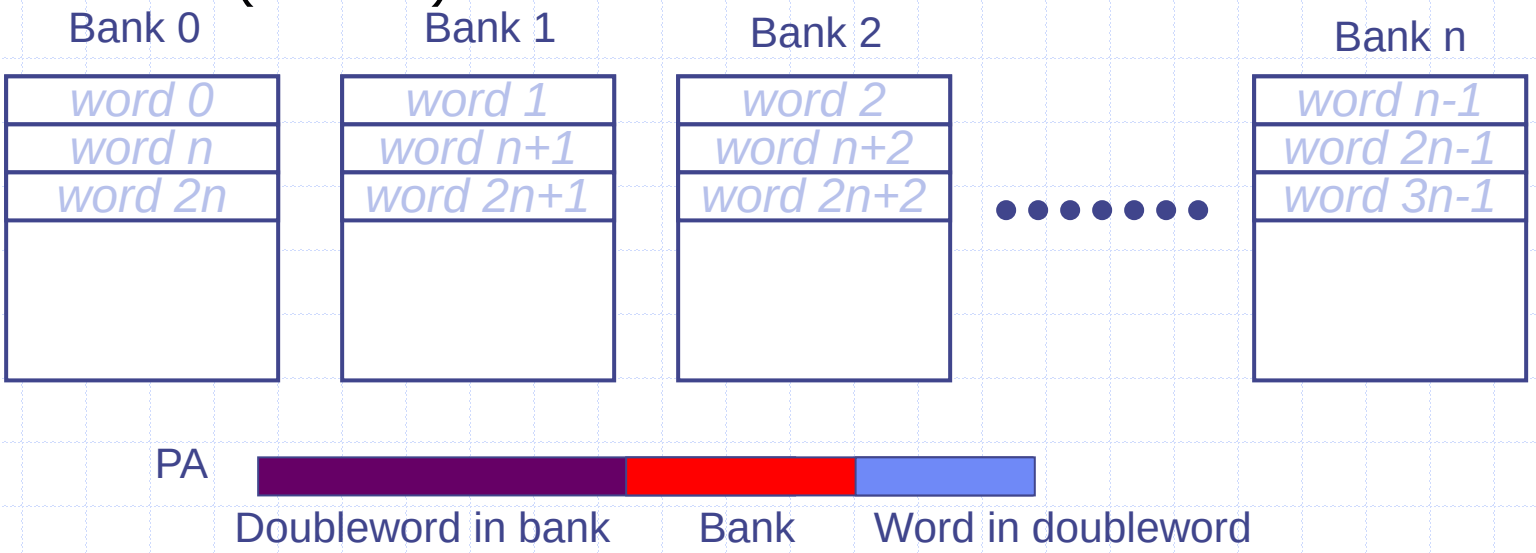
DRAM Error Detection and Correction



- 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

Interleaved Main Memory

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

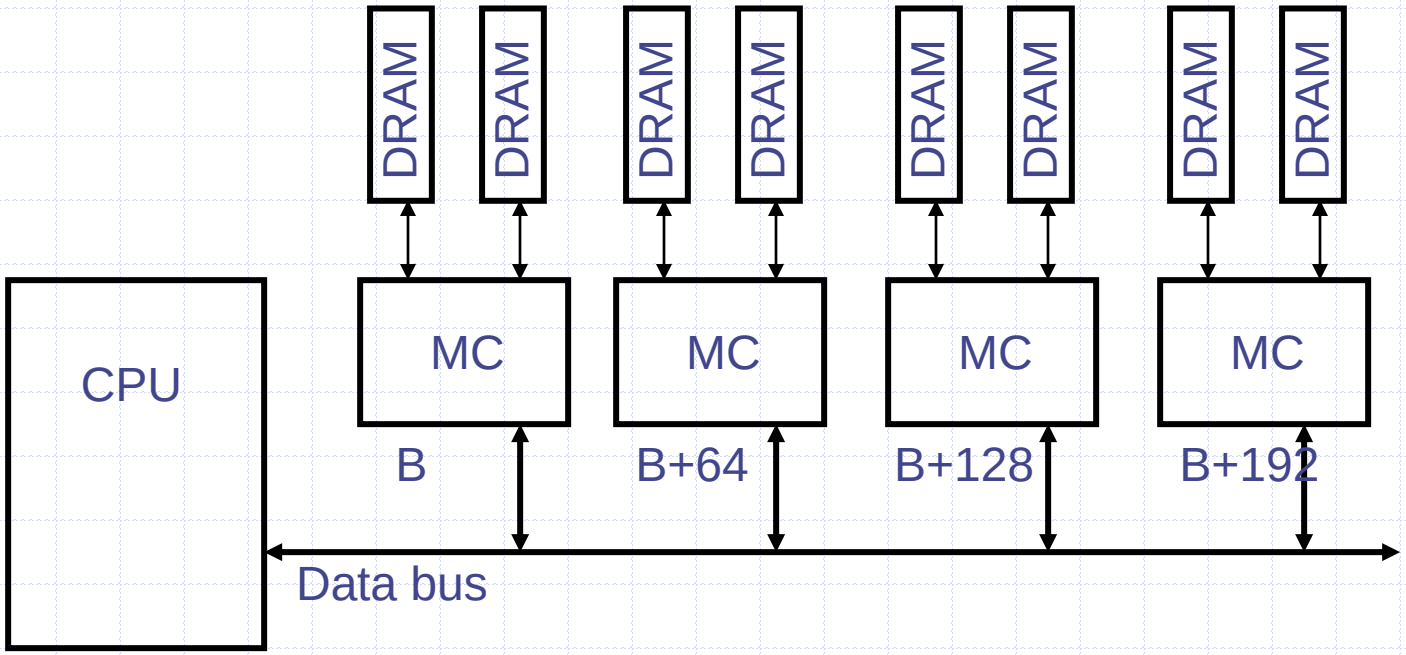


Interleaved memory increases memory BW without wider bus

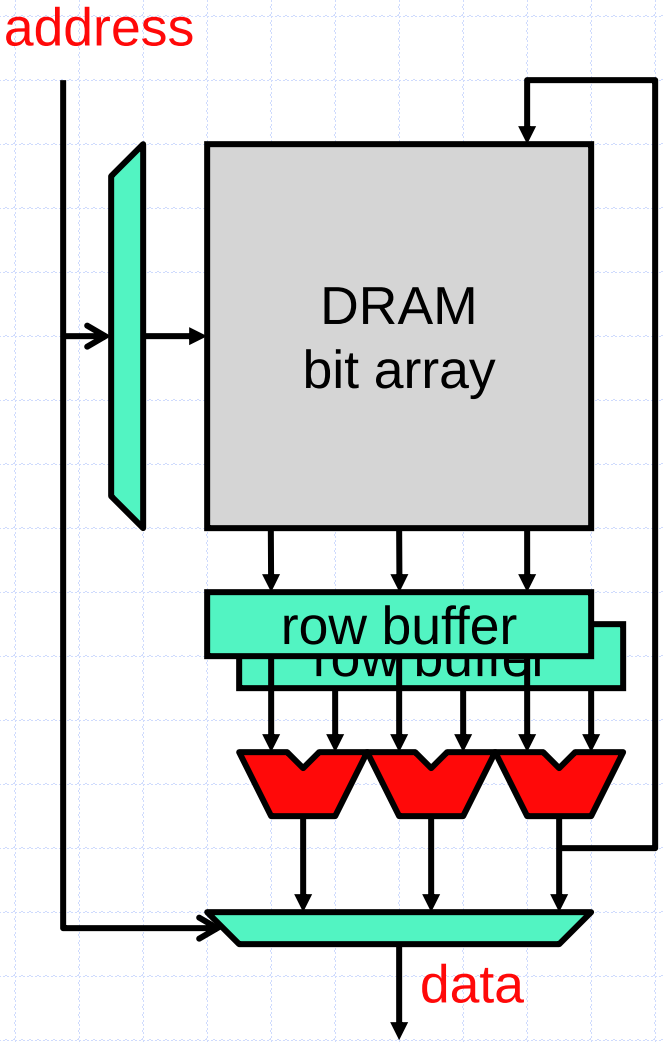
- *Use parallelism in memory banks to hide memory latency*

Block interleaved memory systems

- Cache blocks map to separate memory controllers
  - Interleave across DRAMs w/i a MC
  - Interleave across intra-DRAM banks w/i a DRAM



Research: Processing in Memory



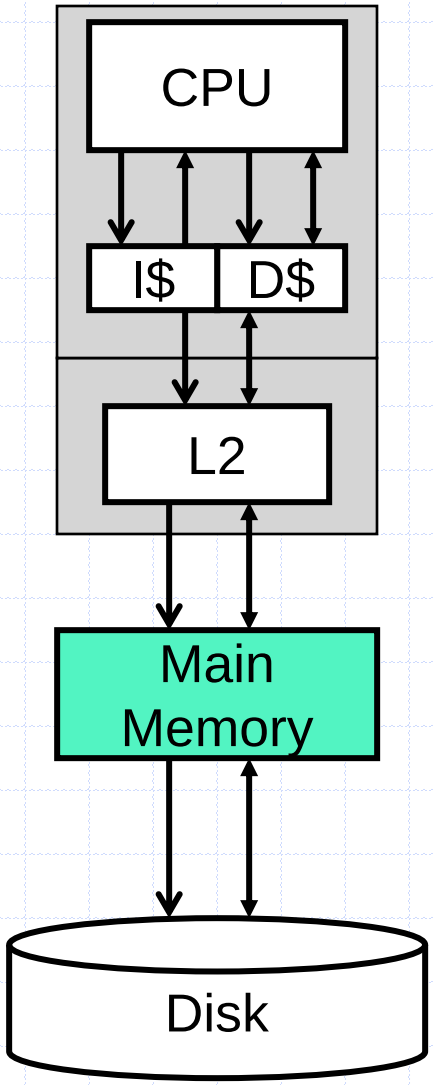
- **Processing in memory**
  - Embed some ALUs in DRAM
    - Picture is logical, not physical
  - Do computation in DRAM rather than…
    - Move data to from DRAM to CPU
    - Compute on CPU
    - Move data from CPU to DRAM
  - Will come back to this in "vectors" unit

- 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
  - This unit
- 4th level: disk (swap space)
  - Made of magnetic iron oxide discs
  - Manage in software
  - Next unit

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

- How is the "memory as a cache" organized?
  - Block size? Pages that are typically 4KB or larger
  - Associativity? Fully associative
  - Replacement policy? In software
  - Write-back vs. write-through? Write-back
  - Write-allocate vs. write-non-allocate? Write allocate

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)

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**

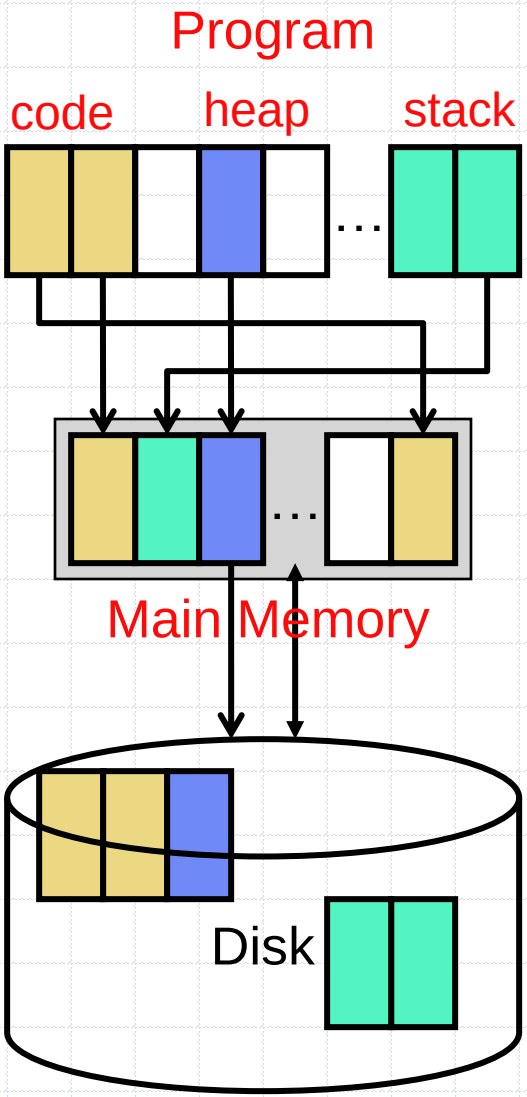
Software Managed Memory

- Isn't full associativity difficult to implement?
  - Yes … in hardware
  - Implement fully associative memory in software
- Let's take a step back…

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)

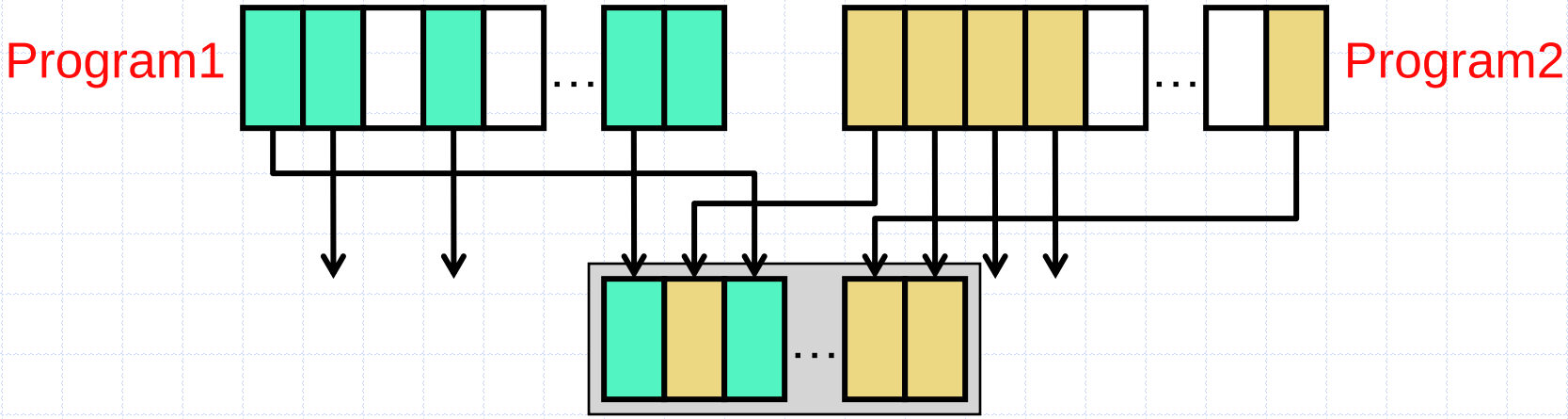
Virtual Memory



- 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)

Uses of Virtual Memory

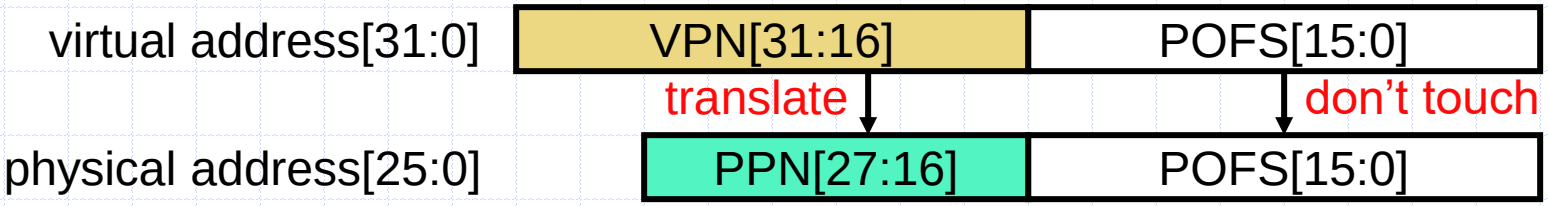
- Virtual memory is quite a useful feature
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **program isolation and multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks 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

Address Translation



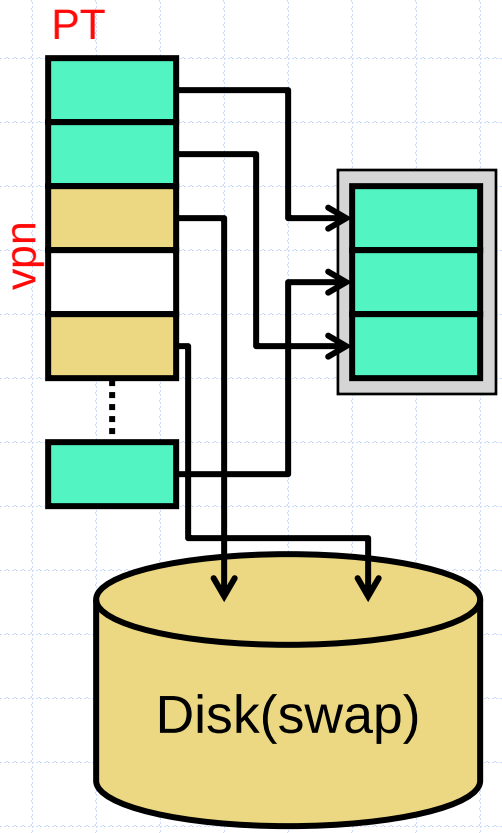
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** and page offset (POFS)
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]

- 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;
}
```



Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB

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

- Page tables can get big
  - There are ways of making them smaller
  - PA = f(VA) ➔ many different data structures possible

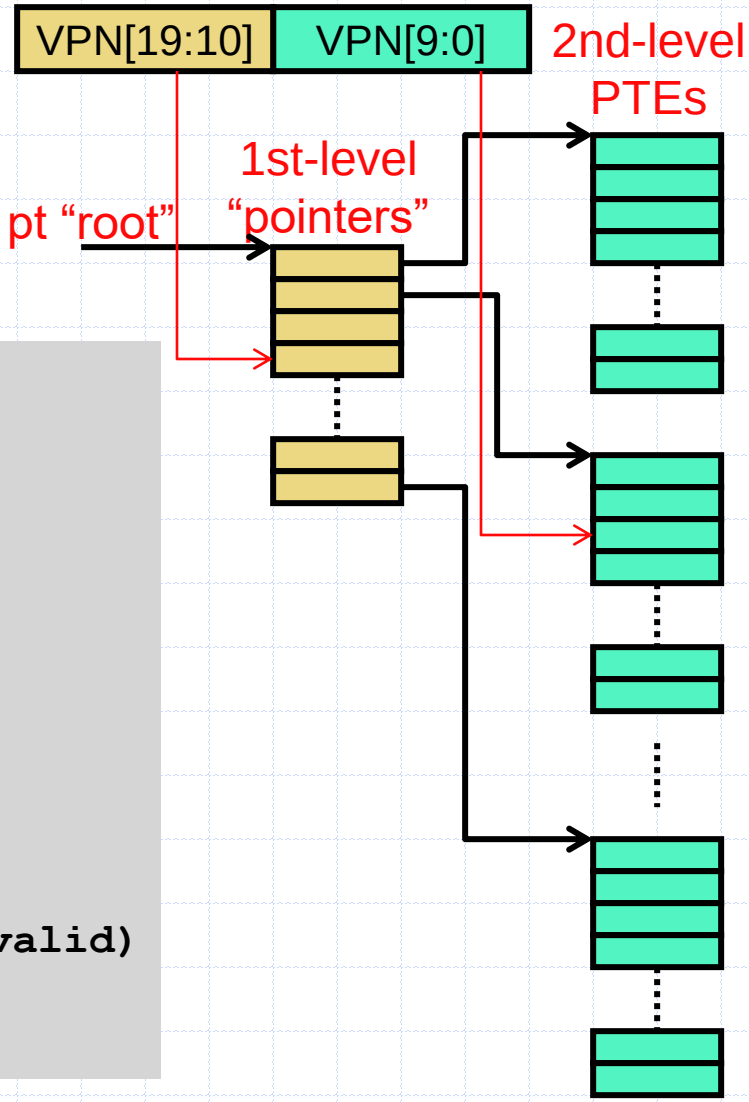
Multi-Level Page Table

- 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 $\rightarrow$ 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) $\rightarrow$ 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages $\rightarrow$ 1K pointers
    - 1K pointers * 32-bit VA $\rightarrow$ 4KB $\rightarrow$ 1 upper level page

Multi-Level Page Table

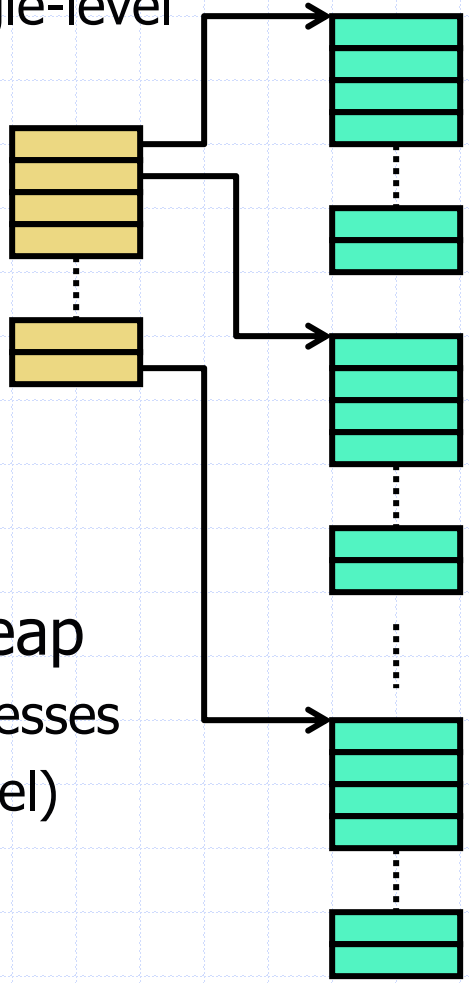
- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table



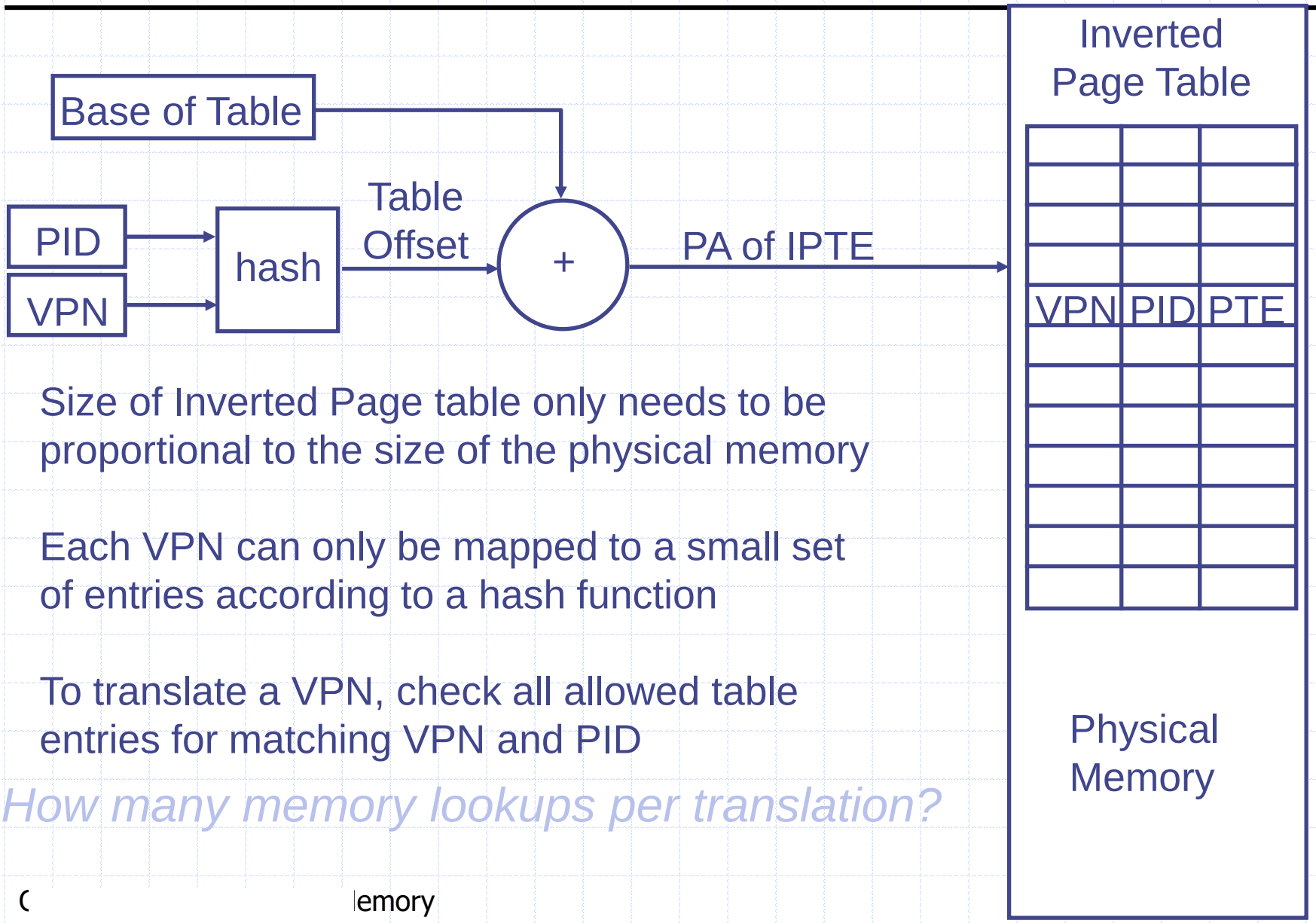

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

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 (PT)

- 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
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



Alternative: Inverted/Hashed Page Tables



Size of Inverted Page table only needs to be proportional to the size of the physical memory

Each VPN can only be mapped to a small set of entries according to a hash function

To translate a VPN, check all allowed table entries for matching VPN and PID

How many memory lookups per translation?

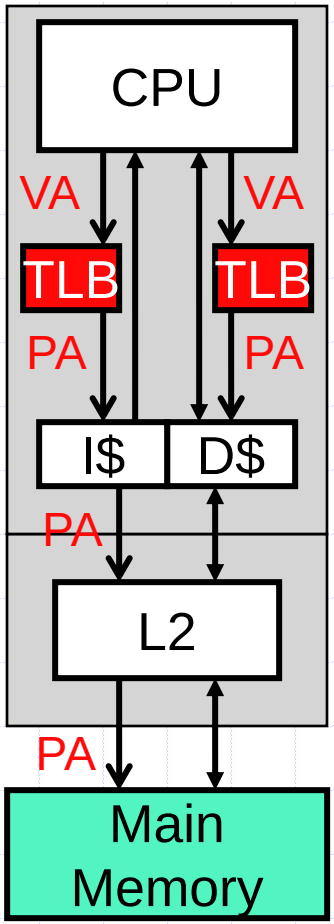
Address Translation Mechanics

- 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?**

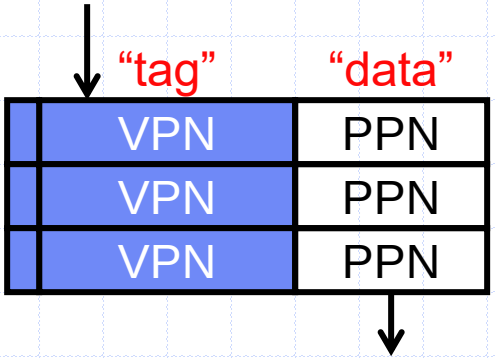
- Conceptual view:
  - Translate virtual address before every cache access
  - Walk the page table for every load/store/instruction-fetch
  - Disallow program from modifying its own page table entries

- Actual approach:
  - Cache translations in a "translation cache" to avoid repeated lookup

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 Misses and Miss Handling

- **TLB miss:** requested PTE not in TLB, search page table
  - **Software routine**, e.g., Alpha, SPARC, MIPS
    - Special instructions for accessing TLB directly
    - Latency: one or two memory accesses + trap
  - **Hardware finite state machine (FSM),** e.g., x86
    - Store page table root in hardware register
    - Page table root and table pointers are physical addresses
    + Latency: saves cost of OS call
  - In both cases, reads use the the standard cache hierarchy
    + Allows caches to help speed up search of the page table
- **Nested TLB miss**: miss handler itself misses in the TLB
  - Solution #1: Allow recursive TLB misses (very tricky)
  - Solution #2: Lock TLB entries for page table into TLB
  - Solution #3: Avoid problem using physical address in page table

TLB Performance

- TLB Reach = # TLB entries * Page size
  = 64 * 4KB = 256KB << L2 cache size

Solution #1: Big pages (e.g., 4MB)

TLB Reach = 256MB, but internal fragmentation

How to support both big and small pages?

Solution #2: Two-level TLB

L1: 64-128 entries, L2: 512-2048 entries

Solution #3: Software TLB (aka TSB)

in memory TLB: 32K entries (or more)

low-associativity (e.g., 2-way), longer hit time

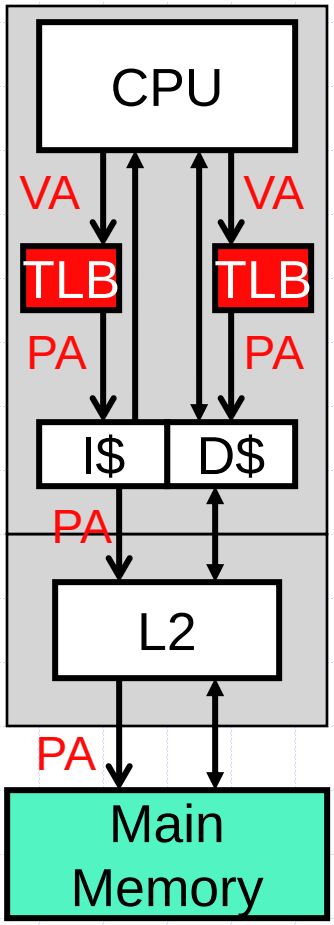
Much faster than page table access

Page Faults

- **Page fault**: PTE not in page table
  - Page is simply not in memory
  - Starts out as a TLB miss, detected by OS handler/hardware FSM

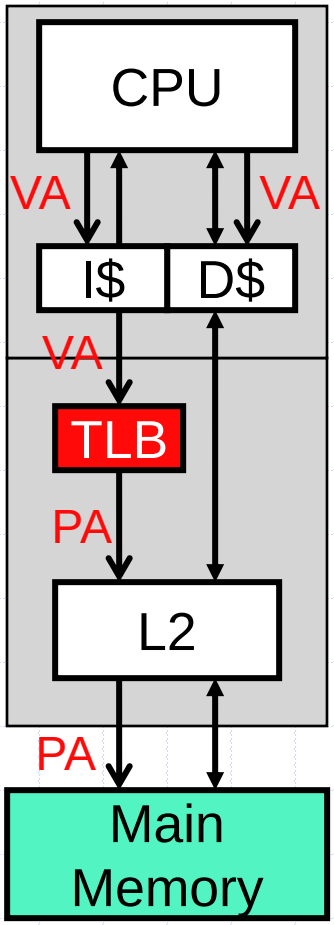
- **OS routine**
  - Choose a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Treat like a normal TLB miss from here

Physical (Address) Caches



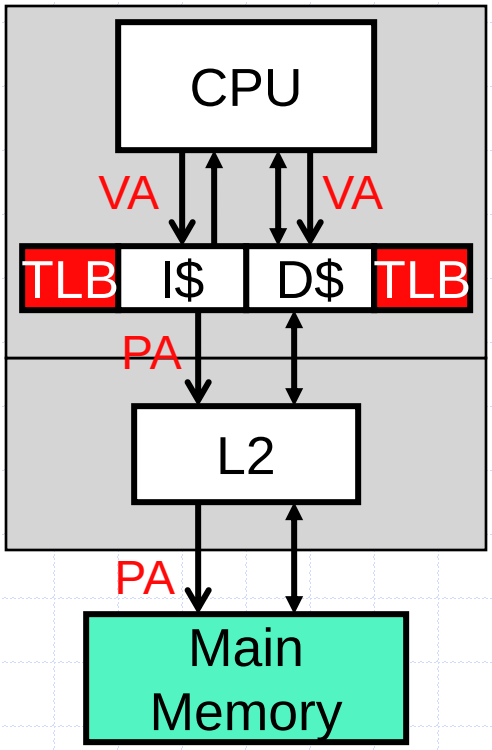
- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by Pas
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA to VA at the outset
  + Cached inter-process communication works
    - Single copy indexed by PA
  – Slow: adds at least one cycle to t_{hit}

Virtual Address Caches (VI/VT)



- Alternative: **virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2
  - \+ Fast: avoids translation latency in common case
  - – Problem: VAs from ***different processes*** are distinct physical locations (with different values) (call homonyms)
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    - Can't allow same PA in the cache twice
    - Also a problem for DMA I/O
  - Can be handled, but very complicated

Parallel TLB/Cache Access (VI/PT)



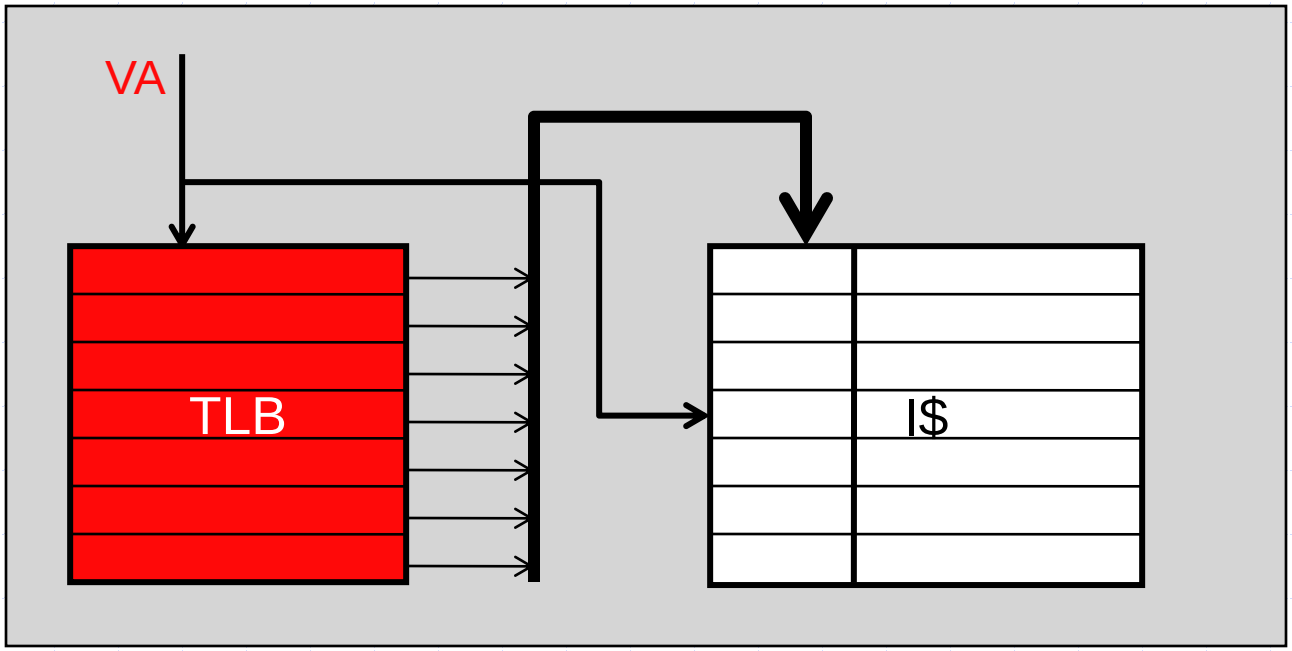
Compromise: **access TLB in parallel**

- *In small caches, index of VA and PA the same*
  - *VI == PI*
- Use the VA to index the cache
- Tagged by PAs
- Cache access and address translation in parallel

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

- Common organization in processors today

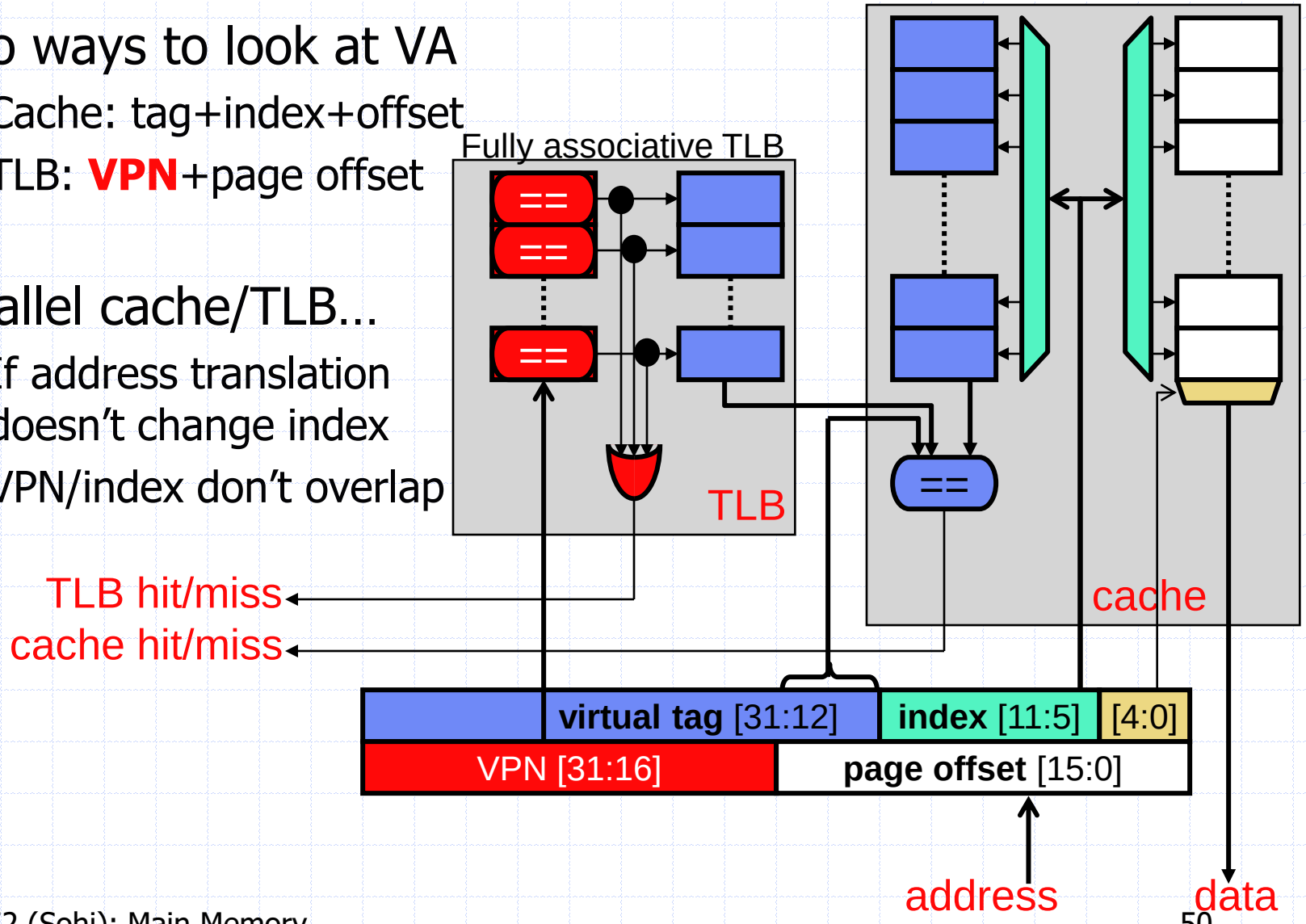
Itanium Prevalidated tags



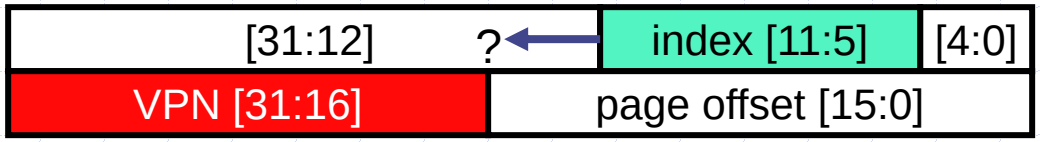
- I$ tag is bit vector, not address tag
  - match TLB location for hit
- TLB miss ➔ I$ miss
- TLB size ➔ tag size (32 entries/32 bits in Itanium 2)

Parallel Cache/TLB Access

- Two ways to look at VA
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset
- Parallel cache/TLB…
  - If address translation doesn't change index
  - VPN/index don't overlap



Cache Size And Page Size



- Relationship between page size and L1 cache size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - **Rule: (cache size) / (associativity) ≤ page size**
  - Result: associativity increases allowable cache sizes
  - Systems are moving towards bigger (64KB) pages
    - To use parallel translation with bigger caches
    - To amortize disk latency
  - Example: Pentium 4, 4KB pages, 8KB, 2-way SA L1 data cache
- If cache is too big, same issues as virtually-indexed caches
  - Other tricks can help (e.g., set-associative main memory)

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 contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Think about relative miss latency in each…

Virtual Memory

- Virtual memory ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even many 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
    - x86 used this rather than 32-bits to break 16-bit (64KB) limit
    - Makes life hell
  - **Paged segments**: don't ask
- How does virtual memory work when system starts up?

Memory Protection and Isolation

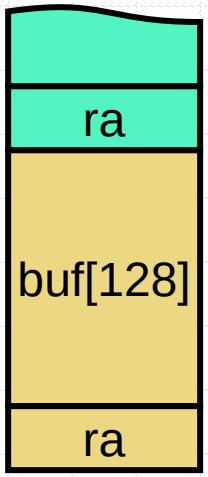
- Most important role of virtual memory today

- Virtual memory protects applications from one another
  - OS uses indirection to isolate applications
  - One buggy program should not corrupt the OS or other programs
  + Comes "for free" with translation
  – However, the protection is limited

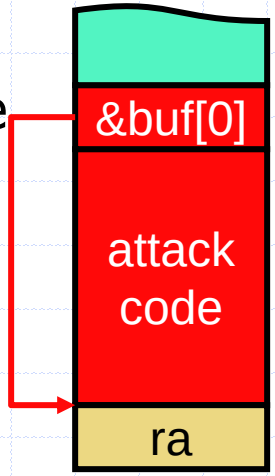
– What about protection from…
    - Viruses and worms?
      - Stack smashing
    - Malicious/buggy services?
      - Other applications with which you want to communicate

Stack Smashing via Buffer Overflow

```
int i = 0;
char buf[128];
while ((buf[i++] = getc()) != '\n') ;
return;
```



- Stack smashing via buffer overflow
  - Oldest trick in the virus book
  - Exploits stack frame layout and…
  - Sloppy code: **length-unchecked copy to stack buffer**
  - "Attack string": **`code`** (128B) + **`&buf[0]`** (4B)
  - Caller return address replaced with pointer to attack code
    - Caller return…
    - …executes attack code at caller's privilege level
  - Vulnerable programs: gzip-1.2.4, sendmail-8.7.5



Page-Level Protection

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
```

- **Page-level protection**
  - Piggy-backs on translation infrastructure
  - Each PTE associated with permission bits: **R**ead, **W**rite, e**X**ecute
    - **Read/execute (RX)**: for code
    - **Read (R)**: read-only data
    - **Read/write (RW)**: read-write data
  - TLB access traps on illegal operations (e.g., write to **RX** page)
  - To defeat stack-smashing? Set stack permissions to **RW**
    - Will trap if you try to execute `&buf[0]`

  \+ **X** bits recently added to x86 for this specific purpose

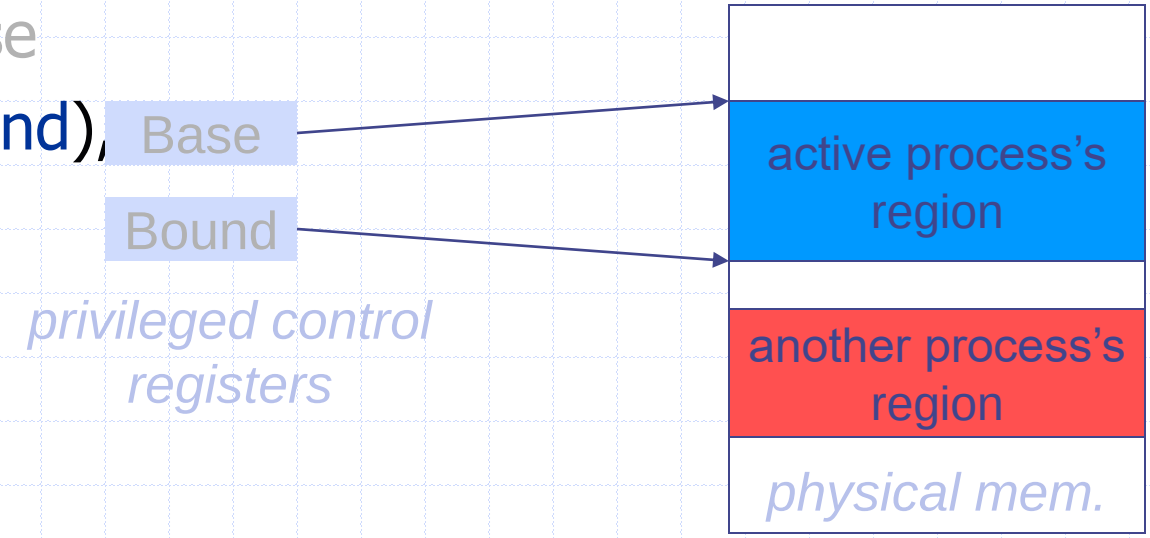
  – Unfortunately, hackers have many other tricks

Safe and Efficient Services

- Scenario: module (application) A wants service B provides
  - A doesn't "trust" B and vice versa (e.g., B is kernel)
  - How is service provided?
- Option I: conventional call in same address space
  - \+ Can easily pass data back and forth (pass pointers)
  - – Untrusted module can corrupt your data
- Option II: trap or cross address space call
  - – Copy data across address spaces: slow, hard if data uses pointers
  - \+ Data is not vulnerable
- Page-level protection helps somewhat, but…
  - Page-level protection can be too coarse grained
  - If modules share address space, both can change protections

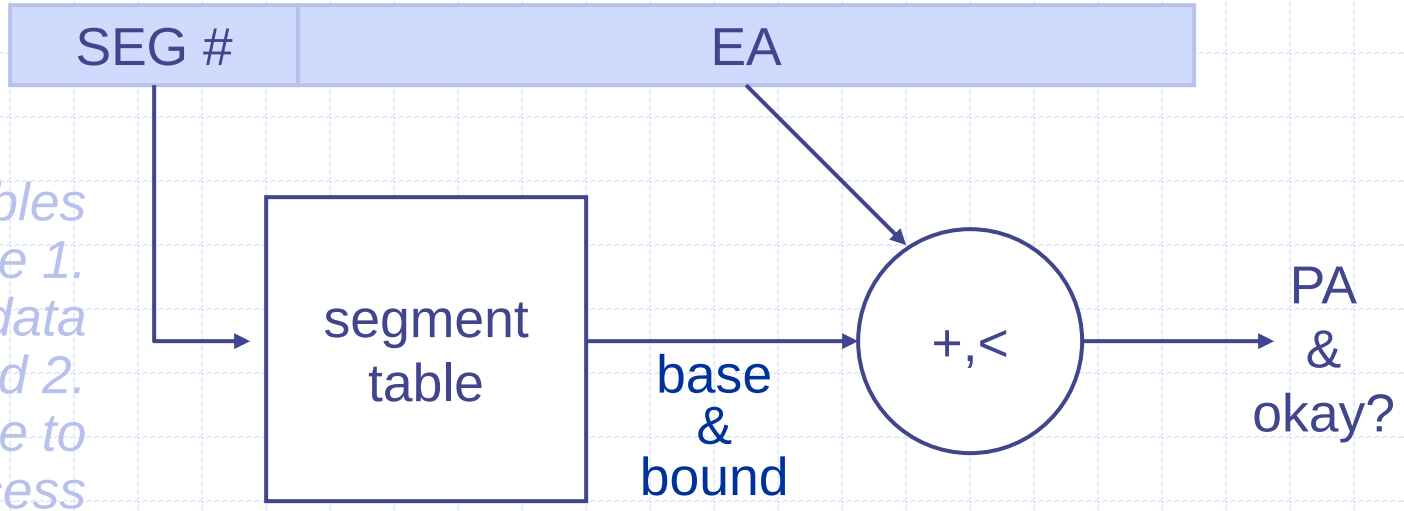
Alternative to VM: base/bound registers

- Each process is given a non-overlapping, contiguous physical memory region
- When a process is swapped in, OS sets base to the start of the process's memory region and bound to the end of the region
- On memory references, HW translation & protection check
- PA = EA + base
- provided (PA < bound),
- else violations



Also Segmented Address Space

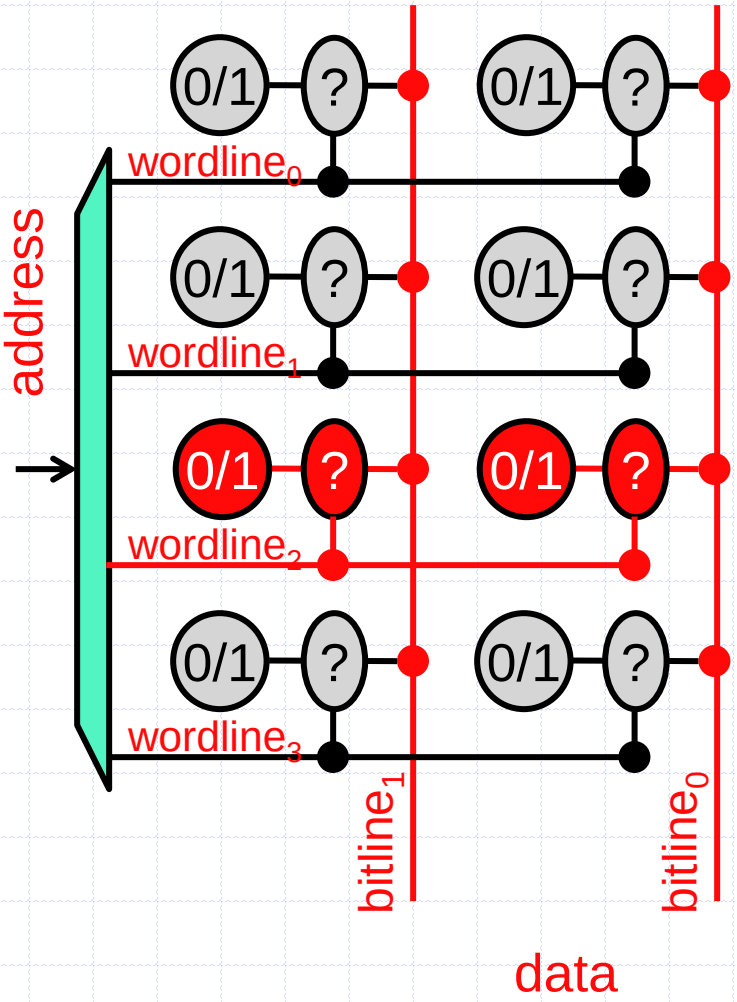
- ~~segment == a base and bound pair~~
- segmented addressing gives each process multiple segments
  - initially, separate code and data segments
    - *2 sets of base-and-bound reg's for inst and data fetch*
    - *allowed sharing code segments*
  - became more and more elaborate: *code, data, stack, etc.*
  - also (ab)used as a way for an ISA with a small EA space to address a larger physical memory space



segment tables must be 1. privileged data structures and 2. private/unique to each process

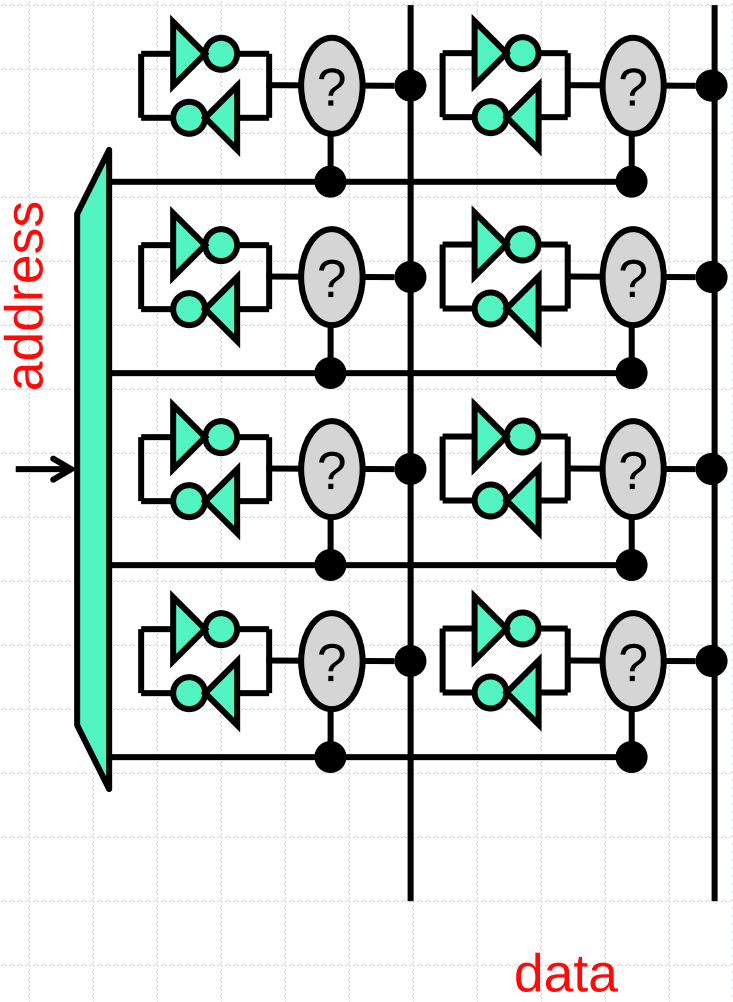
Backups

RAM



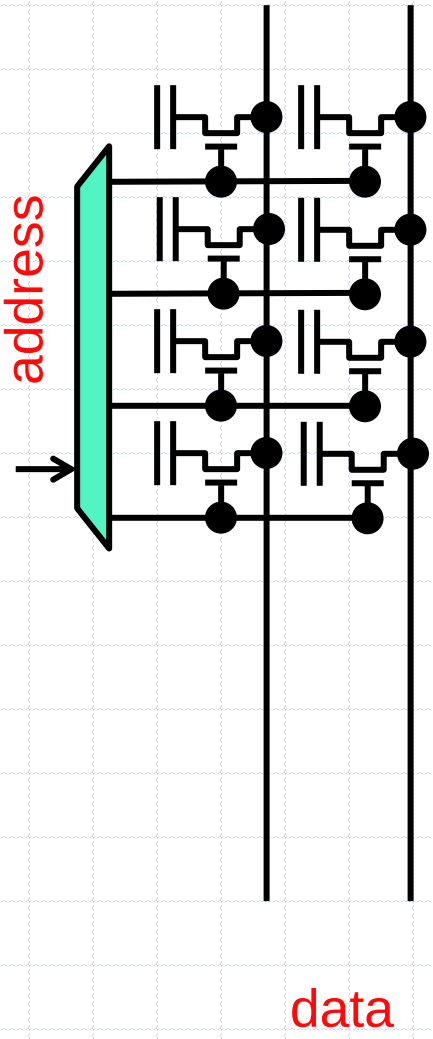
- RAM: large storage arrays
- Basic structure
  - MxN array of bits (M N-bit words)
    - This one is 4x2
  - Bits in word connected by **wordline**
  - Bits in position connected by **bitline**
- Operation
  - Address decodes into M wordlines
  - High wordline → word on bitlines
  - Bit/bitline connection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

SRAM



- **SRAM**: static RAM
  - Bits as cross-coupled inverters (CCI)
  - – Four transistors per bit
  - – More transistors for ports
- **"Static"** means
  - Inverters connected to pwr/gnd
  - \+ Bits naturally/continuously "refreshed"
- Designed for speed

DRAM



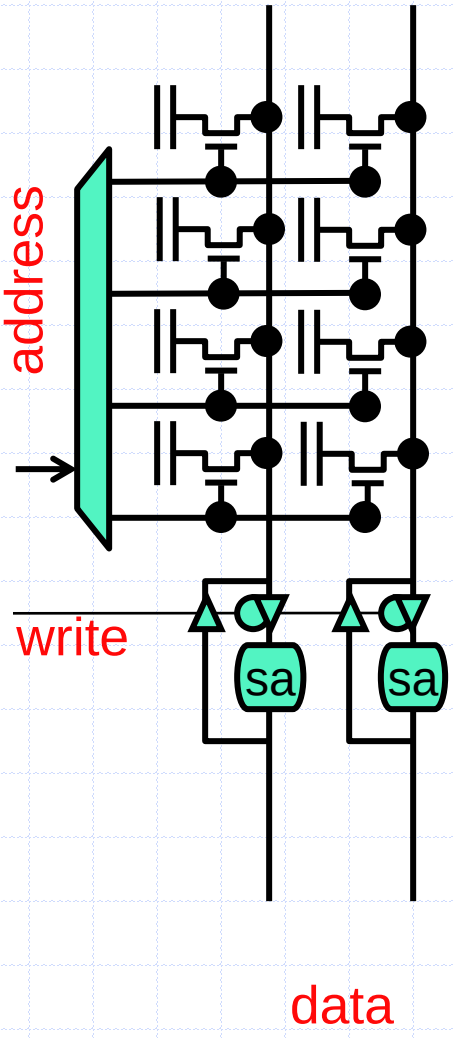
- **DRAM**: dynamic RAM
  - Bits as capacitors
  + Single transistors as ports
  + One transistor per bit/port
- **"Dynamic"** means
  - Capacitors not connected to pwr/gnd
  – Stored charge decays over time
  – Must be explicitly refreshed
- Designed for density
  - Moore's Law

Moore's Law

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

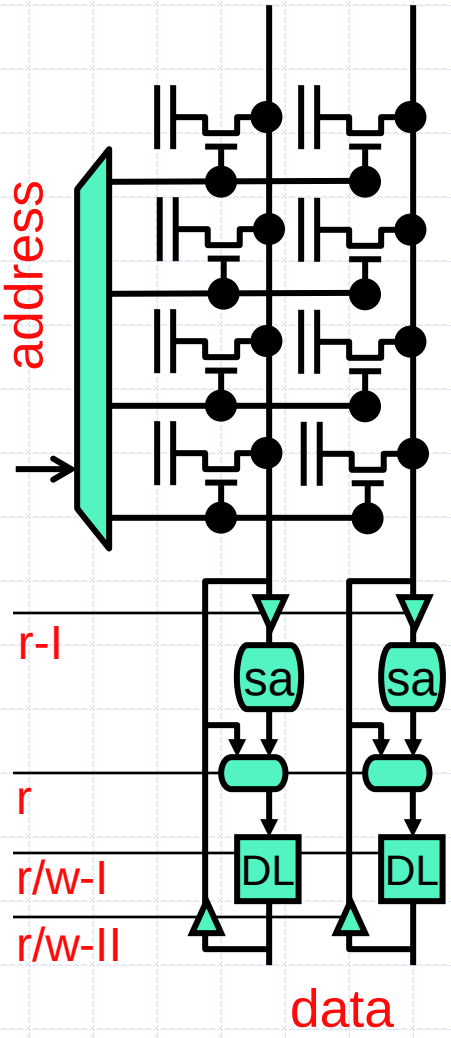
- Commodity DRAM parameters
  - 16X every 8 years is 2X every 2 years
    - Not quite 2X every 18 months but still close

DRAM Operation I



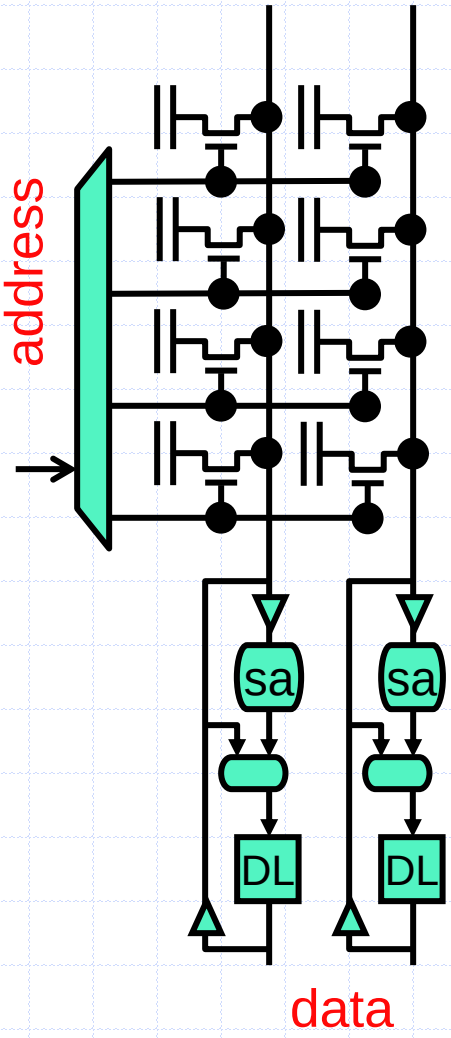
- Read: similar to cache read
  - Phase I: pre-charge bitlines to 0.5V
  - Phase II: decode address, enable wordline
    - Capacitor swings bitline voltage up(down)
    - Sense-amplifier interprets swing as 1(0)
  - **Destructive read**: word bits now discharged
- Write: similar to cache write
  - Phase I: decode address, enable wordline
  - Phase II: enable bitlines
    - High bitlines charge corresponding capacitors
- What about **leakage over time**?

DRAM Operation II



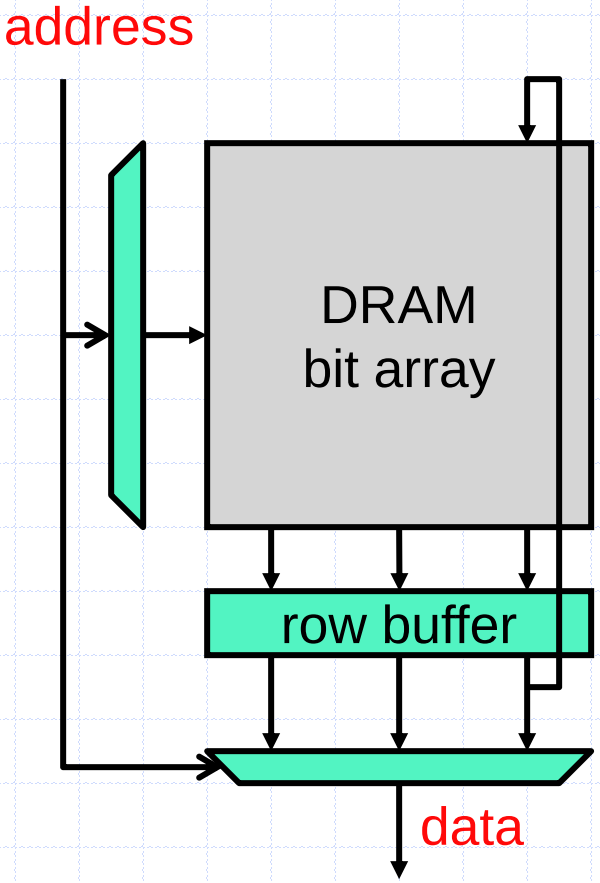
- Solution: add set of D-latches (**row buffer**)
- Read: two steps
  - Step I: read selected word into row buffer
  - Step IIA: read row buffer out to pins
  - Step IIB: write row buffer back to selected word
  - \+ Solves "destructive read" problem
- Write: two steps
  - Step IA: read selected word into row buffer
  - Step IB: write data into row buffer
  - Step II: write row buffer back to selected word
- \+ Also solves 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

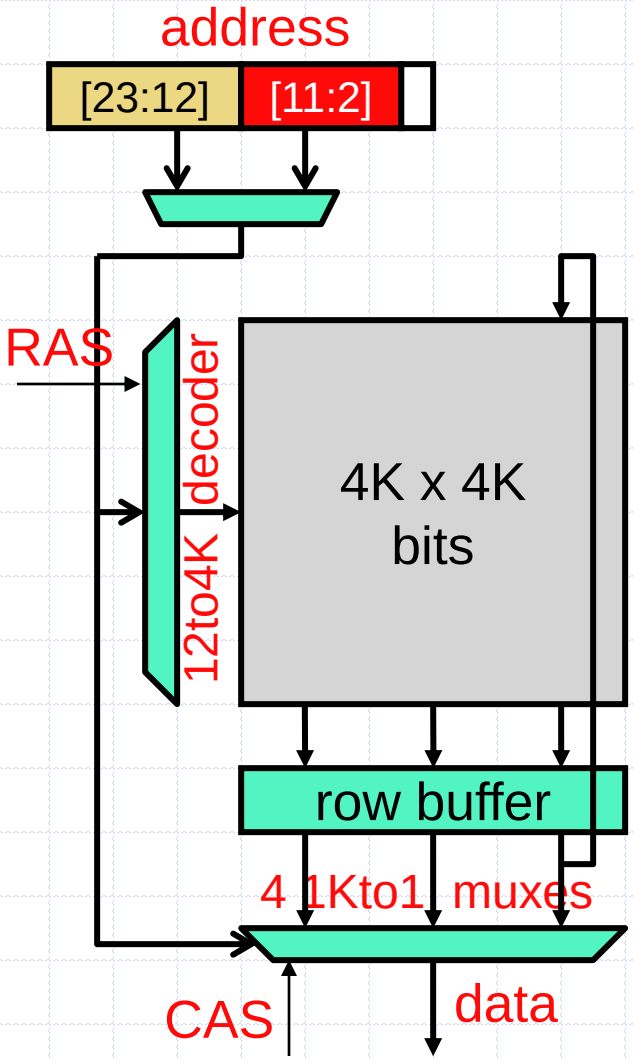


- DRAM parameters
  - Large capacity: e.g., 64–256Mb
    - Arranged as square

    + Minimizes wire length

    + Maximizes refresh efficiency
  - Narrow data interface: 1–16 bit
    - Cheap packages → few bus pins
  - Narrow address interface: N/2 bits
    - 16Mb DRAM has a 12-bit address bus
    - How does that work?

Two-Level Addressing



- **Two-level addressing**
  - Row decoder/column muxes share address lines
  - Two strobes (RAS, CAS) signal which part of address currently on bus
- Asynchronous access
  - Level 1: RAS high
    - Upper address bits on address bus
    - Read row into row buffer
  - Level 2: CAS high
    - Lower address bits on address bus
    - Mux row buffer onto data bus

Access Latency and Cycle Time

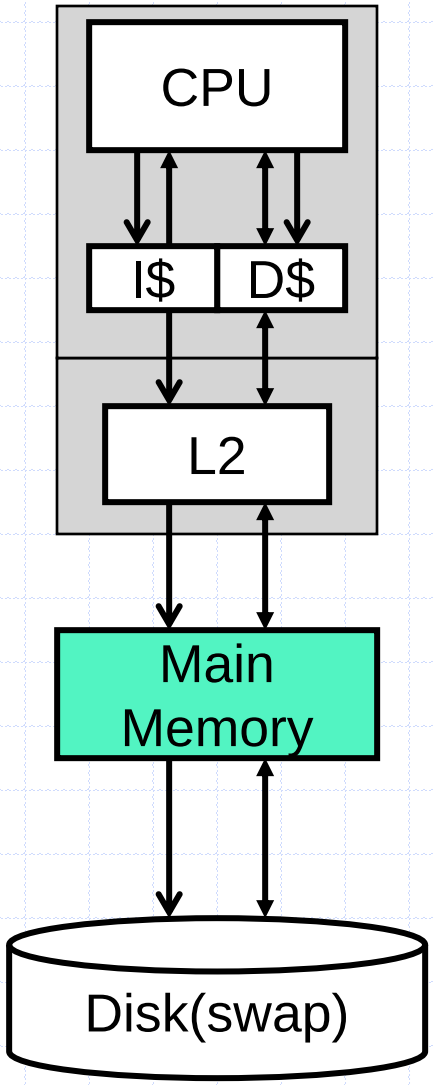
- DRAM access much slower than SRAM
  - More bits → longer wires
  - Buffered access with two-level addressing
  - SRAM access latency: 2–3ns
  - DRAM access latency: 30–50ns

- 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

DRAM Latency and Power Derivations

- Same basic form as SRAM
  - Most of the equations are geometrically derived
  - Same structure for decoders, wordlines, muxes
- Some differences
  - Somewhat different pre-charge/sensing scheme
  - Array access represents smaller part of total access
  - Arrays not multi-ported

Building a Memory System

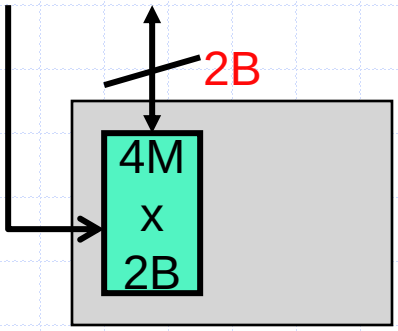


- How to build an efficient main memory out of standard DRAM chips?
  - How many DRAM chips?
  - What width/speed (data) bus to use?
    - Assume separate address bus
- Main memory interface: L2 miss blocks
  - What do you want $t_{miss\text{-}L2}$ to be?

An Example Memory System

- Parameters
  - 32-bit machine
  - L2 with 32B blocks
  - 4Mx16b DRAMs, 20ns access time, 40ns cycle time
  - 100MHz (10ns period) data bus
  - 100MHz, 32-bit address bus
- How many DRAM chips?
- How wide to make the data bus?

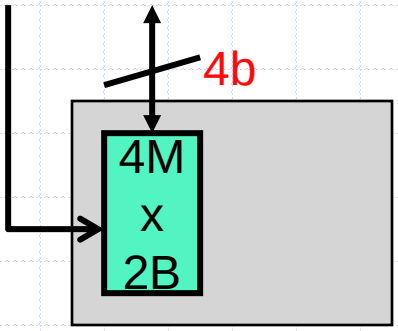
First Memory System Design



- 1 DRAM + 16b bus
  - Access time: 630ns
    - Not including address
  - Cycle time: 640ns
    - DRAM ready to handle another miss

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



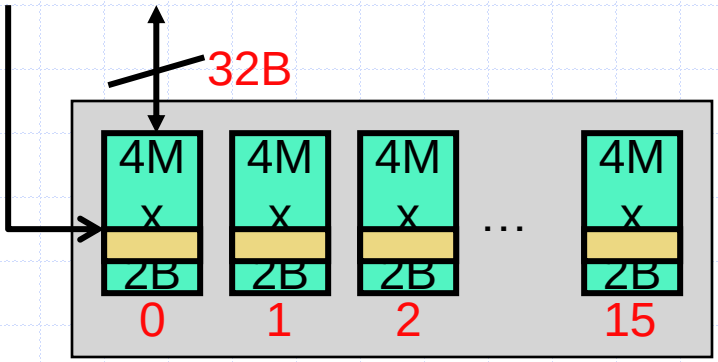
- 1 DRAM + 4b bus
  - One DRAM chip, don't need 16b bus
  - Balanced system → match bandwidths
  - DRAM: 2B / 40ns → 4b / 10ns

  - Access time: 660ns (30ns longer, 4%)
  - Cycle time: 640ns (same)

  + 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



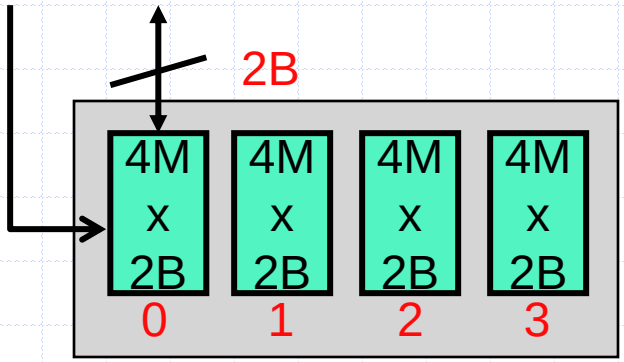
T (ns)	DRAM0	DRAM1	DRAM15	Bus
10	**[31:30]**	**[29:28]**	**[1:0]**	
20	**[31:30]**	**[29:28]**	**[1:0]**	
30	**refresh**	**refresh**	**refresh**	**[31:0]**
40	**refresh**	**refresh**	**refresh**	

- How fast can we go?
- 16 DRAM chips + 32B bus
  - **Stripe data across chips**
  - Byte M in chip (M/2)%16
  - Access time: 30ns
  - Cycle time: 40ns
  - 32B bus is very expensive
  - 128MB of memory isn't, but you may not want that much

Latency and Bandwidth

- In general, given bus parameters…
  - Find smallest number of chips that minimizes cycle time
  - Approach: match bandwidths

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]**

More Bandwidth From One DRAM

- **EDO**: extended data out
  - Multiple row buffer reads/writes
    - Send only column addresses
- **SDRAM**: synchronous DRAM
  - Read/write row buffer chunks on clock edge
    - No need to send column addresses at all
  - **DDR SDRAM**: double-data rate SDRAM
    - Read/write on both clock edges
  - Popular these days
- **RDRAM**: aka RAMBUS
  - Multiple row buffers, "split" transactions, other complex behaviors
  - Very expensive, high end systems only

Memory Access and Clock Frequency

- Nominal **clock frequency** applies to CPU and caches
  - Memory bus has its own clock, typically much slower
  - DRAM has no clock (SDRAM operates on bus clock)

- Careful when doing calculations
  - 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 clock speed won't help at all

Memory/Clock Frequency Example

- Parameters
  - 1GHz CPU, base CPI = 1
  - I$: 1% miss rate, 32B blocks (ignore D$, L2)
  - Data bus: 100MHz, 8B (ignore address bus)
  - DRAM: 10ns access, 20ns cycle, #chips to match bus bandwidth

- What are CPI and MIPS including memory latency?
  - Bus: frequency = 100MHz $\rightarrow$ latency = 10ns (for 8B)
  - Memory system cycle time = bus latency to transfer 32B = 40ns
  - Memory system access time = 50ns (10ns DRAM access + bus)
  - 1GHz clock $\rightarrow$ 50ns = 50 cycles
  - $CPI_{+memory}$ = 1 + (0.01*50) = 1 + 0.5 = 1.5
  - $MIPS_{+memory}$ = 1GHz / 1.5 CPI = 1000MHz / 1.5 CPI = 667

Memory/Clock Frequency Example

- What are CPI and MIPS if clock speed is doubled?
  - Memory parameters same: 50ns access, 40ns cycle
  - 2GHz clock $\rightarrow$ 50ns = 100 cycles
  - $CPI_{+memory}$ = 1 + (0.01*100) = 1 + 1 = 2
  - $MIPS_{+memory}$ = 2GHz / 2 CPI = 2000MHz / 2 CPI = 1000
- What is the peak MIPS if we can only change clock?
  - Available bandwidth: 32B/40ns = 0.8B/ns
  - Needed bandwidth: 0.01*32B/cycle = 0.32B/cycle * **X** cycle/ns
  - Memory is a bottleneck at 0.8/0.32 cycle/ns = 2.5GHz
    - No sustained speedup possible after that point
  - 2.5GHz clock $\rightarrow$ 50ns = 125 cycles
  - $CPI_{+memory}$ = 1 + (0.01*125) = 1 + 1.25 = 2.25
  - $MIPS_{+memory}$ = 2.5GHz / 2.25 CPI = 2500MHz / 2.5 CPI = 1111

Digital Rights Management

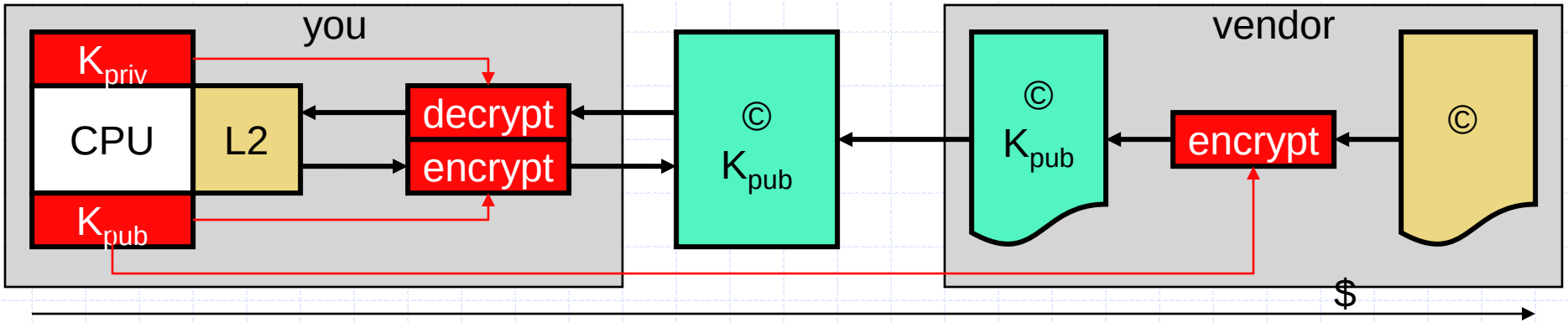
- **Digital rights management**
  - Question: how to enforce digital copyright?
    - Electronically, not legally
  - "Trying to make bits un-copiable is like trying to make water un-wet"

  - Suppose you have some piece of copyrighted material ©…
    - You can easily make a copy of ©
  - But, what if © is encrypted?
    - In order to use ©, you must also have the decryptor
    - Can hack decryptor to spit out unencrypted ©
    - Or hack OS to look at decryptor's physical memory

Aside: Public-Key Cryptography

- **Public-key cryptography**
  - Asymmetric: pair of keys
    - **K_{pub}**: used for encryption, published
    - **K_{priv}**: used for decryption, secret
    - acrypt(acrypt(M, K_{pub}), K_{priv}) = acrypt(acrypt(M, K_{priv}), K_{pub}) = M
    - Well-known example: RSA
- Two uses
  - **Encryption**
    - Someone sends you encrypted message M: C = acrypt(M, K_{pub})
    - You are the only one that can decrypt it
  - **Authentication/Digital Signature**
    - You send someone a chosen plaintext M
    - They "sign" it by sending back DS = acrypt(M, K_{priv})
    - If acrypt(DS, K_{pub}) = M, then they are who K_{pub} says they are

Research: XOM



- **eXecute Only Memory (XOM)**
  - Stanford research project [Lie+, ASPLOS'00]
  - Two registers: K_{priv}, K_{pub} different for every chip (Flash program)
    - Software can get at K_{pub}, but K_{priv} is hardware's secret
  - Hardware encryption/decryption engine on L2 fill/spill path
  - Vendor sells you acrypt(©, K_{pub})
    + Even if someone copies it, they won't have K_{priv} to decrypt it
    - Plaintext © only exists on-chip
    + Even OS can never see plaintext ©

XOM: Not Quite

- Performance consideration
  - Asymmetric en-/de-cryption is slow, **symmetric** (one key) faster
    - E.g., DES, AES (Rijndael)
    - – Problem: can't publish encryption key without also...
- XOM Take II
  - Vendor chooses random symmetric key K_{sym}
  - Sells you scrypt(©, K_{sym}) + acrypt(K_{sym}, K_{pub})
  - Two-stage decryption
    - Decrypt K_{sym} using K_{priv}: slow (but for one piece of data)
    - Decrypt © using K_{sym}: fast
  - Note: SSL does the same thing
    - Uses asymmetric cryptography to choose symmetric session key

Error Detection: 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

Error Correction: Hamming Codes

- **Hamming Code**
  - H(A,B) = number of 1's in A^B (number of bits that differ)
    - Called "Hamming distance"
  - Use D data bits + C check bits construct a set of "codewords"
    - Check bits are parities on different subsets of data bits
  - $\forall$codewords A,B H(A,B) $\geq \alpha$
    - No combination of α–1 transforms one codeword into another
    - For simple parity: $\alpha = 2$
  - Errors of δ bits (or fewer) can be detected if $\alpha = \delta + 1$
  - Errors of β bits or fewer can be corrected if $\alpha = 2\beta + 1$
  - Errors of δ bits can be detected and errors of β bits can be corrected if $\alpha = \beta + \delta + 1$

SEC Hamming Code

- **SEC**: single-error correct
  - $C = \log_2 D + 1$
  - \+ Relative overhead decreases as D grows

- Example: $D = 4 \rightarrow C = 3$
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4$
  - $c_1 = d_1 \wedge d_2 \wedge d_4$, $c_2 = d_1 \wedge d_3 \wedge d_4$, $c_3 = d_2 \wedge d_3 \wedge d_4$
  - Syndrome: $c_i \wedge c'_i = 0$? no error : points to flipped-bit
- Working example
  - Original data = 0110 $\rightarrow c_1 = 1, c_2 = 1, c_3 = 0$
  - Flip d_2 = 0010 $\rightarrow c'_1 = 0, c'_2 = 1, c'_3 = 1$
    - Syndrome = 101 (binary 5) $\rightarrow$ 5th bit? D_2
  - Flip $c_2 \rightarrow c'_1 = 1, c'_2 = 0, c'_3 = 0$
    - Syndrome = 010 (binary 2) $\rightarrow$ 2nd bit? c_2

SECDED Hamming Code

- **SECDED**: single error correct, double error detect
  - $C = \log_2 D + 2$
  - Additional parity bit to detect additional error
- Example: $D = 4 \rightarrow C = 4$
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4\ \mathbf{c_4}$
  - $c_4 = c_1 \wedge c_2 \wedge d_1 \wedge c_3 \wedge d_2 \wedge d_3 \wedge d_4$
  - Syndrome == 0 and $c'_4 == c_4 \rightarrow$ no error
  - Syndrome != 0 and $c'_4 != c_4 \rightarrow$ 1-bit error
  - Syndrome != 0 and $c'_4 == c_4 \rightarrow$ 2-bit error
  - Syndrome == 0 and $c'_4 != c_4 \rightarrow c_4$ error
- Many machines today use 64-bit SECDED code
  - C = 8 (one additional byte, 12% overhead)
  - ChipKill - correct any aligned 4-bit error
    - If an entire DRAM chips dies, the system still works!