U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

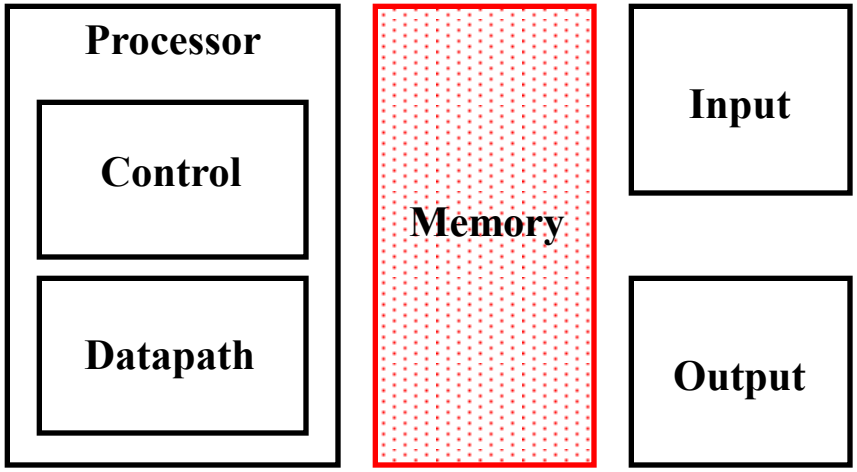
Prof. Karu Sankaralingam

Memory (Chapter 7)

www.cs.wisc.edu/~karu/courses/cs552

Slides combined and enhanced by Mark D. Hill from work by Falsafi, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Outline



- Memory
  - Technology, organization, motivation for hierarchical organization

Memory

- Storage elements
  - registers, latches.
    - Small
    - In processor
    - Expensive to add (??)

Processor Datapath

  - SRAM (Caches)
    - Medium
    - Onchip or board, close to processor
    - Costly
  - DRAM (Main memory)
    - Large
    - 50ns access time
    - Cheap $0.12-0.15/MB (512MB for 60-75$ *)

Memory subsystem

  - Disk/Tape etc.
    - Large, far from processor
    - Slow (~ms)
    - Cheap $0.37-0.40/GB (160GB for 60-65$ *)

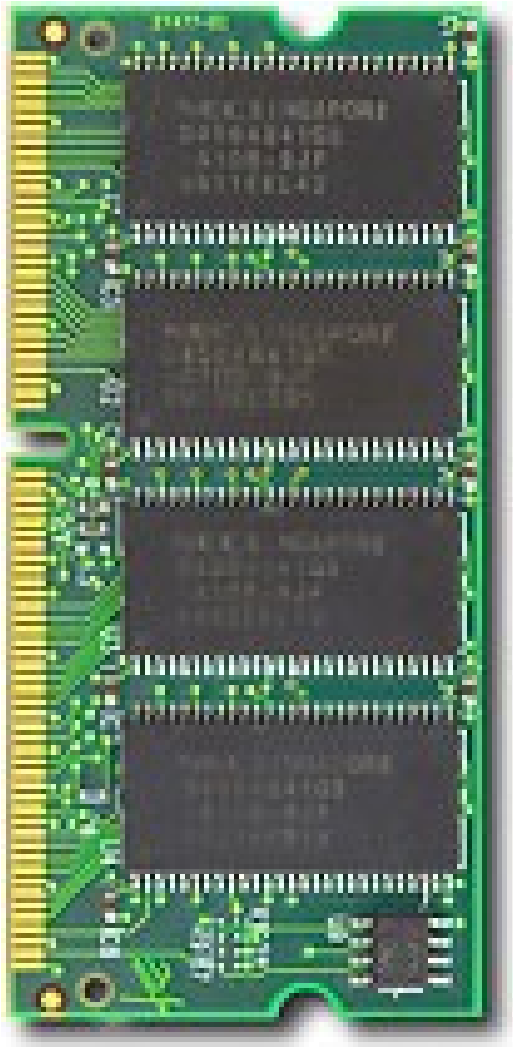
I/O subsystem

Memory Hierarchy Technology

- Random Access:
  - "Random" is good: access time is the same for all locations
  - DRAM: Dynamic Random Access Memory
    - High density, low power, cheap, slow
    - Dynamic: need to be "refreshed" regularly
  - SRAM: Static Random Access Memory
    - Low density, high power, expensive, fast
    - Static: content will last "forever"(until lose power)
- "Not-so-random" Access Technology:
  - Access time varies from location to location and from time to time
  - Examples: Disk, CDROM
- Sequential Access Technology: access time linear in location (e.g.,Tape)
- The Main Memory: DRAMs + Caches: SRAMs

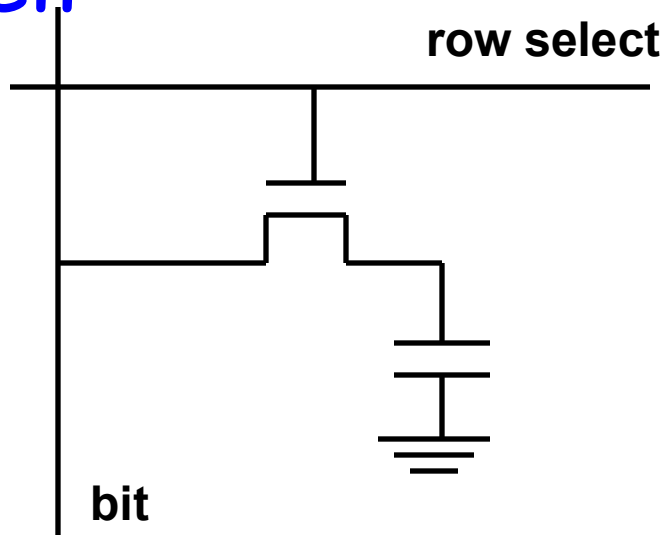
DRAM

- Dynamic RAM
  - Dense, 1T/bit-cell
  - Forgets after a while
  - 16Mb : 4K x 4K cell-array
  - 24 bit address
    - 12 bit for row, 12 for column—reflected in the interface
- Implementation
  - Word/byte DRAM built as DIMM/SIMMs

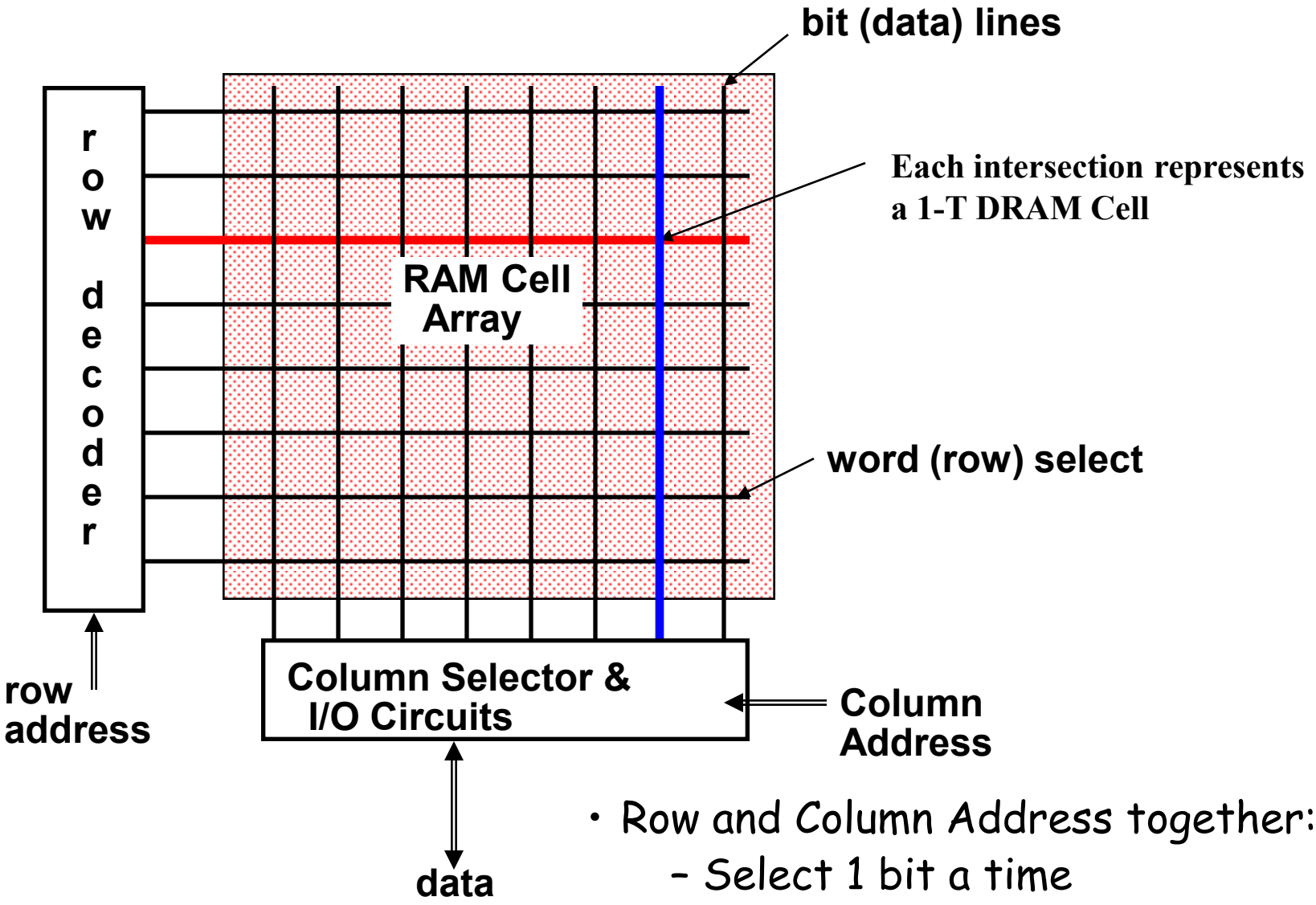


1T DRAM cell

- Charge on capacitor
- Write:
  - 1. Drive bit line
  - 2.. Select row
- Read:
  - 1. Precharge bit line to Vdd
  - 2.. Select row
  - 3. Cell and bit line share charges
    - Very small voltage changes on the bit line
  - 4. Sense (fancy sense amp)
    - Can detect changes of ~1 million electrons*
  - 5. Write: restore the value
- Refresh
  - 1. Just do a dummy read to every cell.



Classical DRAM Organization

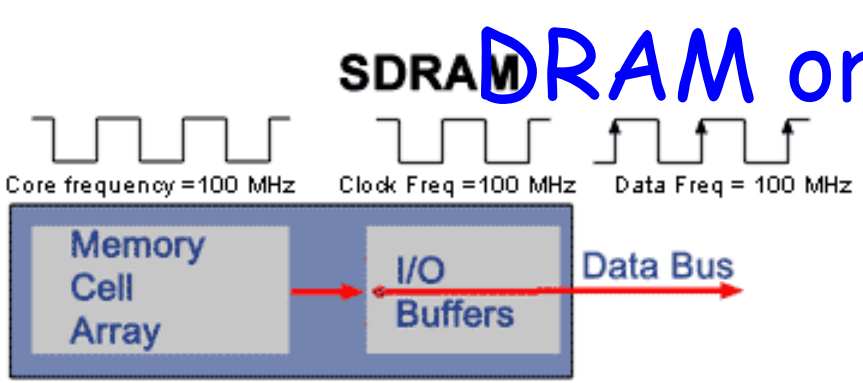


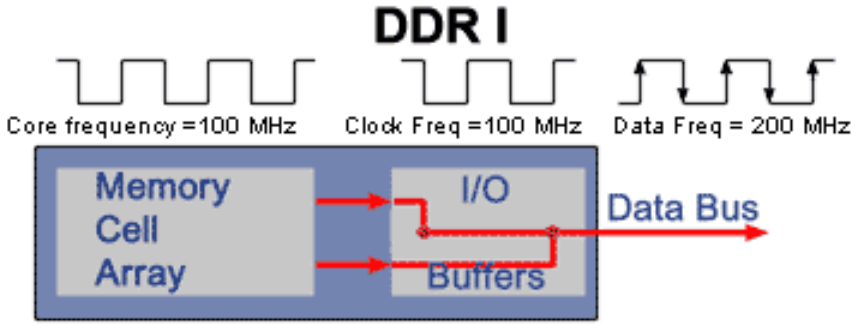
- Row and Column Address together:
  - Select 1 bit a time

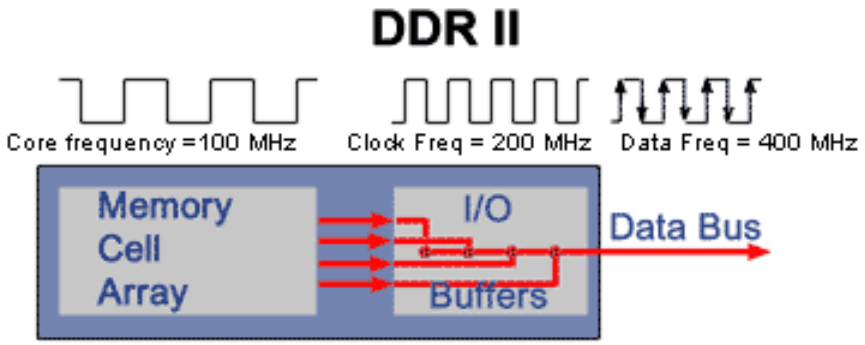
DRAM Optimizations

- Fast Page Mode:
  - Row once, vary column address
- EDO DRAM: Extended data out
  - FPM plus pipelining
- Synchronous DRAM
  - Tied to system clock, increasing bus-speed
  - SDRAM-DDR, DDR-2?
- Fully Buffered DRAM (FB-DIMM)

DRAM organizations





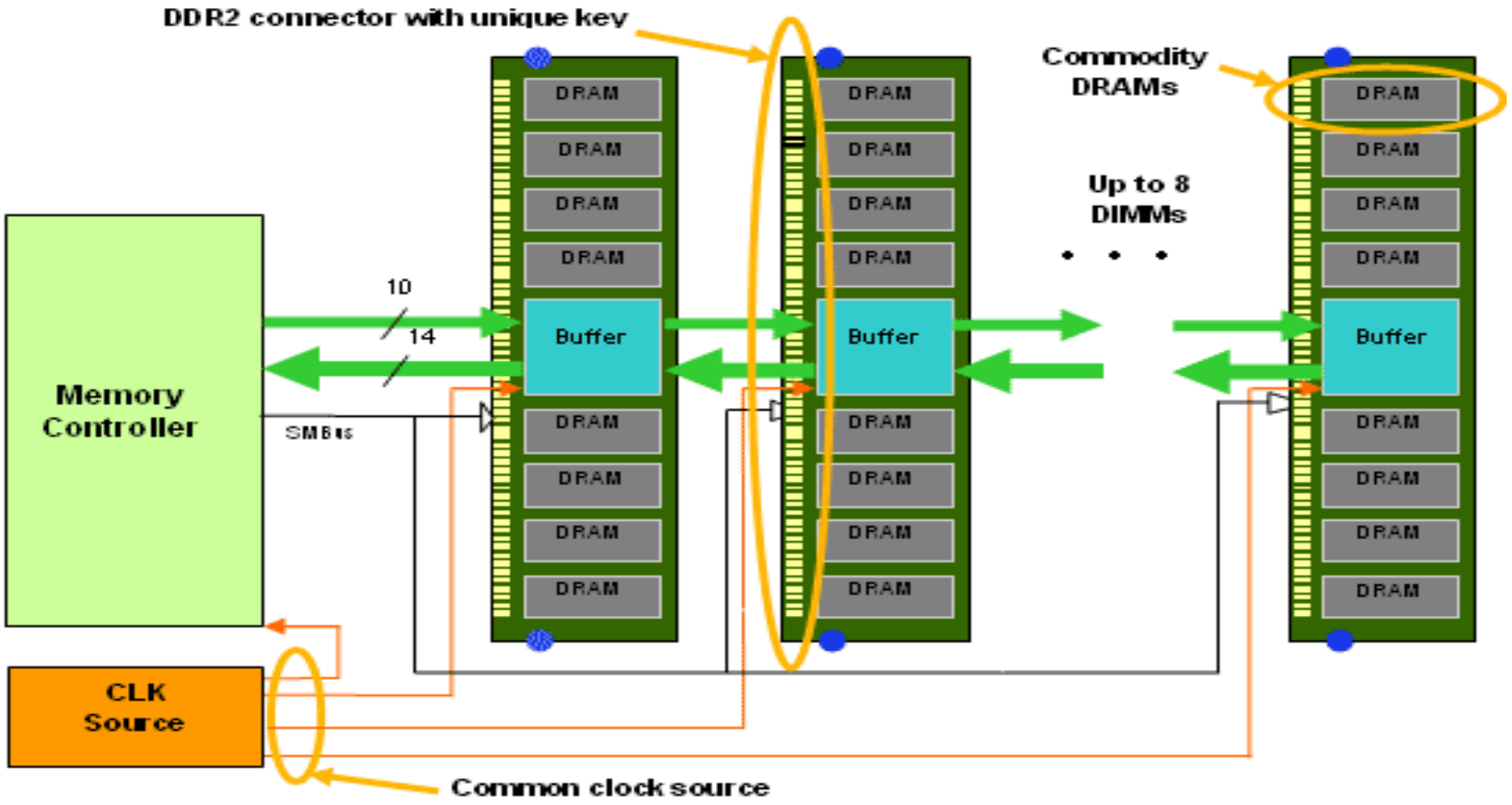


- DRAM core unchanged
- Organization/data transfer optimizations
- Compare SDRAM vs. DDR vs DDR2

- *Picture source:*

  http://www.lostcircuits.com/ via xbitlabs.com

FB-DIMM



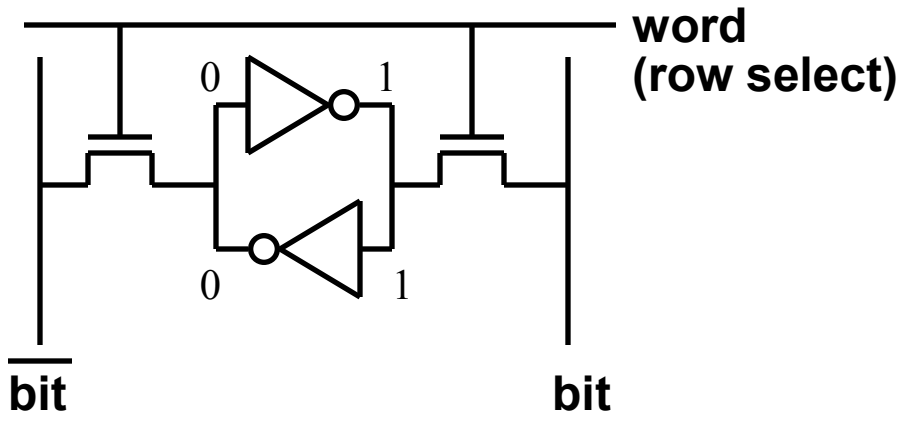
Source: http://www.intel.com/technology/magazine/computing/Fully-buffered-DIMM-0305.htm

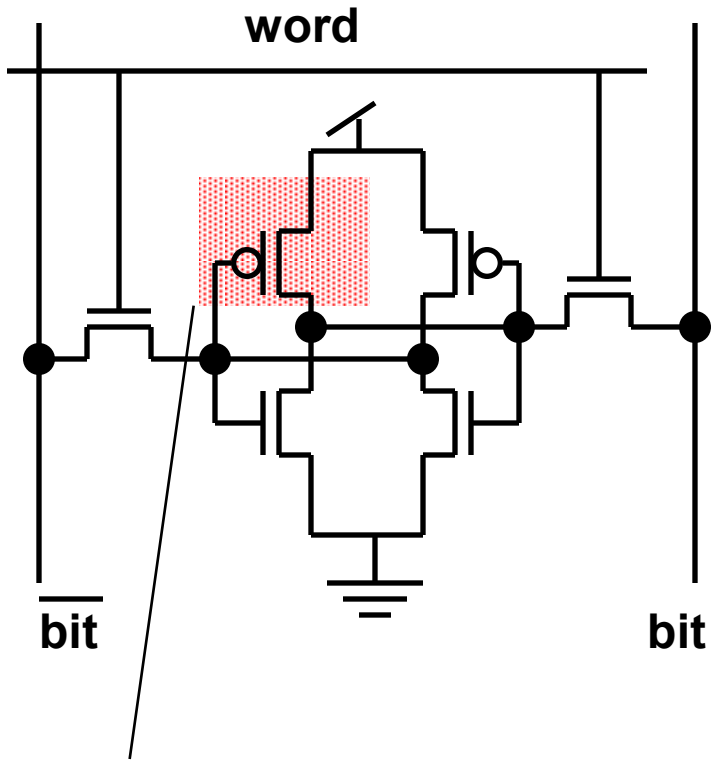
SRAM

- Data is static (as long as power is applied)
- Logically, two cross-connected inverters with switches
  - CMOS inverter, MOS switch
  - 6-transistor implementation

6T SRAM Cell

6-Transistor SRAM Cell

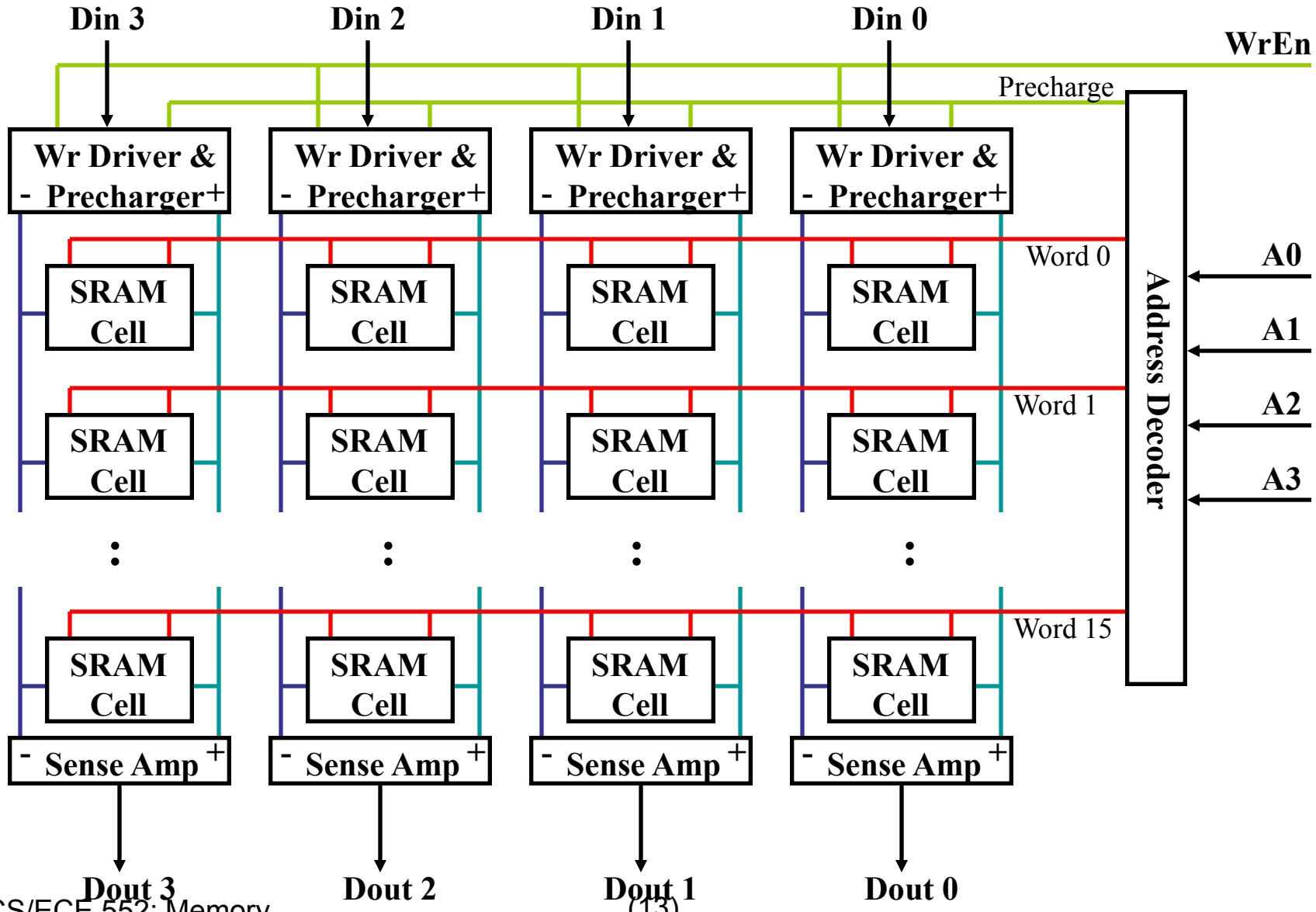




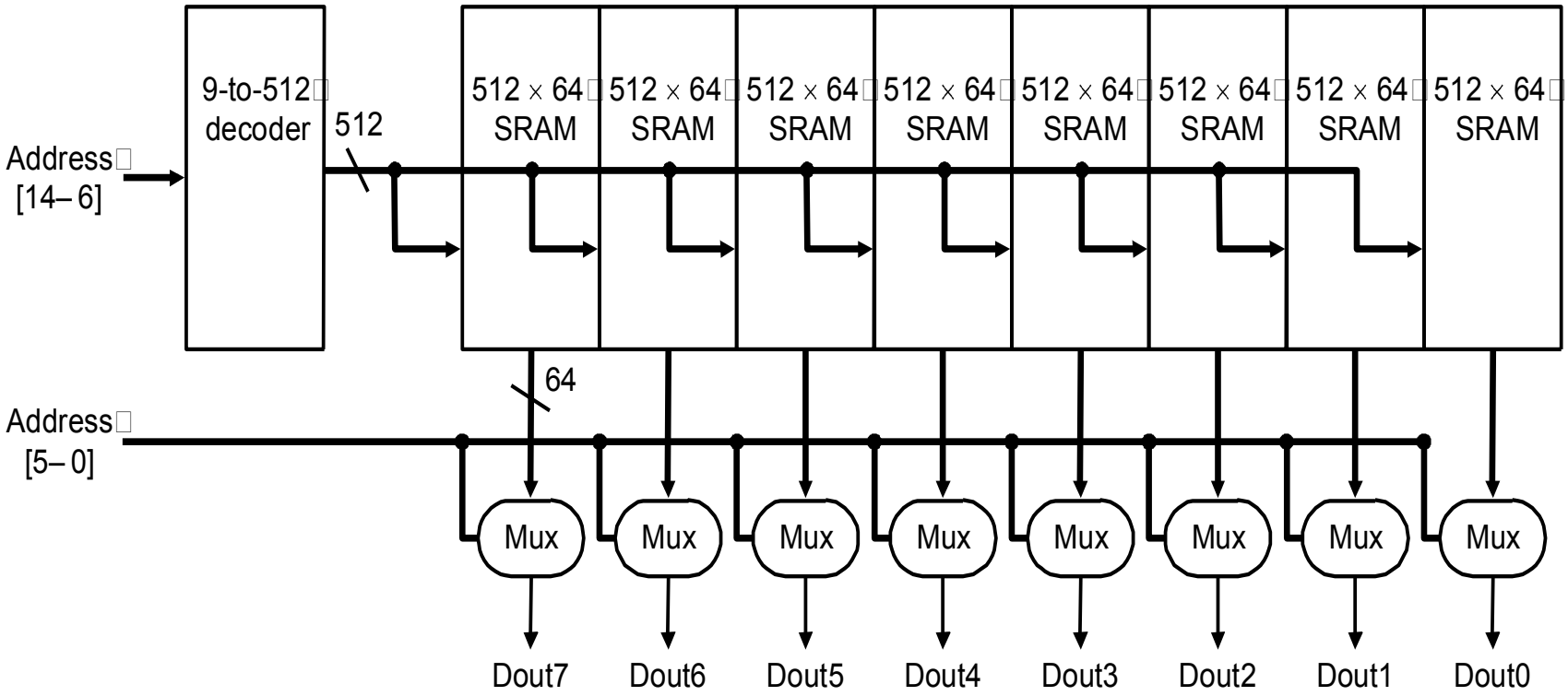
5T version: replaced with pullup to save area

- Write:
  1. Drive bit lines (bit=1, $\overline{bit}$=0)
  2.. Select row
- Read:
  1. Precharge bit and $\overline{bit}$ to Vdd
  2.. Select row
  3. Cell pulls one line low
  4. Sense amp on column detects difference between bit and $\overline{bit}$

SRAM Organization (16x4)



SRAM Organization



- Internal arrays may be different
  - 32Kx8 array realized with 8 512x64 arrays

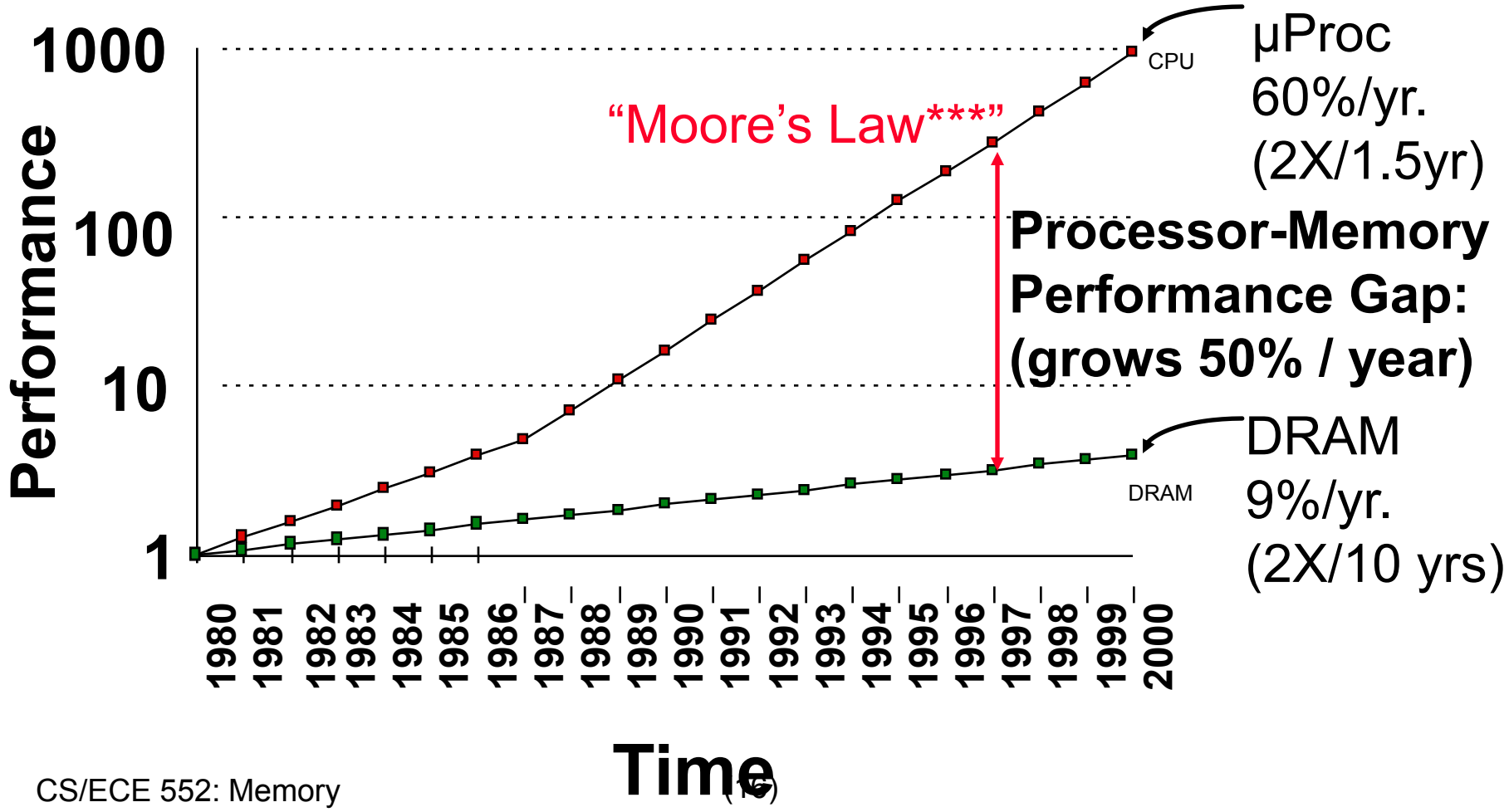
Technology Trends

DRAM		
Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns
1998	256 Mb	
2001	1 Gb	60ns
2004	4 Gb	50ns**

	Capacity	Speed (latency)
Logic:	2x in 3 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

Consequences

- Large and growing Processor-Memory Gap

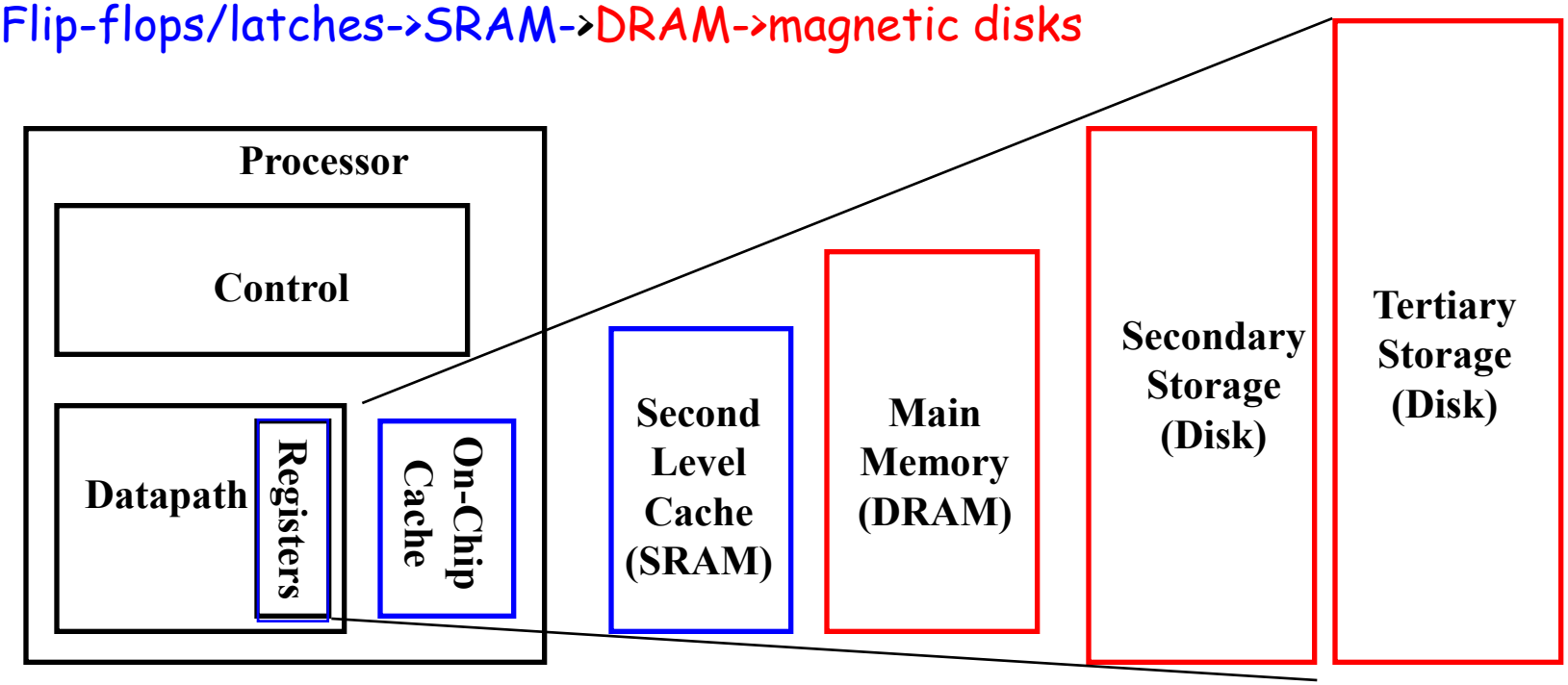


Challenge: Proc-Mem Gap

- Fact: Large memories are slow (and cheap), fast memories are small (and expensive)
- How do we create a memory that is large, cheap and fast (most of the time)?
  - Hierarchy
  - Parallelism

The Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.
- Flip-flops/latches->SRAM->DRAM->magnetic disks

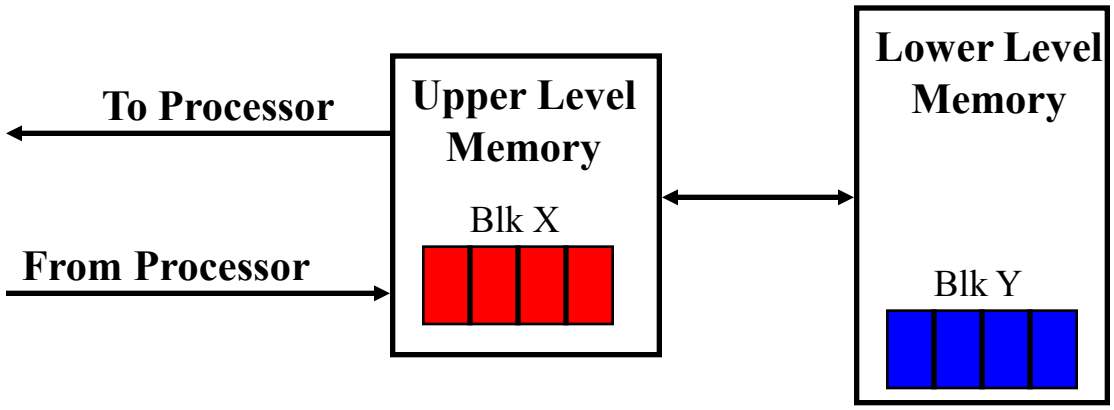


Speed (ns):	1s	5-10s	50-100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks(~Ms)	Ms(~Gs)	Gs	Ts

Locality

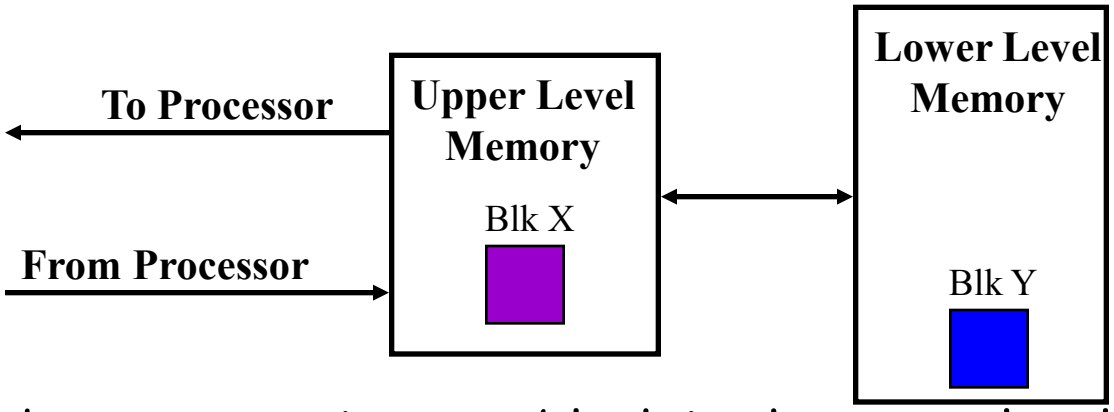
- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
- A library
  - Finding the few books you want: Slow
  - One you found the books
    - Reading various chapters: Fast
    - Switching between books: Fast
  - Library -> Memory: Larger the better
  - Books at table -> Cache: Size is limited but access is faster

Two flavors of locality



- Temporal Locality (Locality in Time):
  - ⇒ Keep most recently accessed data items closer to the processor
  - ⇒ Odds are you'll refer to books on your table more than once
- Spatial Locality (Locality in Space):
  - ⇒ Move blocks consists of contiguous words to the upper levels
  - ⇒ Odds are you'll read read contiguous pages/chapters

Illusion of Speed and Capacity



- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty

Why Memory Hierarchies Work

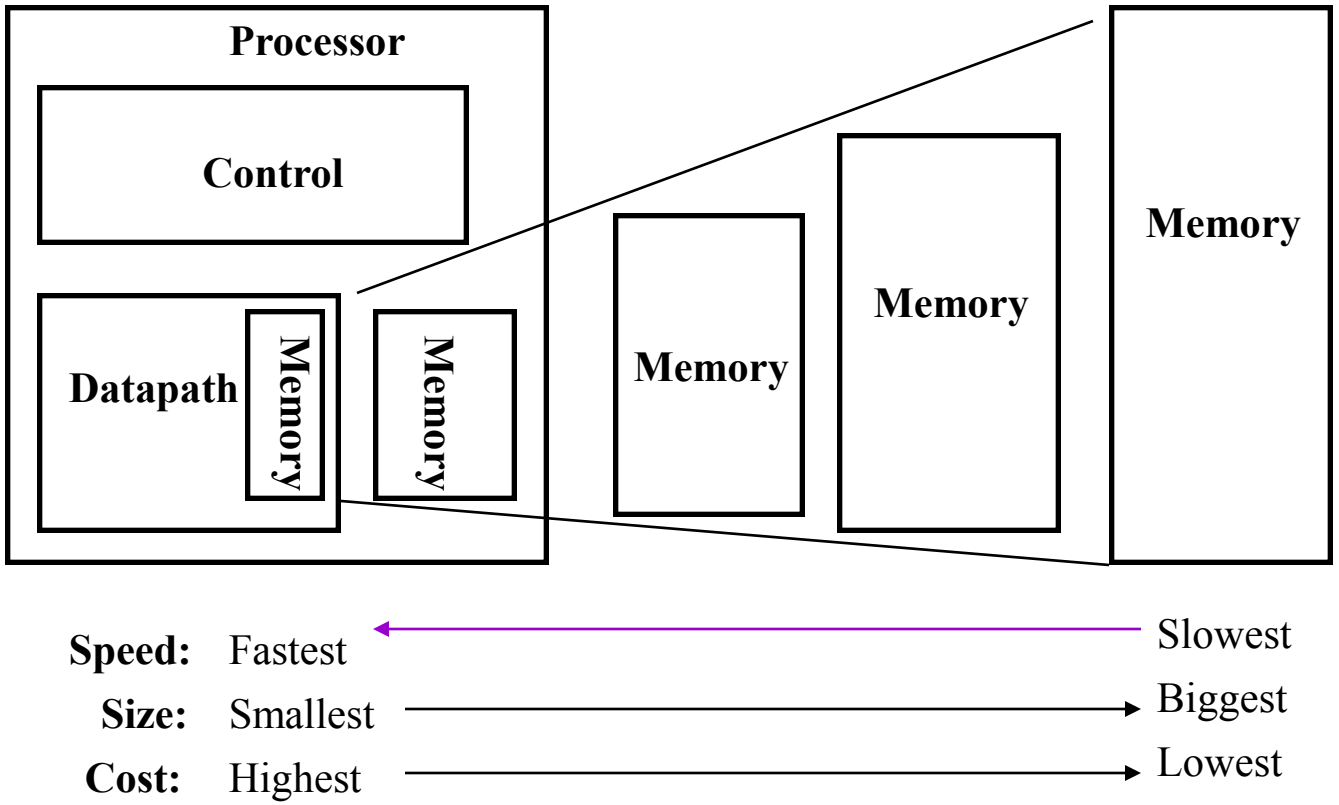
- Amdahl's Law: Make the common case fast
- Locality (usually) makes cache hit common

- Average memory access time (AMAT)
  = access-time + miss-rate * miss-penalty
  - 1 ns + 0.02 * 10ns = 1.2ns << 10ns

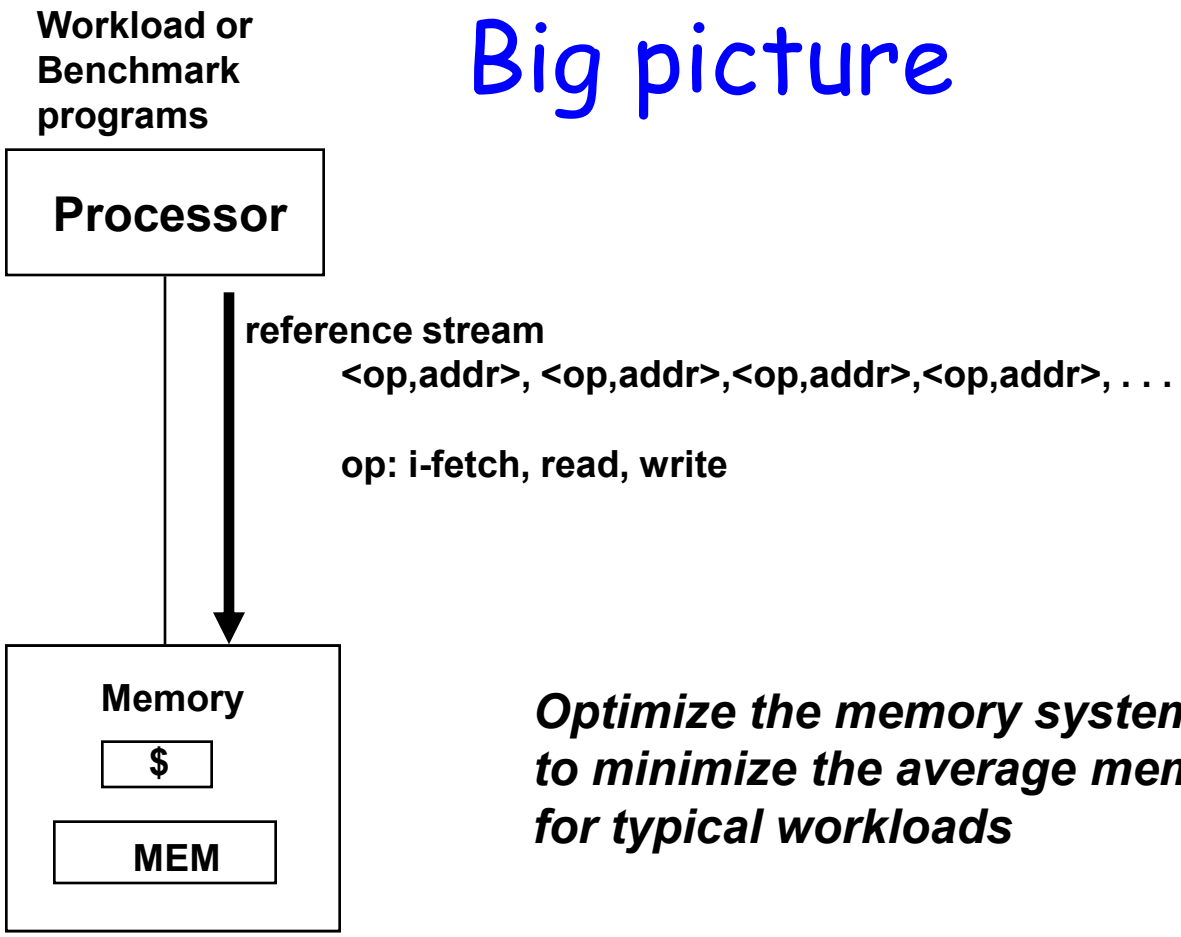
Summary

- Why do we care about the memory system?
  - CPU only as fast as mem-system can supply
- Understand SRAM/DRAM technology
- Exploit locality to (partially) overcome processor-memory gap

The Solution: Hierarchy



Big picture



Optimize the memory system organization to minimize the average memory access time for typical workloads

- Why do we care about AMAT? AMAT affects CPI
  - Remember there is 1.x memory ops per instruction

Impact on Pipelined Performance

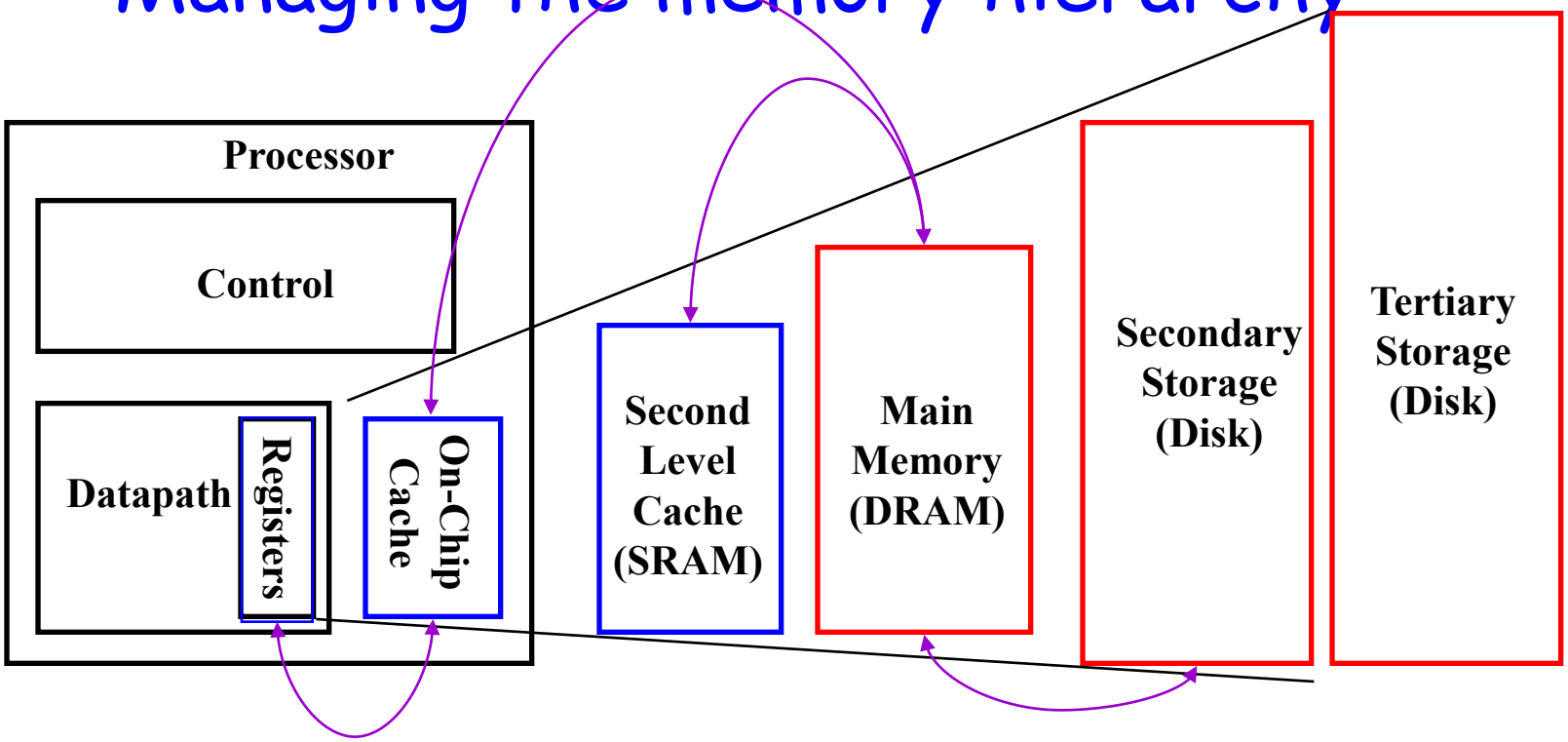
- Suppose a processor executes at
  - Clock Rate = 2 GHz (0.5 ns per cycle)
  - CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

- Suppose that 5% of memory operations get 100 cycle (50ns) miss penalty

- CPI = ideal CPI + average stalls per instruction
  = 1.1(cyc) +(0.30 (datamops/ins)
  x 0.05 (miss/datamop) x 100 (cycle/miss))
  = 1.1 cycle + 1.5 cycle
  = 2. 6
- ~58 % of the time the processor is stalled waiting for memory!

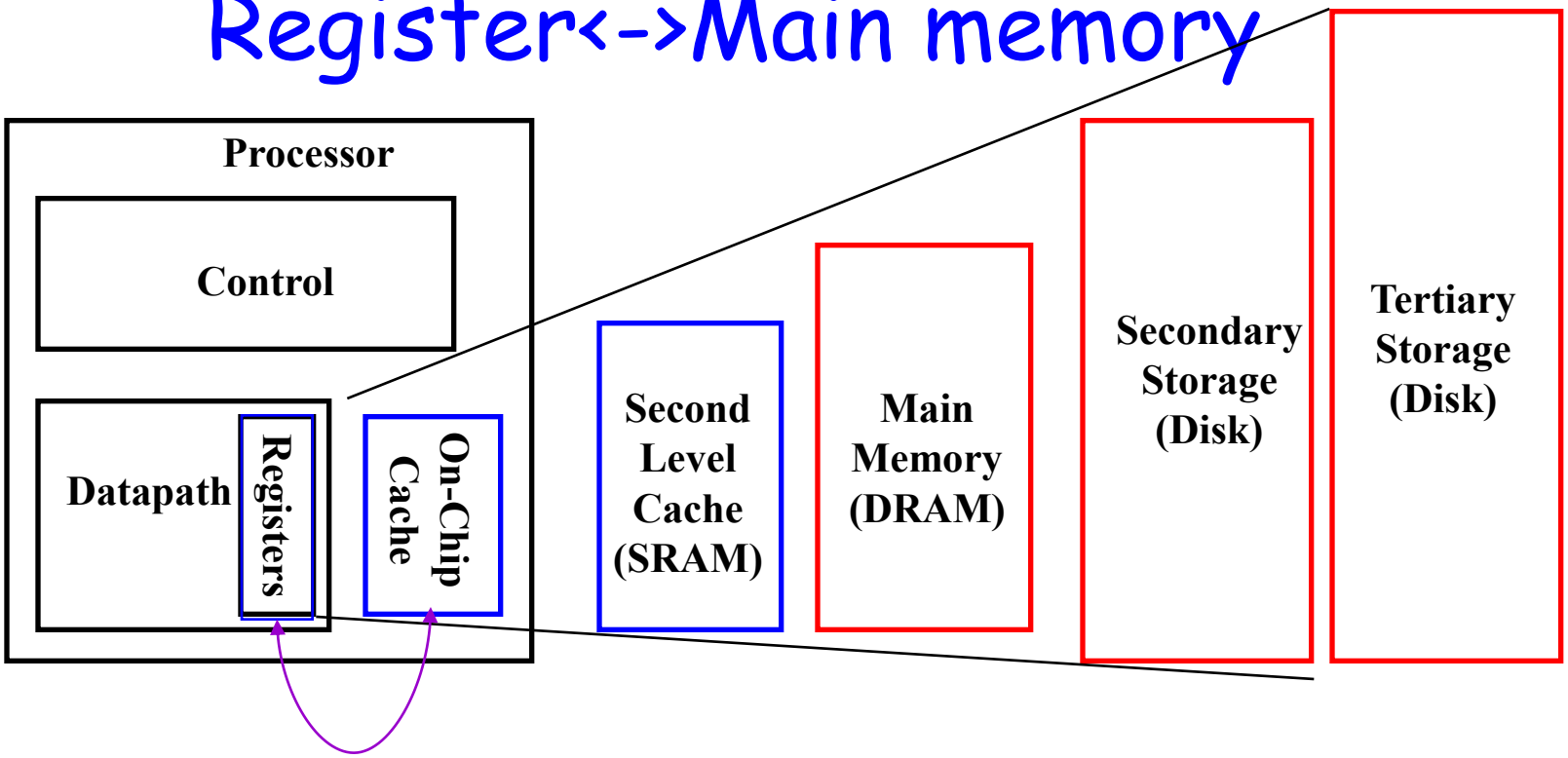
- A 0.5% instruction miss rate would add an additional 0.5 cycles to the CPI

Managing the memory hierarchy



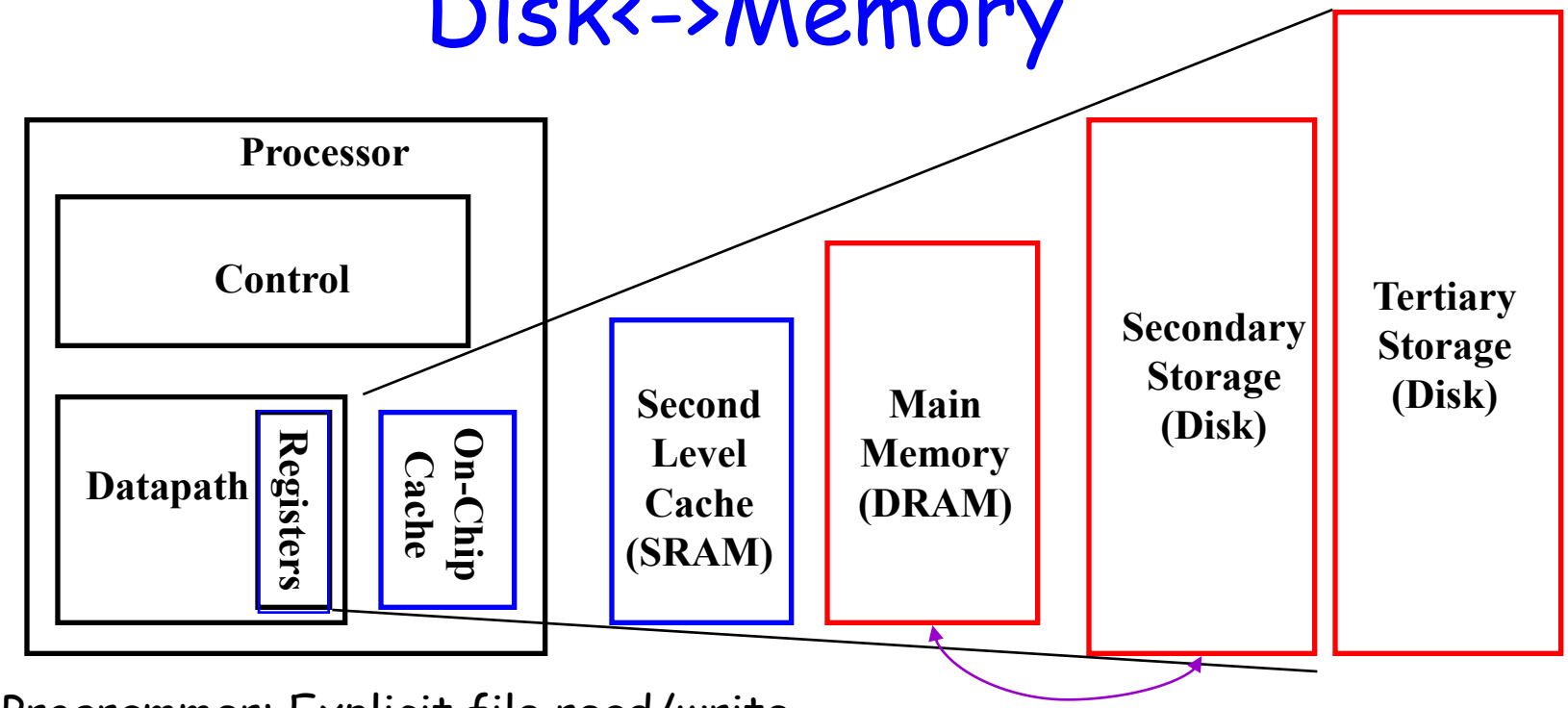
- Whose responsibility is it?
  - Short answer: it depends on the level

Register<->Main memory



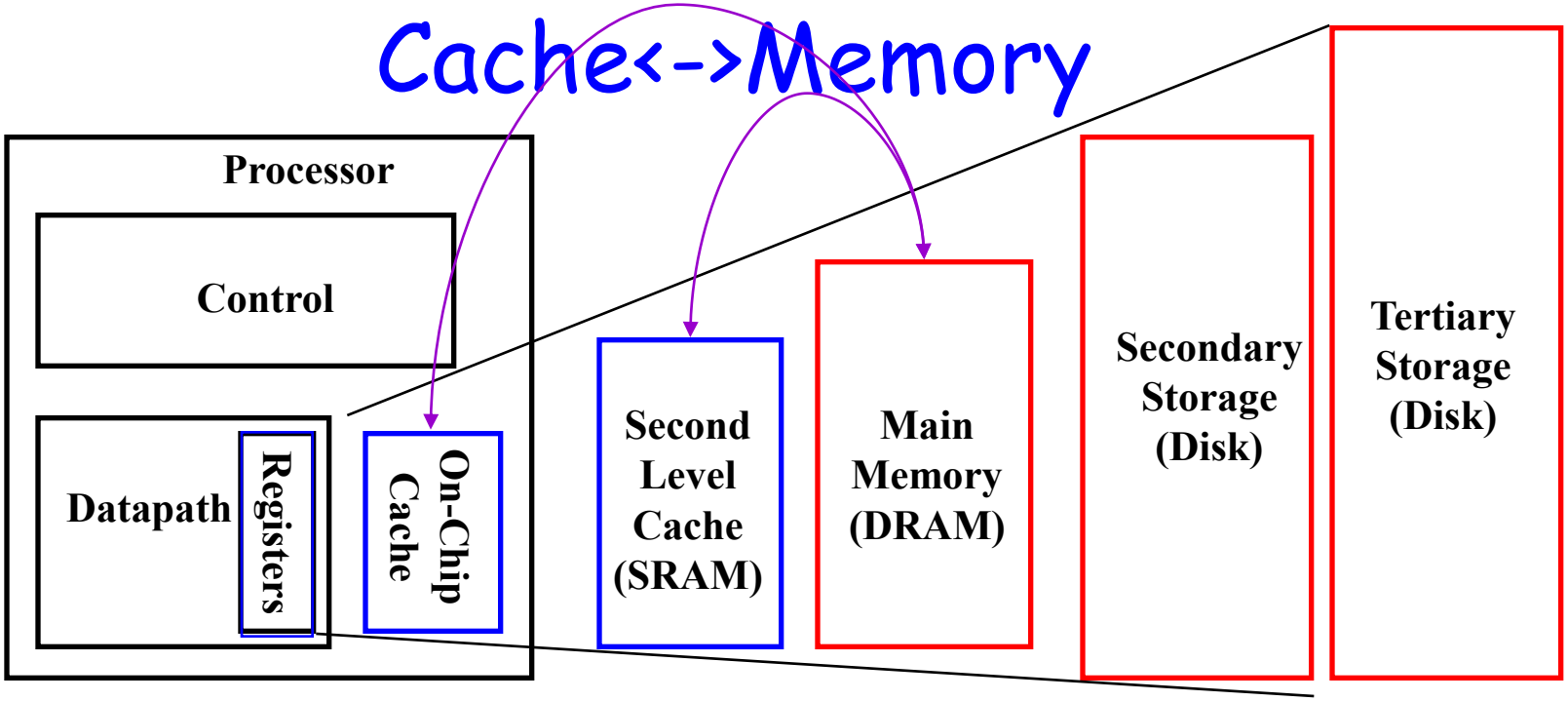
- Managed explicitly by compiler/programmer
  - "Word" granularity
  - Load/store ties memory locations to registers (allocation)
  - Register temporaries ("spill" to memory when needed)
- Complexity!

Disk<->Memory



- Programmer: Explicit file read/write
- Disk-block/Page granularity
- OS: Automatic transparent to user
  - Virtual memory
  - Illusion of large memory, protection
  - More later

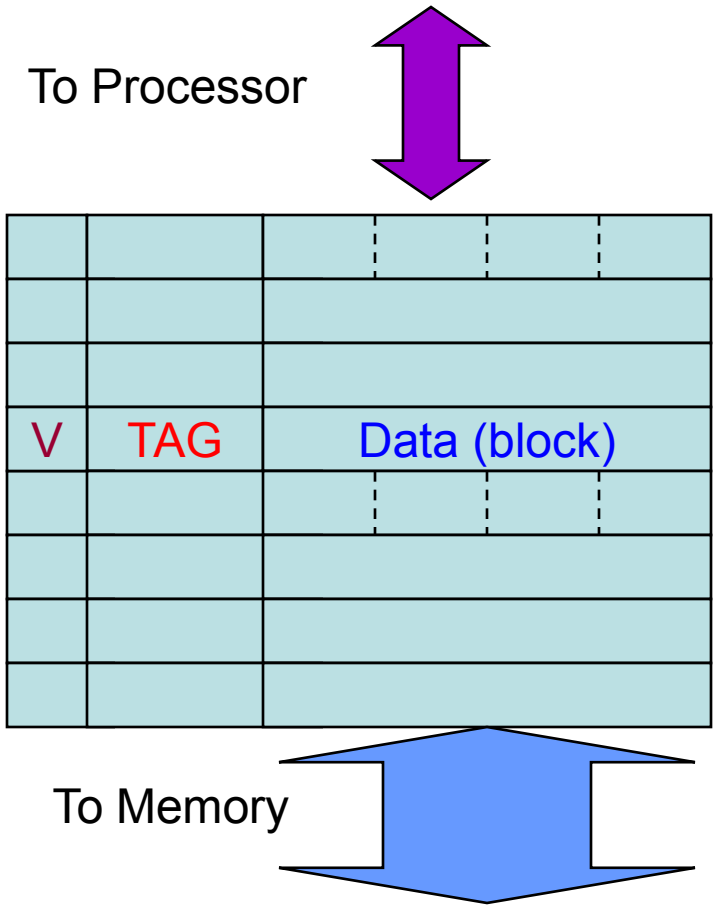
Cache<->Memory



- Hardware managed: needs to be fast
- Automatic: to avoid complexity of explicit management
- "Block" granularity to exploit spatial locality
- Retain recently accessed blocks to exploit temporal locality

Cache Operation

- Tag, data, valid
- Tag:
  - Mapping larger space (all addresses) to a smaller space (cache)
  - To identify which block (address) is resident
- Data:
  - Block: more than one word
- Valid:
  - Not everything in cache is meaningful
- Frame (block-frame/cache-frame)

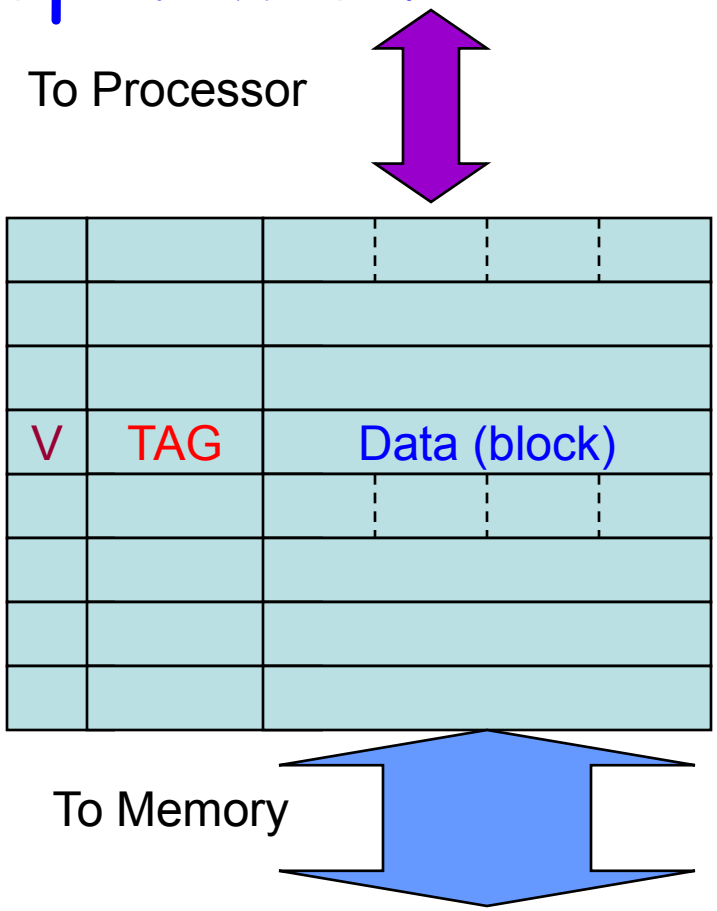


Cache Operation

- Hit/Miss detection
  - If (incoming tag == stored tag)
    - Hit //i.e. block is resident in cache
    - Return word to processor
  - Else
    - Miss
      - Make space : replace some other block
      - Get block from memory
      - Put block in "data" part, set tag using new address tag

Example of cache operation

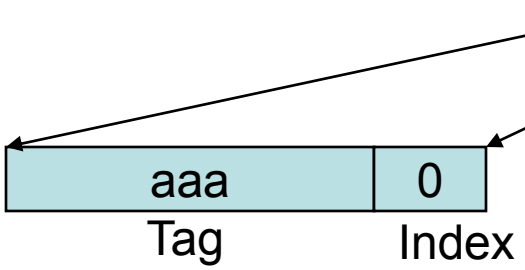
- 16-frame cache
- 16 bit address-space
- Use***:
  - Lower four bits of address as index of frame
  - All other bits of address as tag

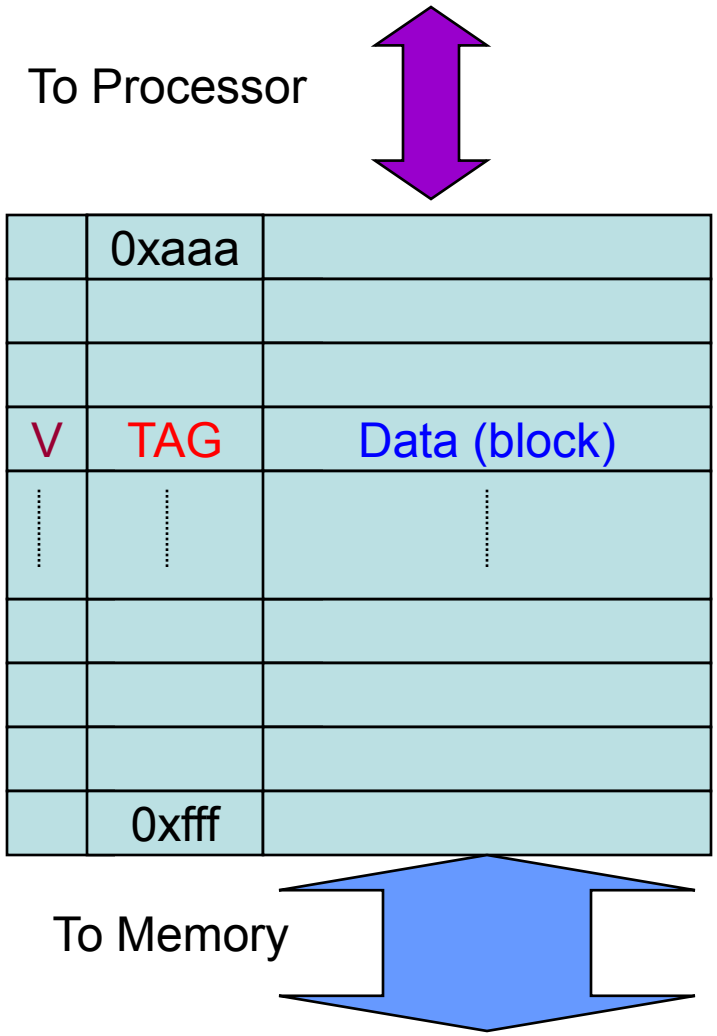


Cache Operation

- Cache operation for the following address-stream

```
0xaa10
0xaaa0
0xffff
0xaaa0
0xffff
0xaaa0
```



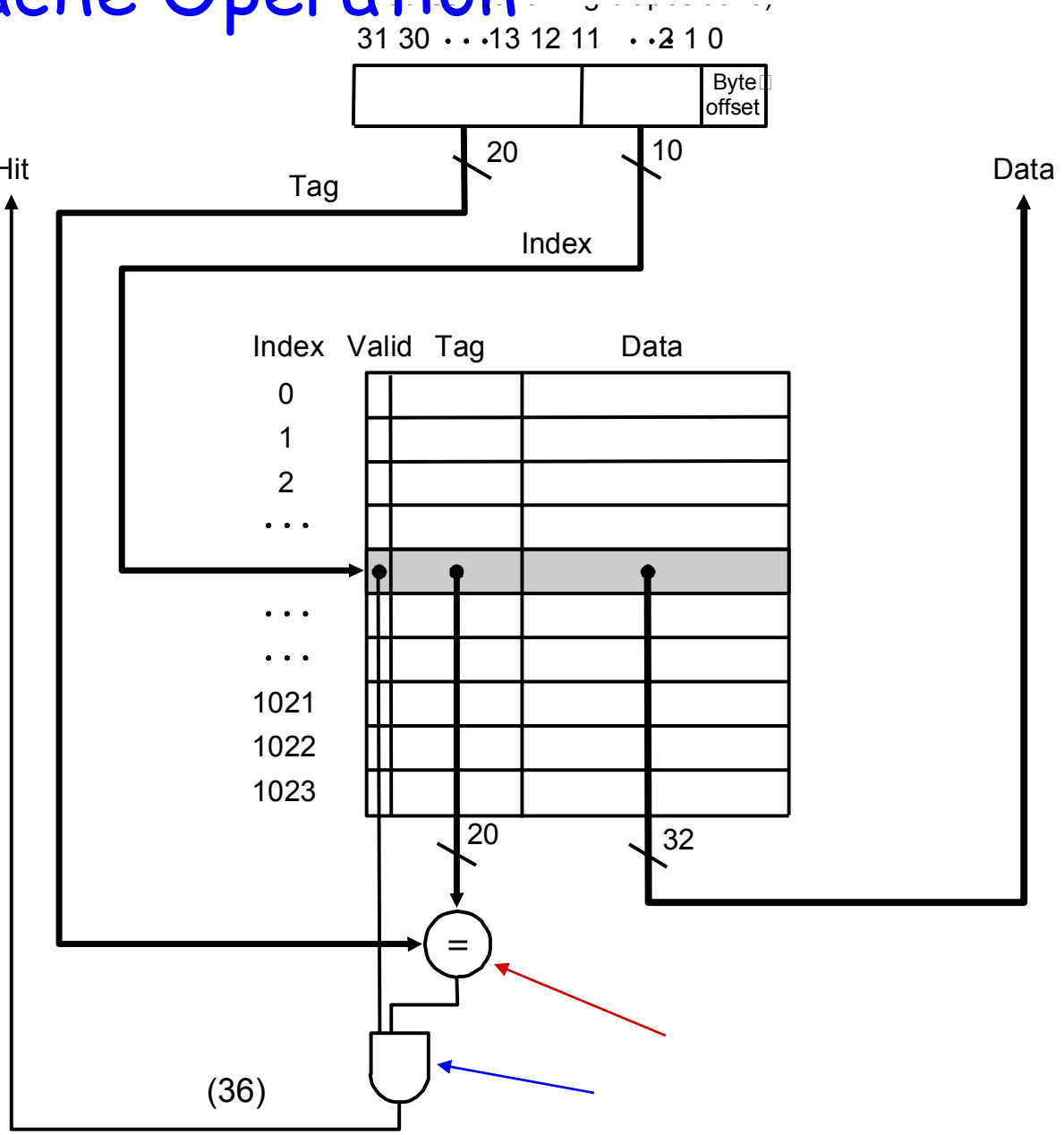


Lookahead

- Summary:
  - Cache management in hardware
  - Caches terminology and organization
    - Frames
    - Blocks
    - Tags
  - Example of Cache operation
- Next lecture : 4 questions
  - Where is a block placed?
  - How is a block found?
  - Which block is replaced?
  - What happens on a write?

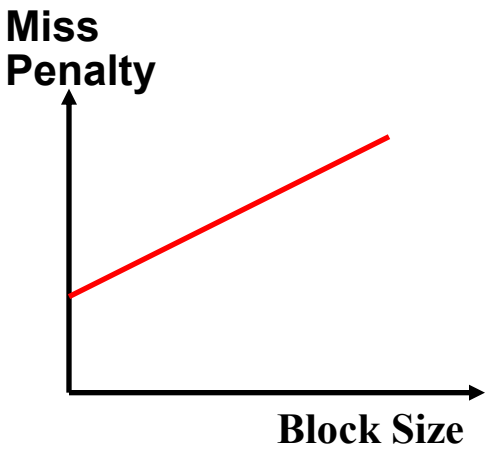
Cache Operation

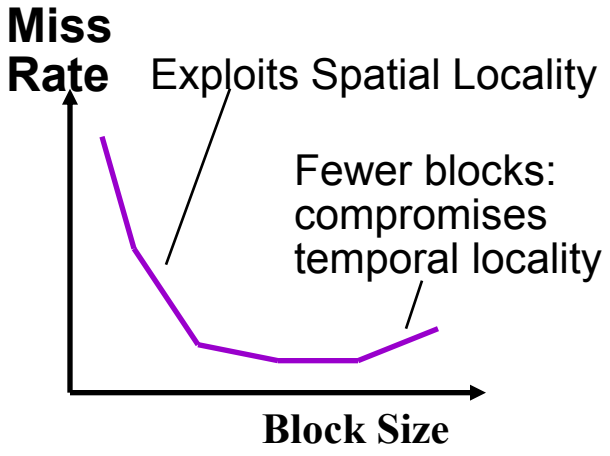
- Tag comparators
- Hit detection
- How many frames?
- What is the cache size?
- What if block size is more than one word?

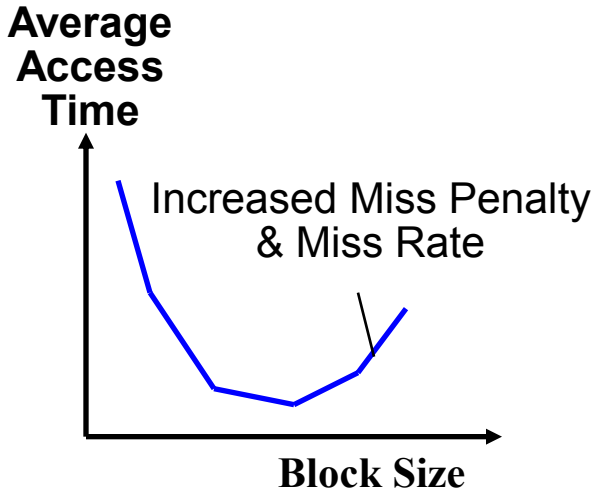


Block Size Tradeoff

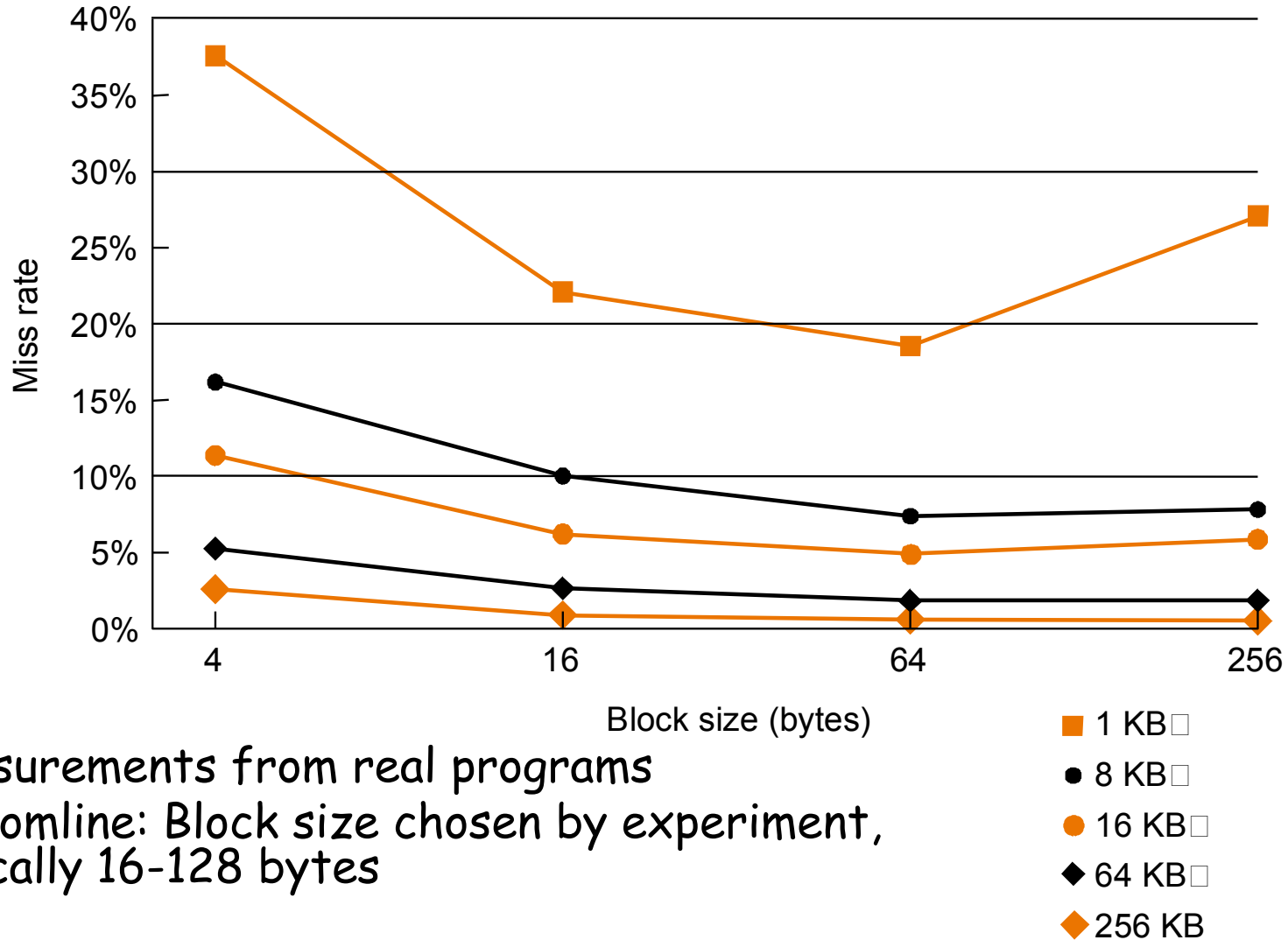
- In general, larger block size take advantage of spatial locality BUT:
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
    - Too few cache blocks
- In general, Average Memory Access Time:
  - = Acces Time + Miss Penalty x Miss Rate





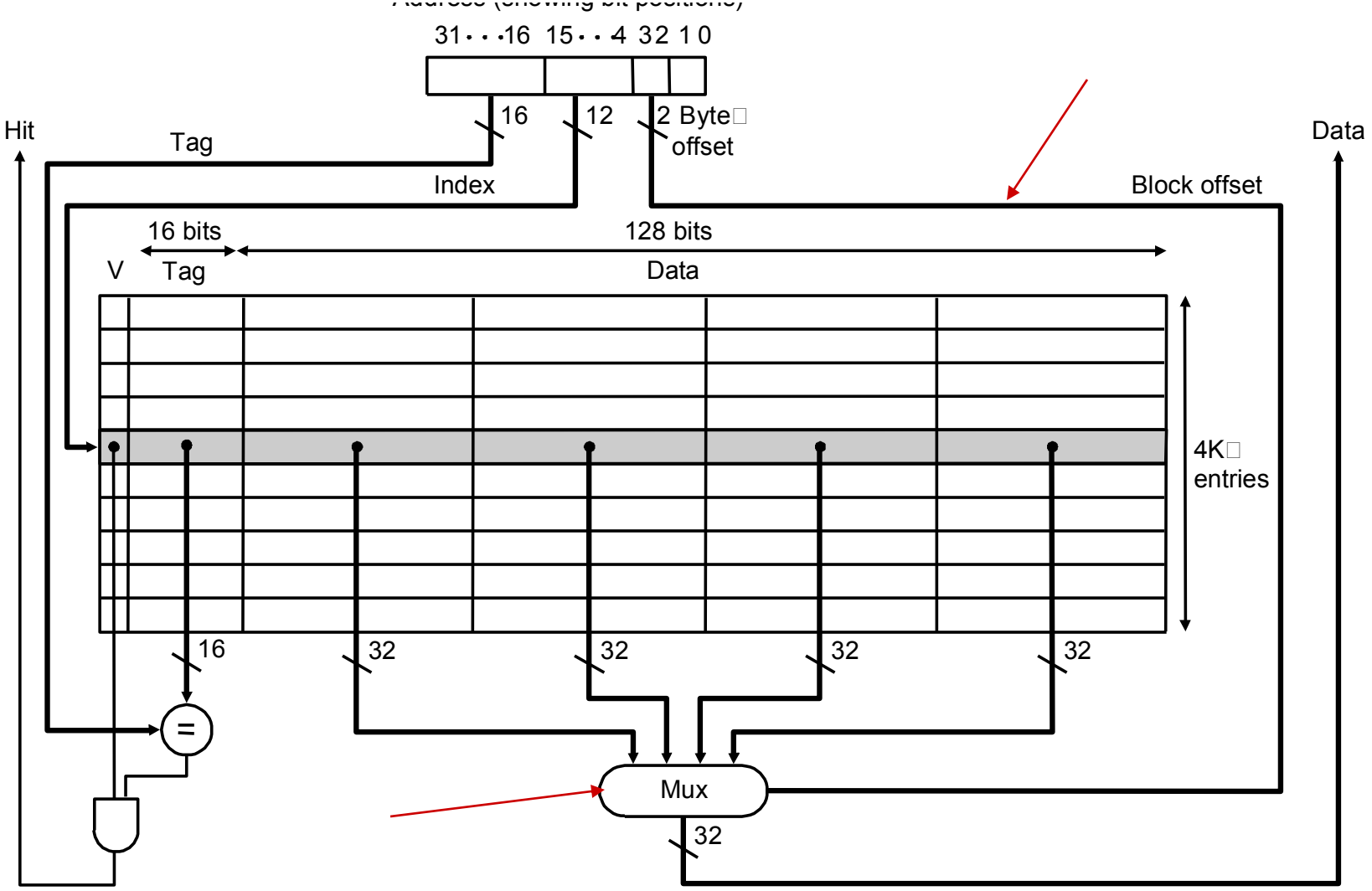


Block Size



- Measurements from real programs
- Bottomline: Block size chosen by experiment, typically 16-128 bytes

Multi-word Cache Blocks

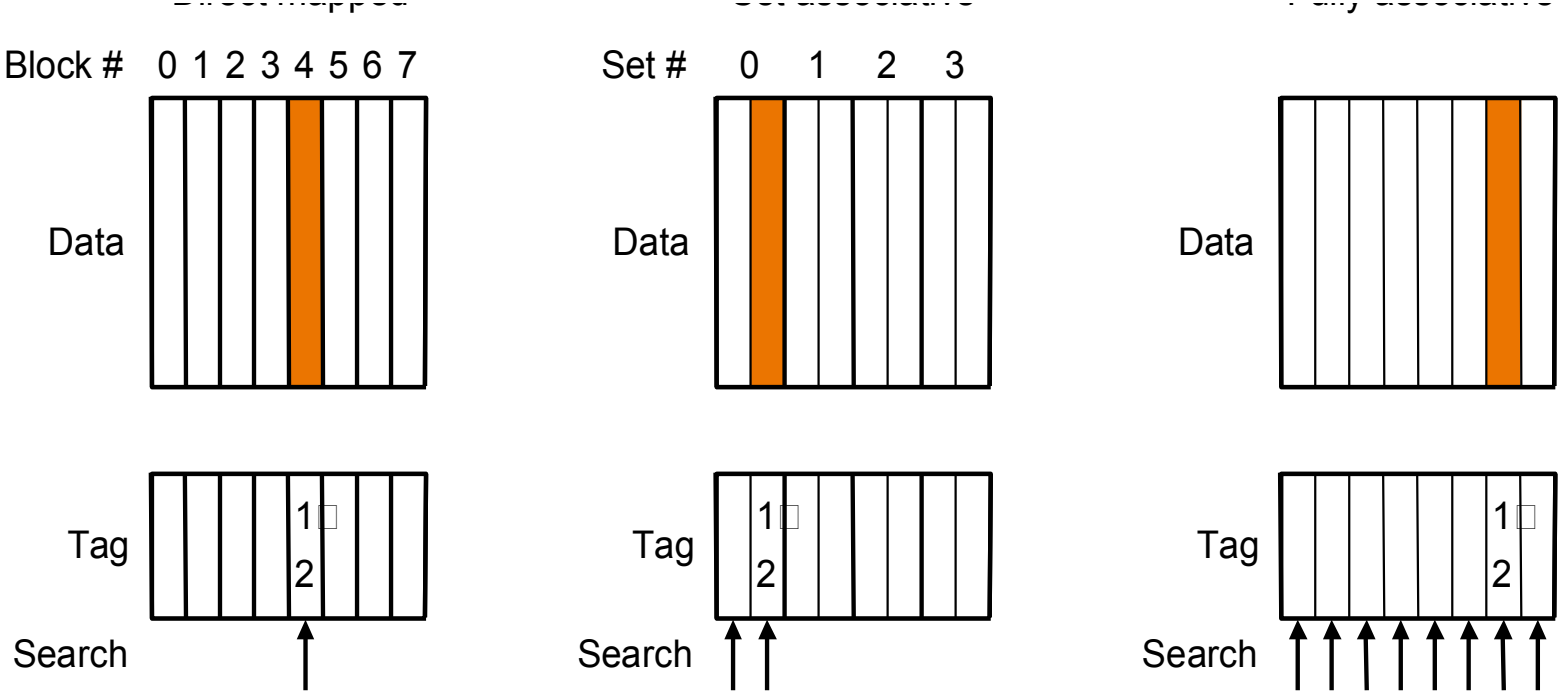


- Use block-offset lines to select word

Four Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

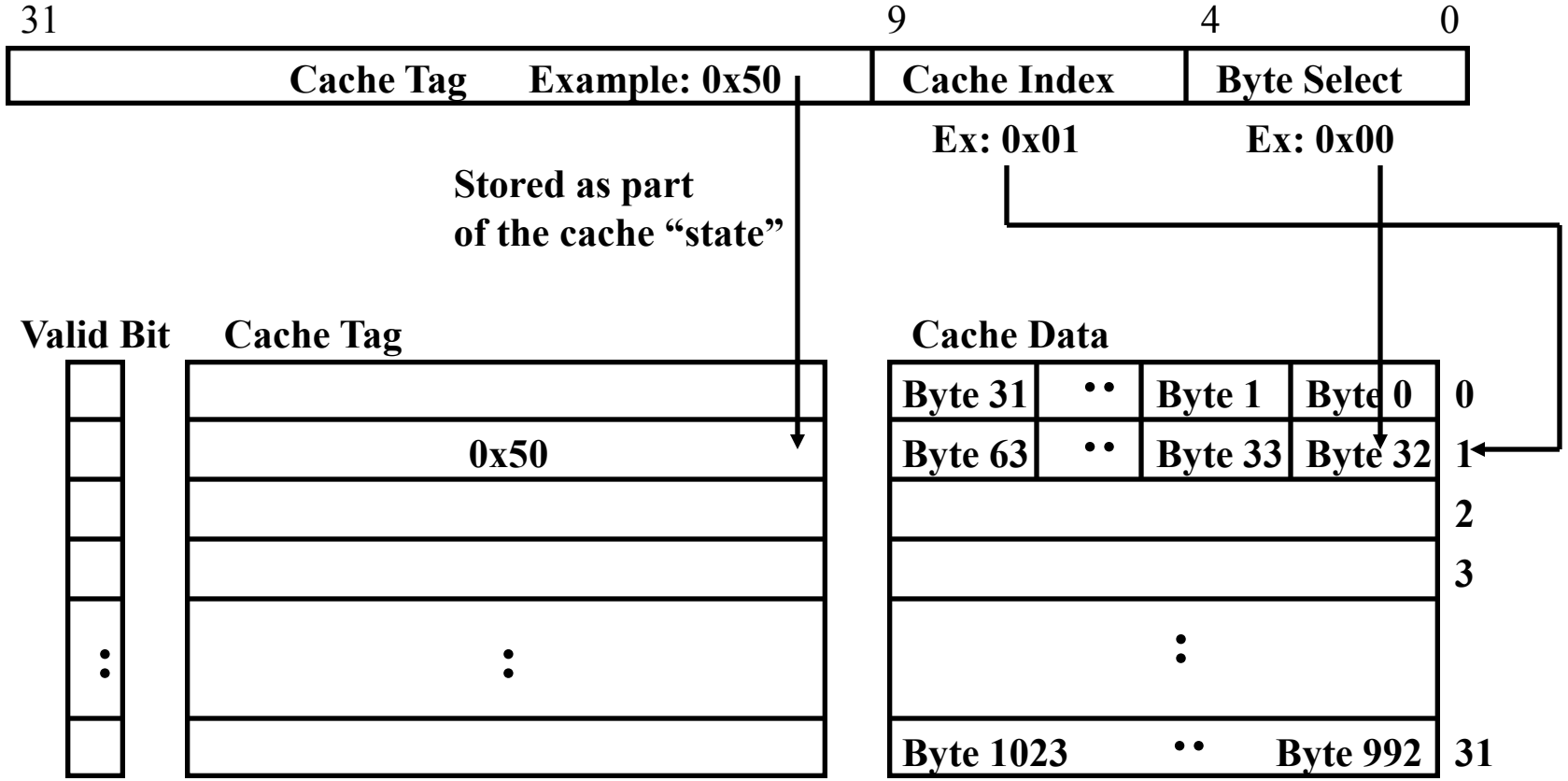
Q1. Block Placement



- In previous example:
  - Block may reside in one fixed frame. (frame[addr mod 16])
- Other points in the design space
- Fully-associative
  - Block a reside in any frame
- N-way set-associative
  - Block may reside in a set of N-frames

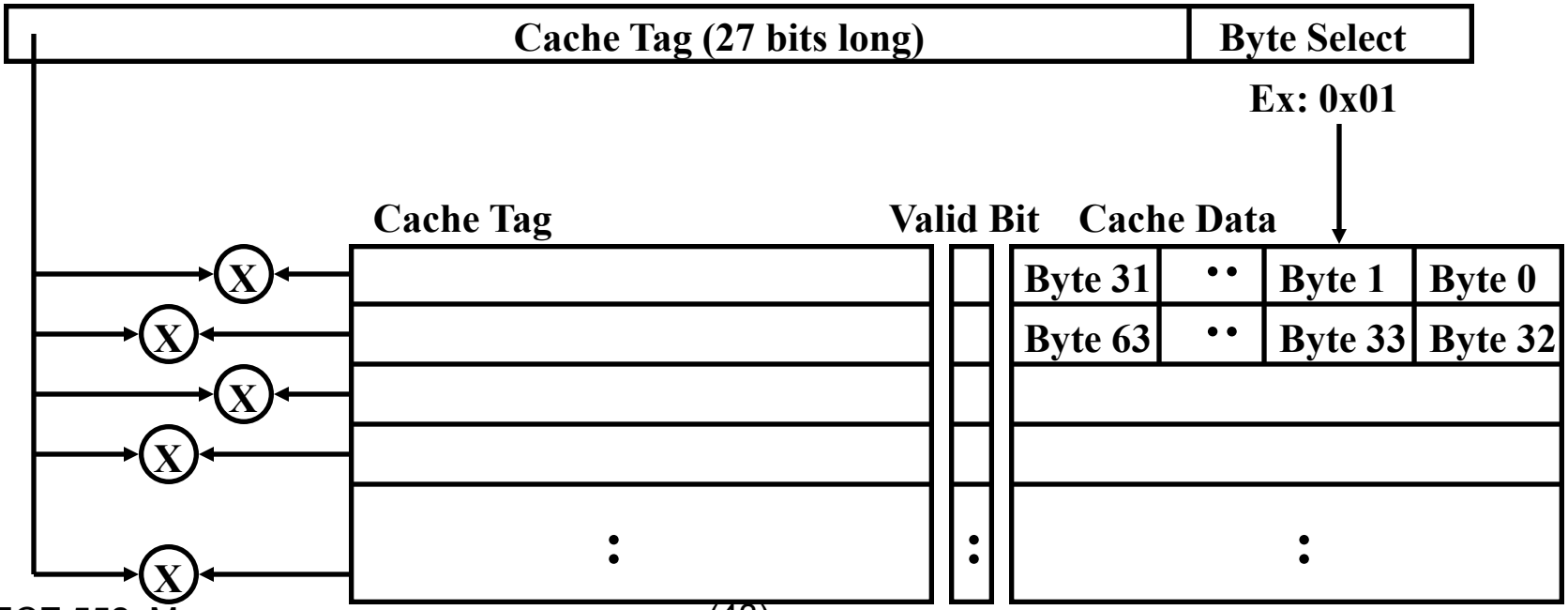
Example: 1 KB Direct Mapped Cache with 32 B Blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



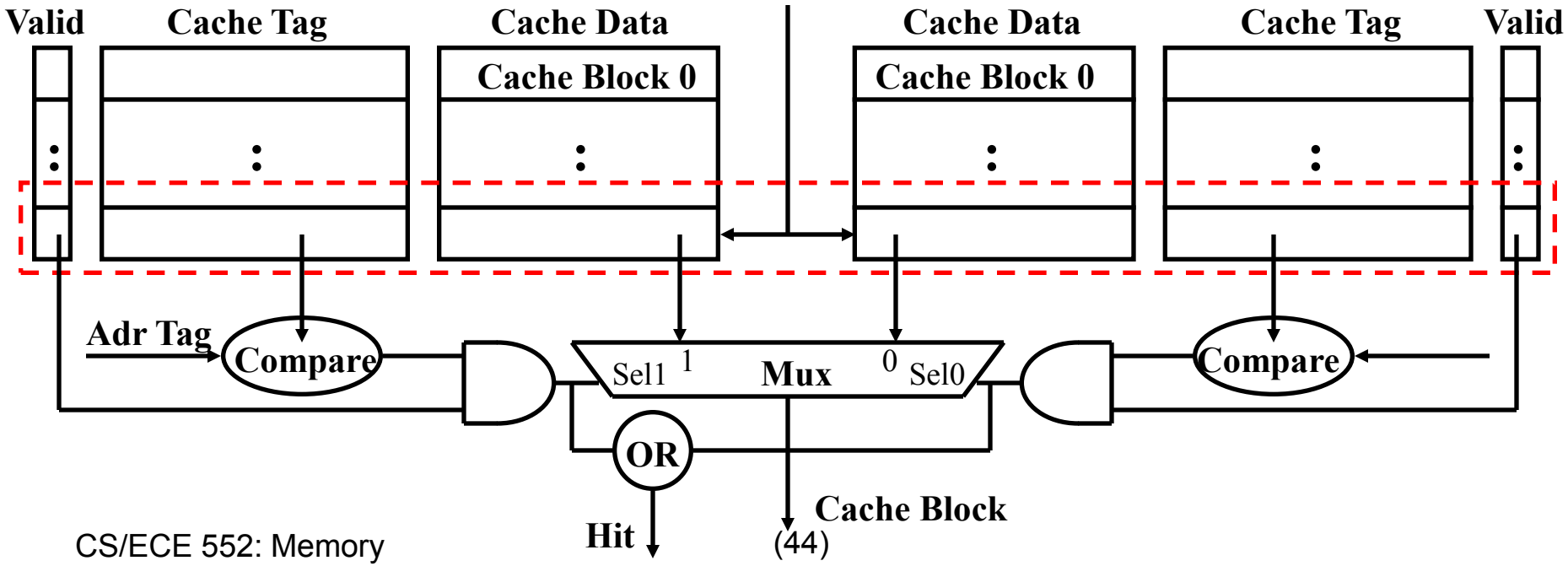
Another Extreme Example: Fully Associative

- Fully Associative Cache
  - Forget about the Cache Index
  - Compare the Cache Tags of all cache entries in parallel
  - Example: Block Size = 2 B blocks, we need N 27-bit comparators
- By definition: Conflict Miss = 0 for a fully associative cache



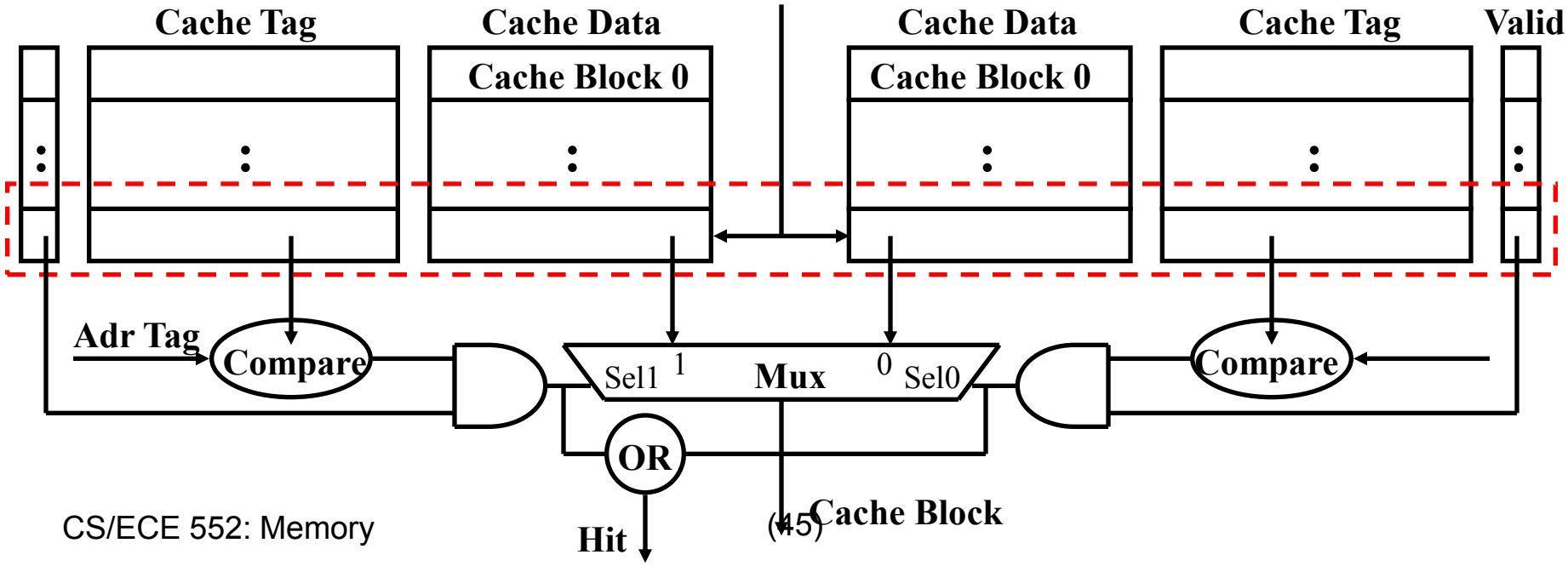
A Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

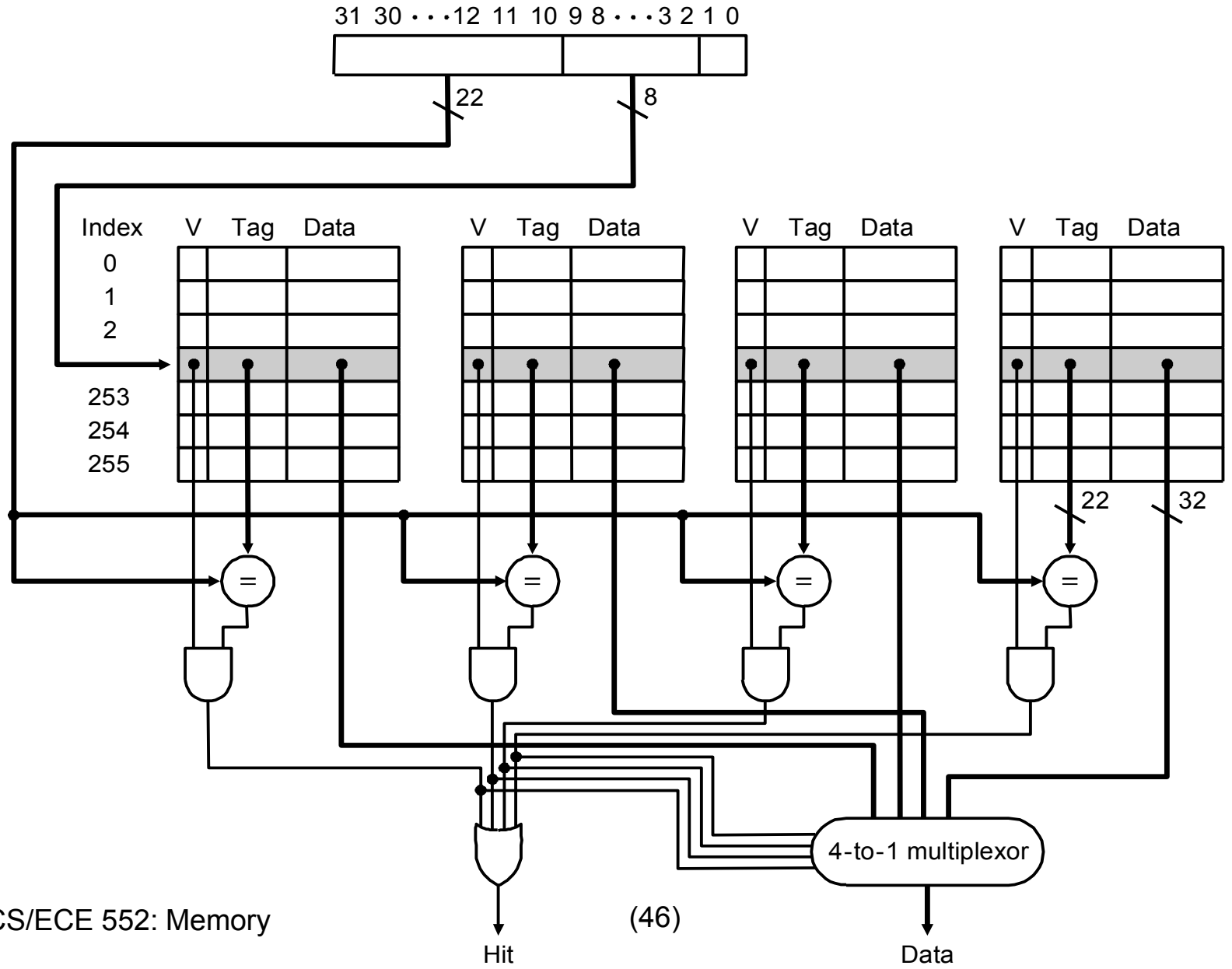


Disadvantage of Set Associative Cache

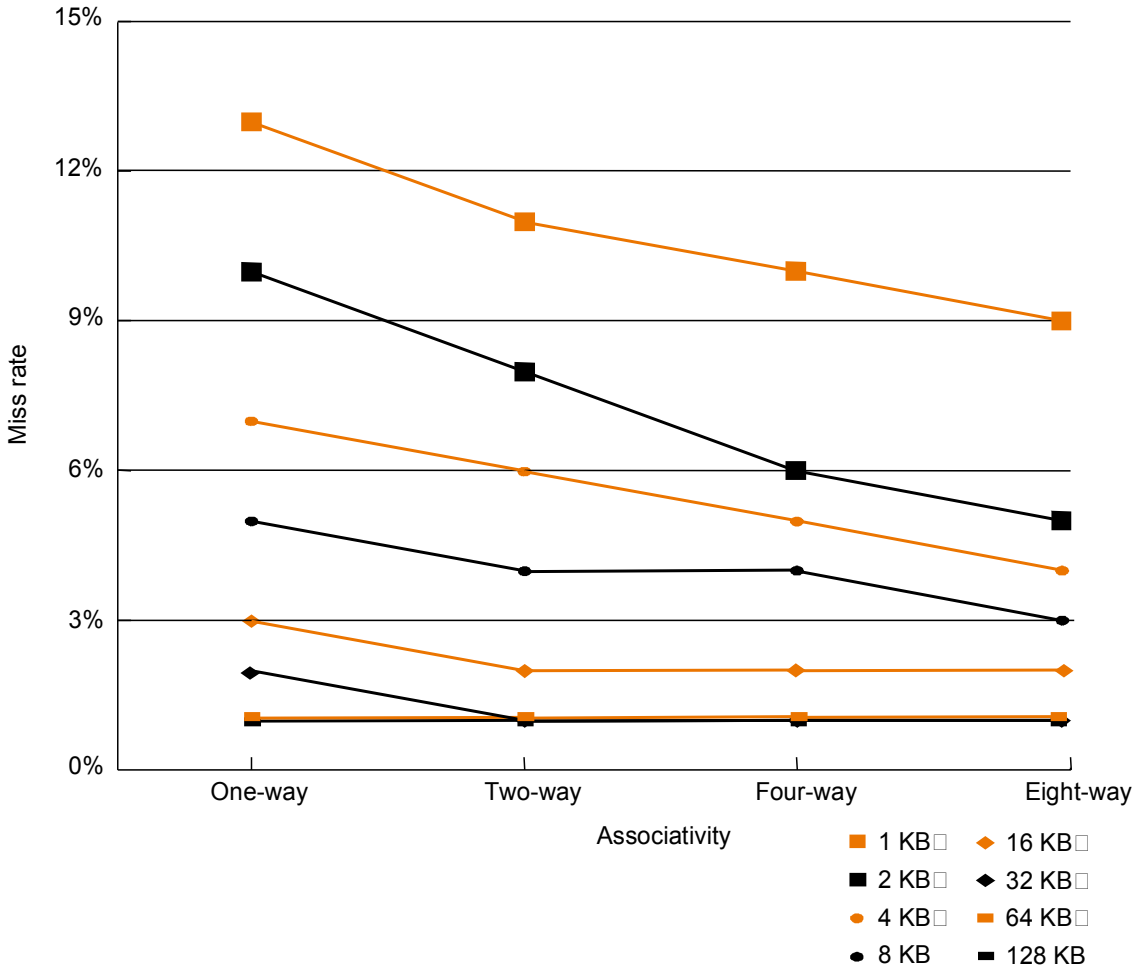
- N-way Set Associative Cache versus Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss decision and set selection
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



4-way set associative cache

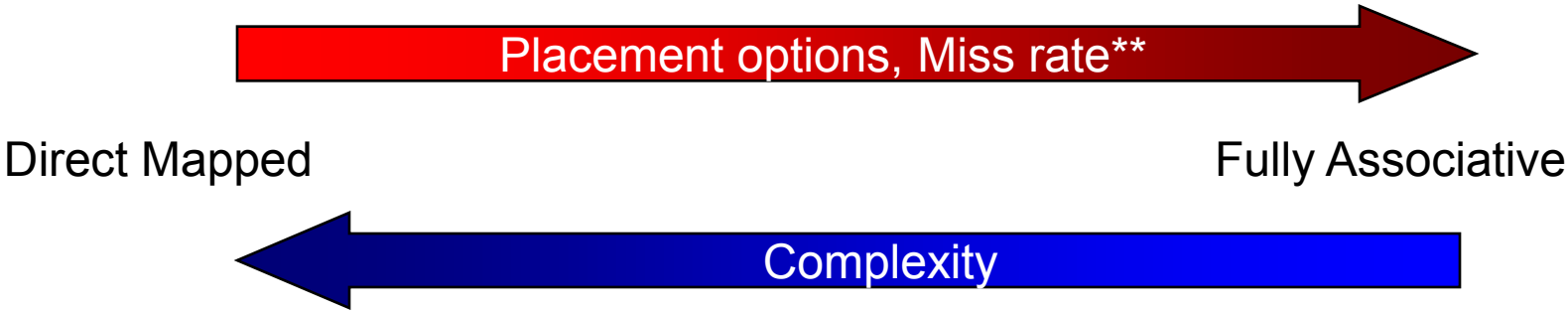


Performance



- A little associativity goes a long way

Cache design spectrum



- Conflict misses reduced with higher associativity
- Associative search is complex, suited for smaller caches
- Increasing associativity:
  - Increases tag bits, shrinks index bits
  - Increases comparator size (~ tag bits)

Exact design

- How to determine:
  - Number of bits for
    - Index, tag and block offset
- Walkthrough example

Cache Design

- Cache size = 32 KB (CS)
- Block size = 32 B (BS)
  - Frames (F) = CS/BS = 1024 (= 1K)
- Associativity = 2-way (A)
  - Number of frames/way = F/A = 512

- Address-bits = 32 bits (Ad)
  - Block-offset bits (b) = lg(BS) = lg(32) = 5
  - Index bits (i) = lg(FpW) = lg(512) = 9
  - Tag bits (t) = Ad – i – b = 32 – 9 – 5 = 18

Cache Design

- Draw the cache organization

4-Questions

- Q1: Where can a block be placed in the upper level? *(Block placement)*
  - In one of N-frames in N-way associative cache
  - N = 1 => Direct mapped
  - N = #frames => Fully associative
  - Setindex = Blocknum (mod numsets)
- Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag match (no need to examine index/block-offset bits --- why?)
  - Valid bit

Q3. Block Replacement

- Q3: Which block should be replaced on a miss?
  (Block replacement)
  - Easy for Direct Mapped
  - Set Associative or Fully Associative:
    - Random
    - LRU (Least Recently Used)
    - Approximate LRU

3C Miss Classification

- Compulsory (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not much you can do about it
  - Note: If you are going to run "billions" of instructions, Compulsory Misses are insignificant
- Conflict (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

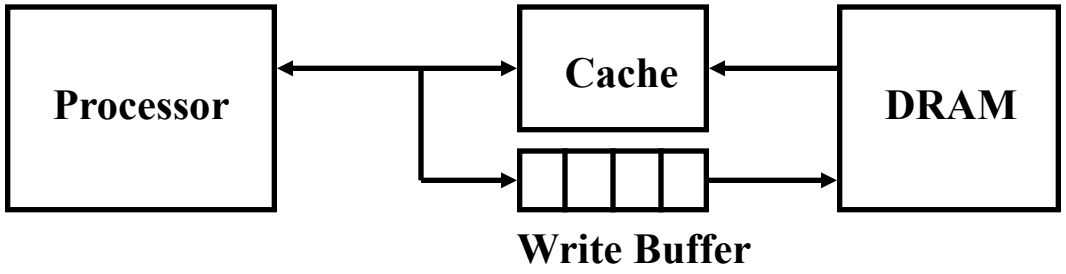
Summary and Lookahead

- Simple case : direct mapped
- Associativity: trade-offs
- Replacement
- Next:
  - Write strategies
  - How to design memory hierarchies?
  - How does software interact with caches?
  - Is programmer aware of the existence of caches?
  - Can programmers benefit by being aware of caches?

Q4. Write strategy

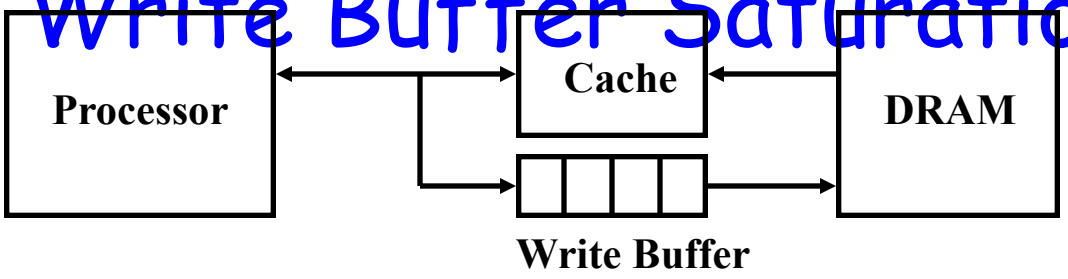
- Q4: What happens on a write?
  (Write strategy)
- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
- WT always combined with write buffers so that don't wait for lower level memory

Write Buffer for Write Through

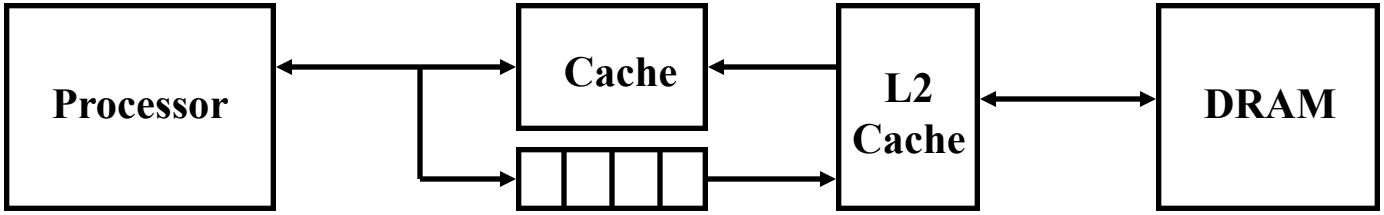


- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

Write Buffer Saturation

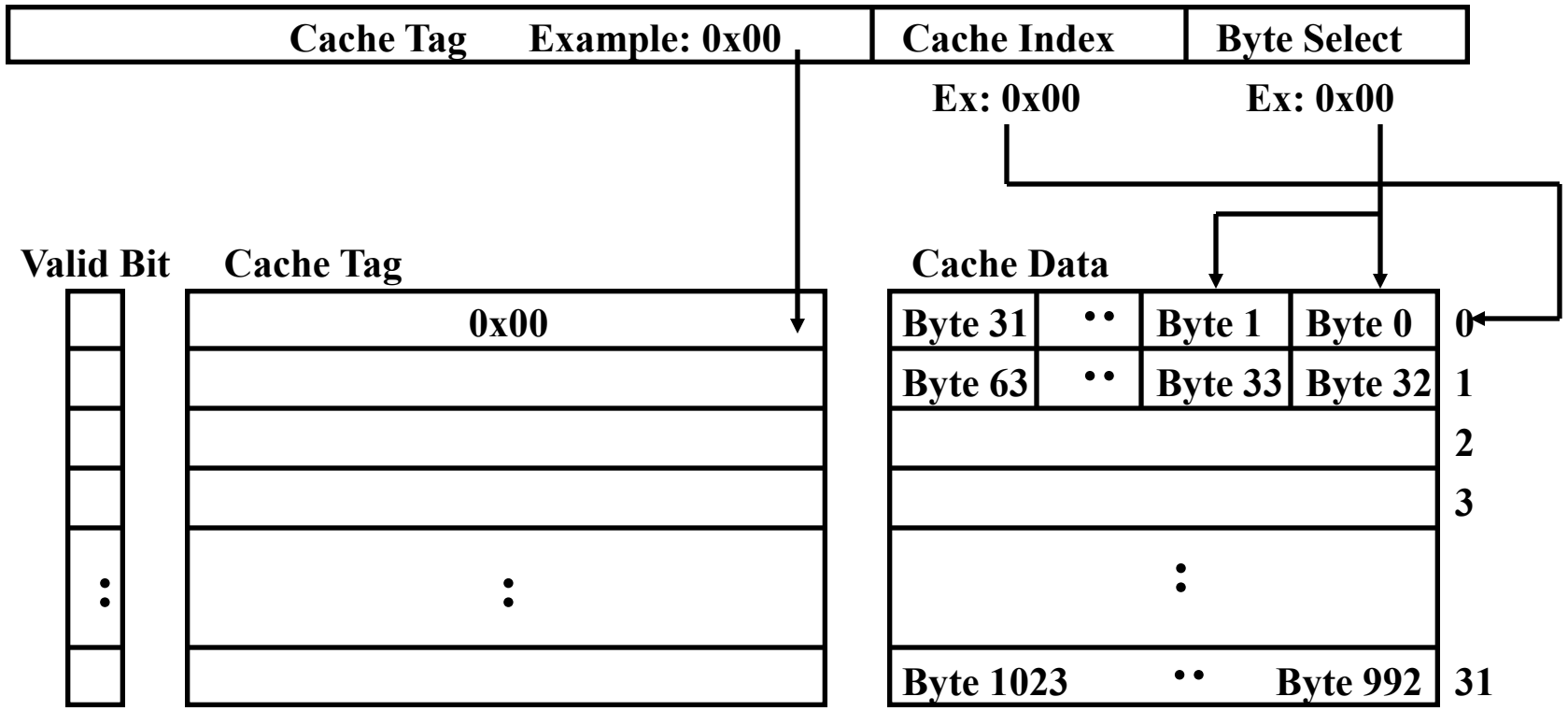


- Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - If this condition exist for a long period of time (CPU cycle time too quick and/or too many store instructions in a row):
    - Store buffer will overflow no matter how big you make it
    - The CPU Cycle Time <= DRAM Write Cycle Time
- Solution for write buffer saturation:
  - Use a write back cache
  - Install a second level (L2) cache:



Write-miss Policy: Write Allocate versus Not Allocate

- Assume: a 16-bit write to memory location 0x0 and causes a miss
  - Do we read in the block?
    - Yes: Write Allocate
    - No: Write Not Allocate



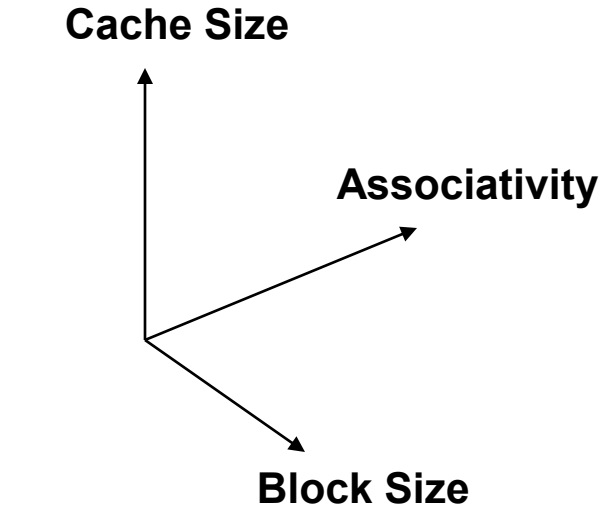
Improving Cache Performance

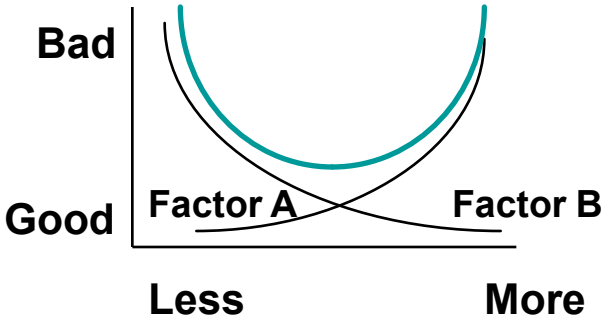
- Reduce Hit time
  - small and simple-> direct mapped
- Reduce miss rate
  - Large cache, large blocksize, associative,
- Reduce miss penalty
  - Reduce block-size

- Remember Amdahl's law
  - Common case : hit
  - Reduce miss-rate at the cost of hit time

Cache design space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





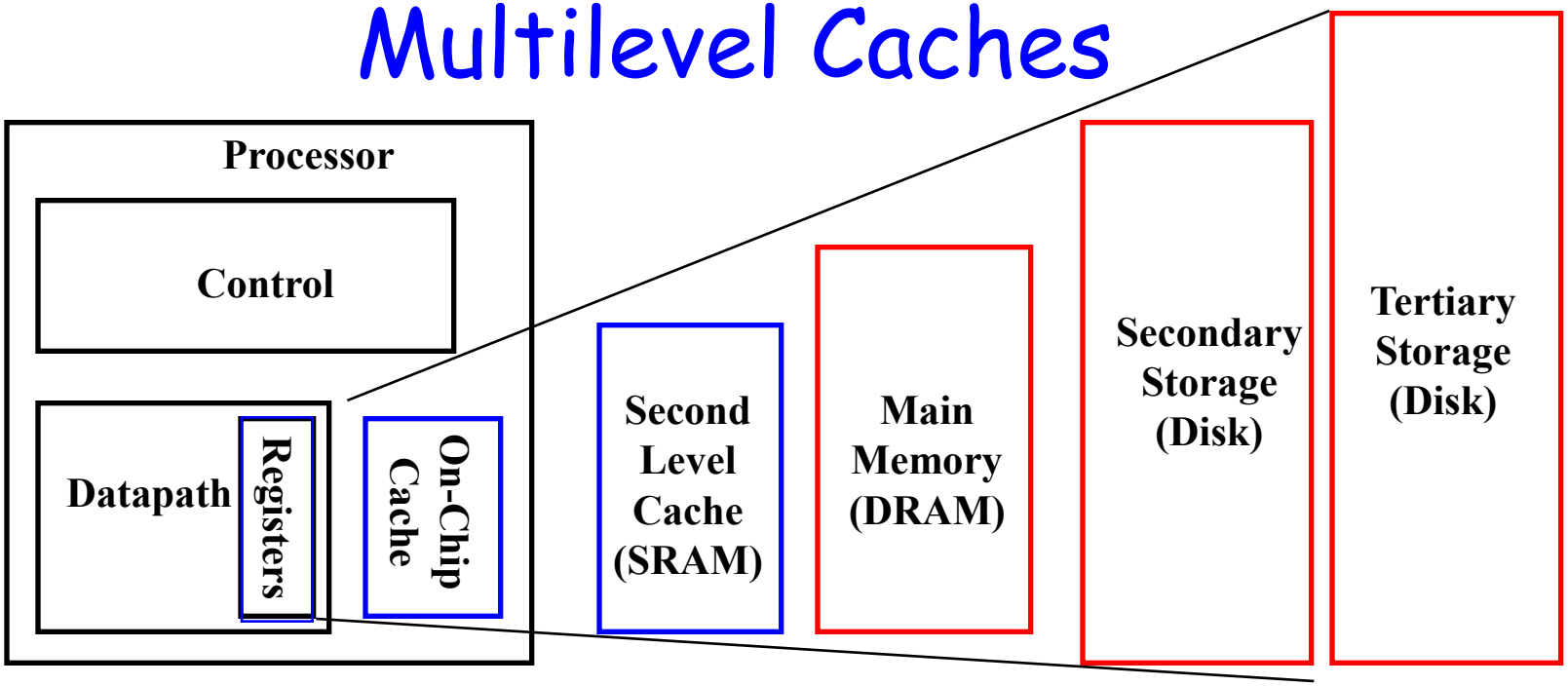
Practical design issues

- Split Cache vs. unified cache
- Multi-level Caches

Split caches

- One for instruction, one for data
- Split cache
  - Instructions account for 75% of mem accesses
  - I-missrate = 5%, D-missrate = 6%
  - AMAT = (1 + 0.05*10)*0.75 + (1 + 0.06*10) * 0.25
- Unified Cache
  - Aggregate missrate = 4%
  - AMAT = (1 + 0.04*10) = 1.4???
  - For modern pipelined processor:
    - single-memory structural hazard

Multilevel Caches



- $AMAT_{L1}$ = hit $time_{L1}$ + $miss\text{-}rate_{L1}$ * $miss\text{-}penalty_{L1}$
- What is $miss\text{-}penalty_{L1}$?
  - Access time of memory
- Put in a large L2 cache between L1 and memory
  - What is the $miss\text{-}penalty_{L1}$?
  - $AMAT_{L2}$ = hit $time_{L2}$ + $miss\text{-}rate_{L2}$ * $miss\text{-}penalty_{L2}$

Multilevel Caches

- Cycle time = 1ns (~ 1GHz clock)
- Main memory access = 100ns = 100 cycles
- L1 miss rate = 5%
- Without 2nd level cache
  - $AMAT_{L1}$ = 1 + 5% * 100 = 6 cycles
- With 2nd level cache
  - L2 miss-rate = 2% (local miss-rate)
  - L2 hit time = 10 cycles
  - $AMAT_{L2}$ = 10 + 2% * 100 = 12 cycles
  - $AMAT_{L1}$ = 1 + 5% * 12 = 1.6

State of the Art

- 2-3 levels of SRAM cache
- Split I- and D-caches at Level 1
- 2-4-way set-associative at Level 1
- 2-8-way set-associative at higher levels

Summary

- Memory technology (Capacity/cost/speed)
- Need for hierarchy
- Performance
  - AMAT, ideal vs. real CPI
- Cache management:
  - Associativity, indexing, write handling, multi-word blocks etc.
- Diagrams of arbitrary cache organizations
- Next:
  - Cache-friendly coding techniques
  - Virtual Memory

Error Correcting Codes (ECC)

- Low Probability of Bit Flipping X Vast Memory
- = Substantial Probability of a Few Bits Wrong

- Model
  - Assume small number of random errors
  - So for a single word (e.g., 64 bits)
  - P(no flips) >> P(1 flip) >> P(2 flip) >> P(>2 flips)

- Actions
  - Single Error Detection (Parity)
  - Single Error Correction with Double Error Detection (SECDED)
  - More in Future

Naïve 1-Bit ECC

- Store 1-bit dataword {0, 1} in longer codeword
- Single Error Detection (Parity)
  - Save 0→00, 1→11
  - Read 00→0, 11→1, {01,10}→error
- Single Error Correction
  - Save 0→000, 1→111
  - Read {000,001,010,001}→0,
    {111,011,101,110}→1

Naïve 1-Bit ECC, cont.

- Single Error Detection with Double Error Correction (SECDED)
  - Save 0→0000, 1→1111
  - Read {0000,0001,0010,0100,1000}→0,
    {1111,1110,1101,1011,0111}→1,
    {two zeros + two ones}→error
- Note
  - 4 bit flip between legal datawords
  - Must be true for SECDED code

Hamming Distance & Code Strength

- Hamming Distance
  = # bit flips between datawords
  - 00→11 (SED) ➔ Hamming = 2
  - 000→111 (SEC) ➔ Hamming =3
  - 0000→1111 (SECDED) ➔ Hamming = 4

- But 300% memory overhead?
- Build Code on Multi-bit data word

SECDED Memory Overhead

# Data Bits	# Check Bits	Overhead
1	3	300%
8	5	63%
32	7	22%
64	8	13%
128	9	7%
n	$1+\log_2 n+a\_little$	

ECC Process (work example on board)

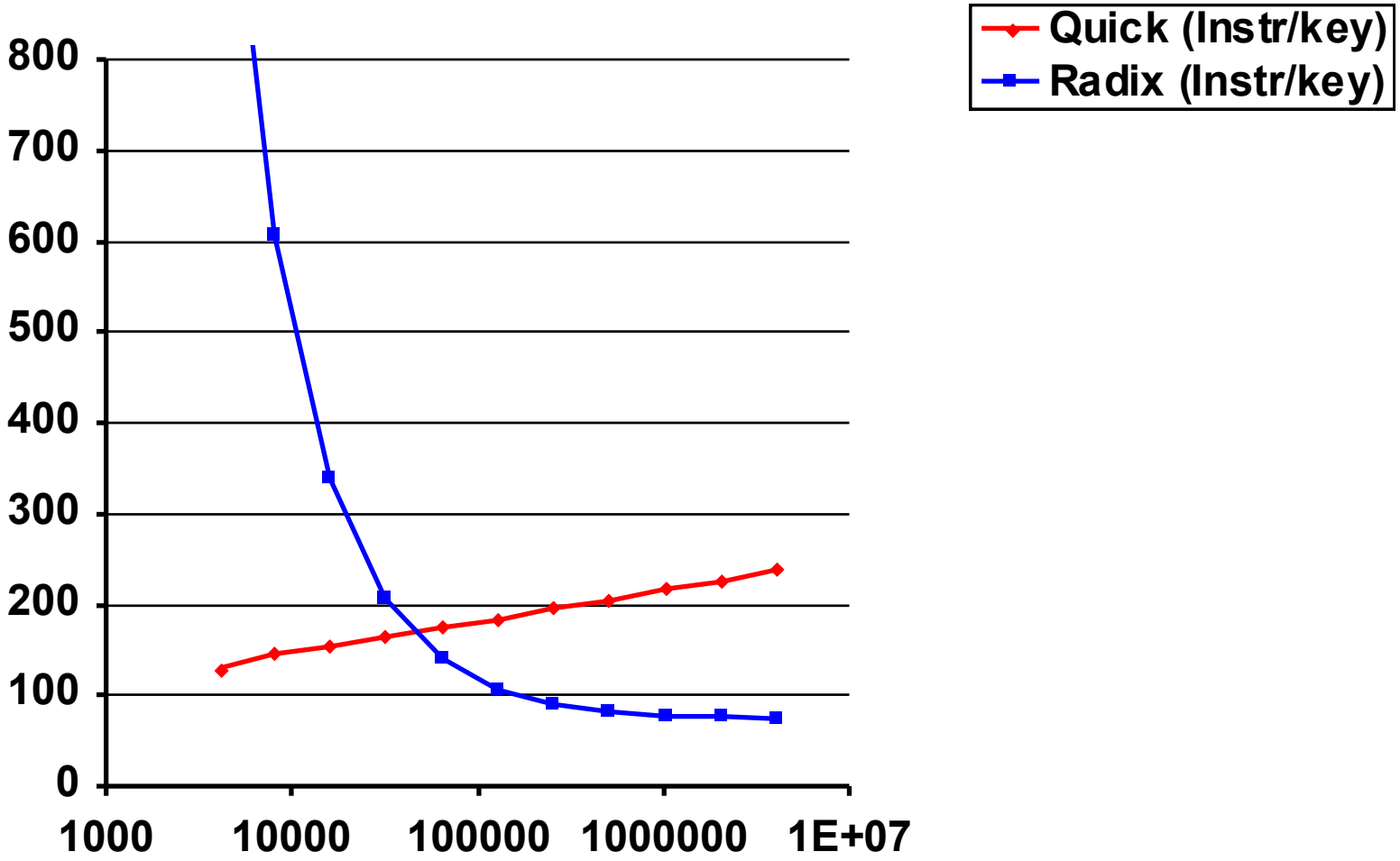
- Start with dataword D
- Compute checkbits C = f(D)
- Store codeword CD
- Error(s) may occur CD'
- Read codeword CD'
- Recompute checkbits C' = f(D')
- Compute S = C xor C'
  - S==0 ➔ return dataword D
  - S!=0 ➔ correct D' to D & return D
- Add additional parity for DED
- Works even if bit flip is in C or additional parity

Software Interaction

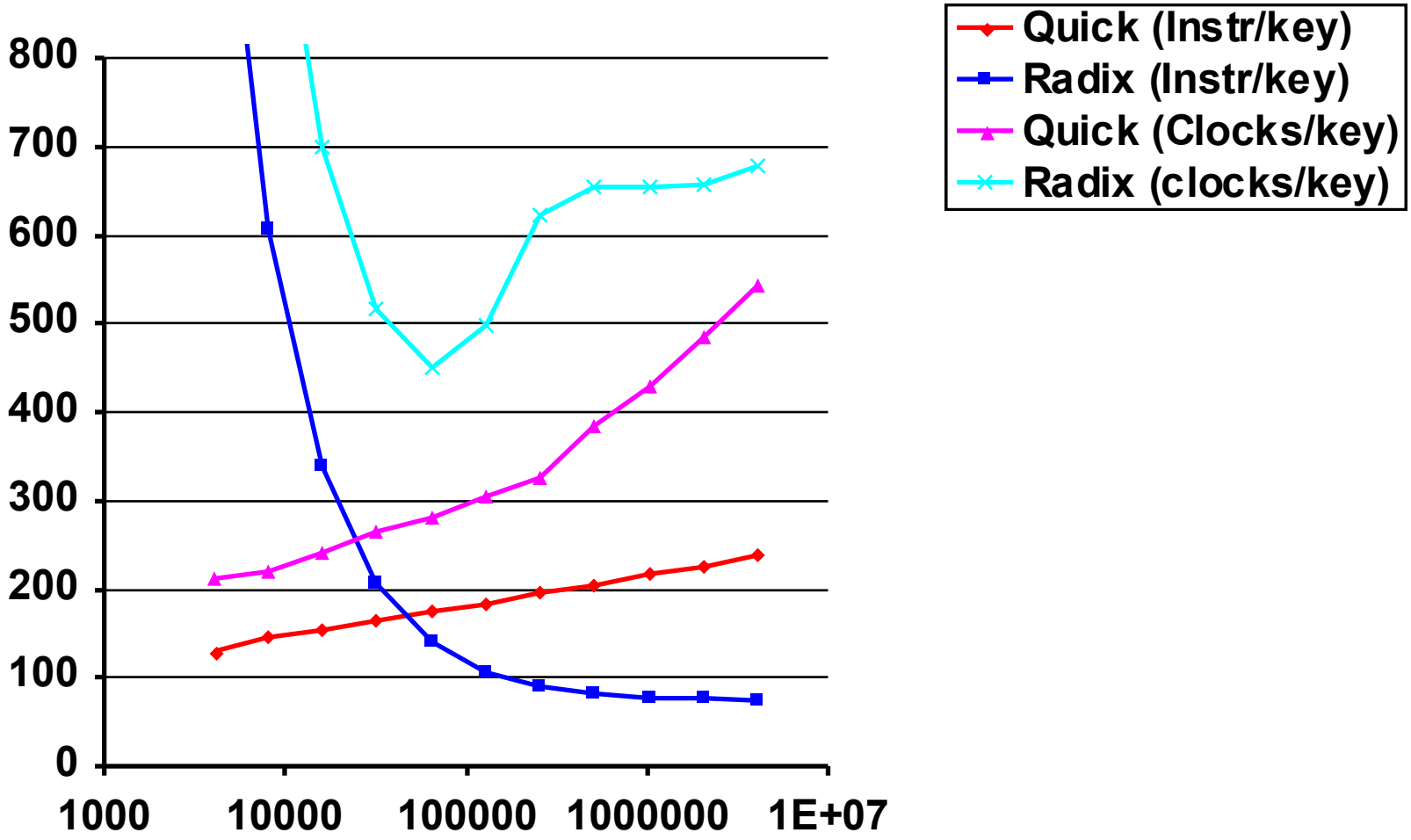
- RAM model of computation
  - All memory accesses take the same amount of time
  - Theoretical Model – has nothing to do with DRAM
- Reality:
  - Caches introduce non-uniformity
  - Hits take less time than misses

- Quicksort
  - fastest comparison based sorting algorithm when all keys fit in memory: $\Theta(n \lg(n))$
- Radixsort
  - also called "linear time" sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys: $\Theta(n)$

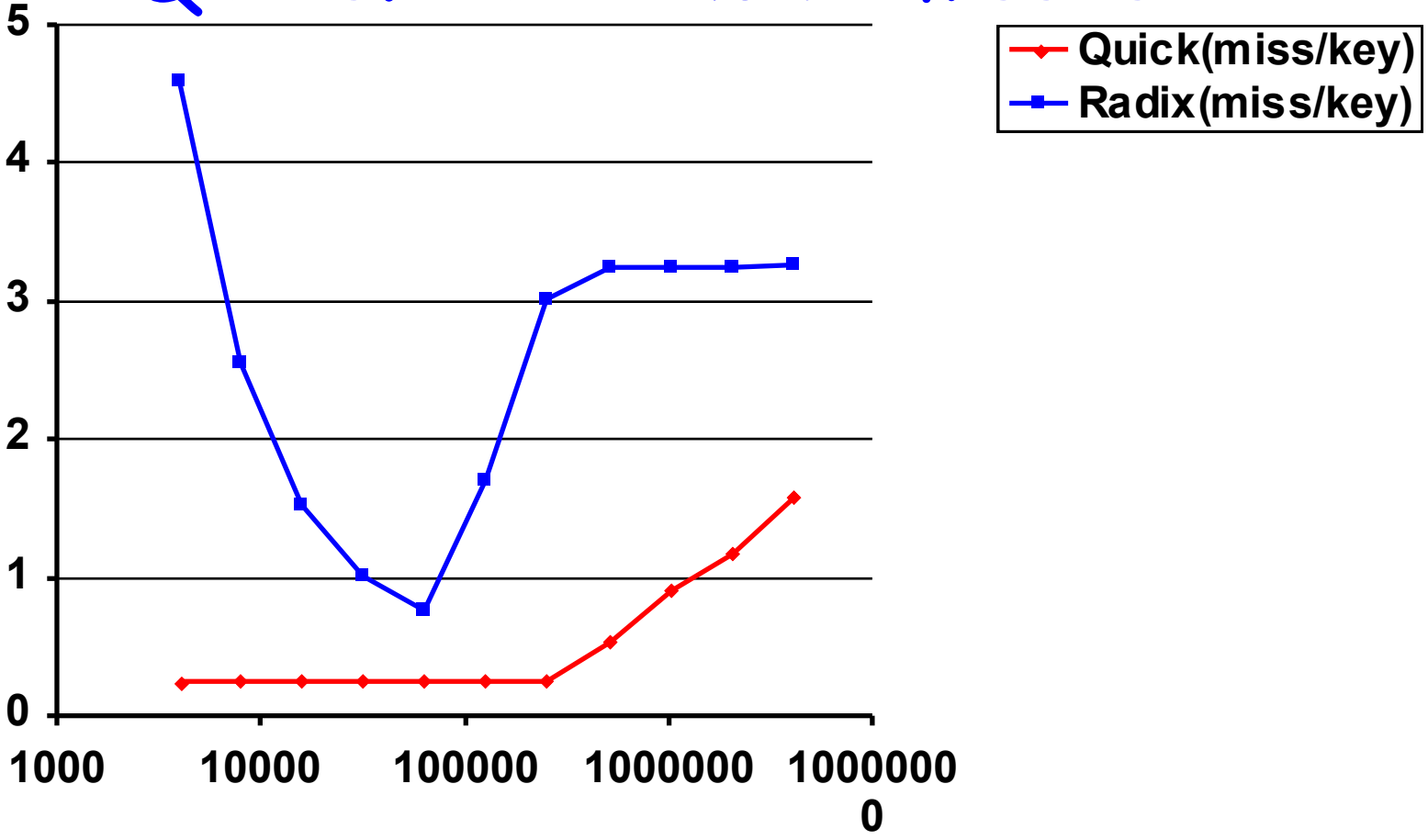
QS vs. RS : Instructions



QS vs. RS : Time, Instructions



QS vs. RS : Cache misses



- RAM model results are still valid… but at much larger input sizes
- How does one create practical, fast algorithms?
- Cache-aware programming/compilation

Data Cache Performance

- Instruction Sequencing
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down entire columns or rows
- Data Layout
  - *Merging Arrays*: Improve spatial locality by single array of compound elements vs. 2 separate arrays
  - *Nonlinear Array Layout*: Mapping 2 dimensional arrays to the linear address space
  - *Pointer-based Data Structures*: node-allocation

- Example walkthrough: Loop fusion, Blocking, Merging Arrays

#1 : Loop Fusion

- Coverts distant reuse to near reuse
- Enhances temporal locality
- Code Transformation

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
    D[i] = max( A[i], B[i] );
}
```

- - - - - - - - - - - - - - - - - -

```
for(i=0;i<64;i++) {
    C[i] = min( A[i] , B[i] );
    D[i] = max( A[i], B[i] );
}
```

#2: Array merging

- Eliminates conflicts
  - Array of compound structure vs.
  - multiple arrays of simple data
- Enhances spatial and temporal locality
- Data layout transformation

```
for(i=0;i<64;i++) {
      C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
      D[i] = max( A[i], B[i] );
}
```

```
Struct merge {
    int  A;
    int  B;
};
Struct merge M[64];
for(i=0;i<64;i++) {
    C[i] = min( M[i].A , M[i].B);
}
for(i=0;i<64;i++) {
    D[i] = max( M[i].A, M[i].B);
}
```

#3: Blocking (Tiling)

- Exploits re-use across loops
  - Divide into pieces that fit in the cache vs.
  - Marching through whole array
- Capacity misses
- Code Transformation

```
for(i=0;i<64;i++) {
      C[i] = min( A[i] , B[i] );
}
for(i=0;i<64;i++) {
      D[i] = max( A[i], B[i] );
}
```

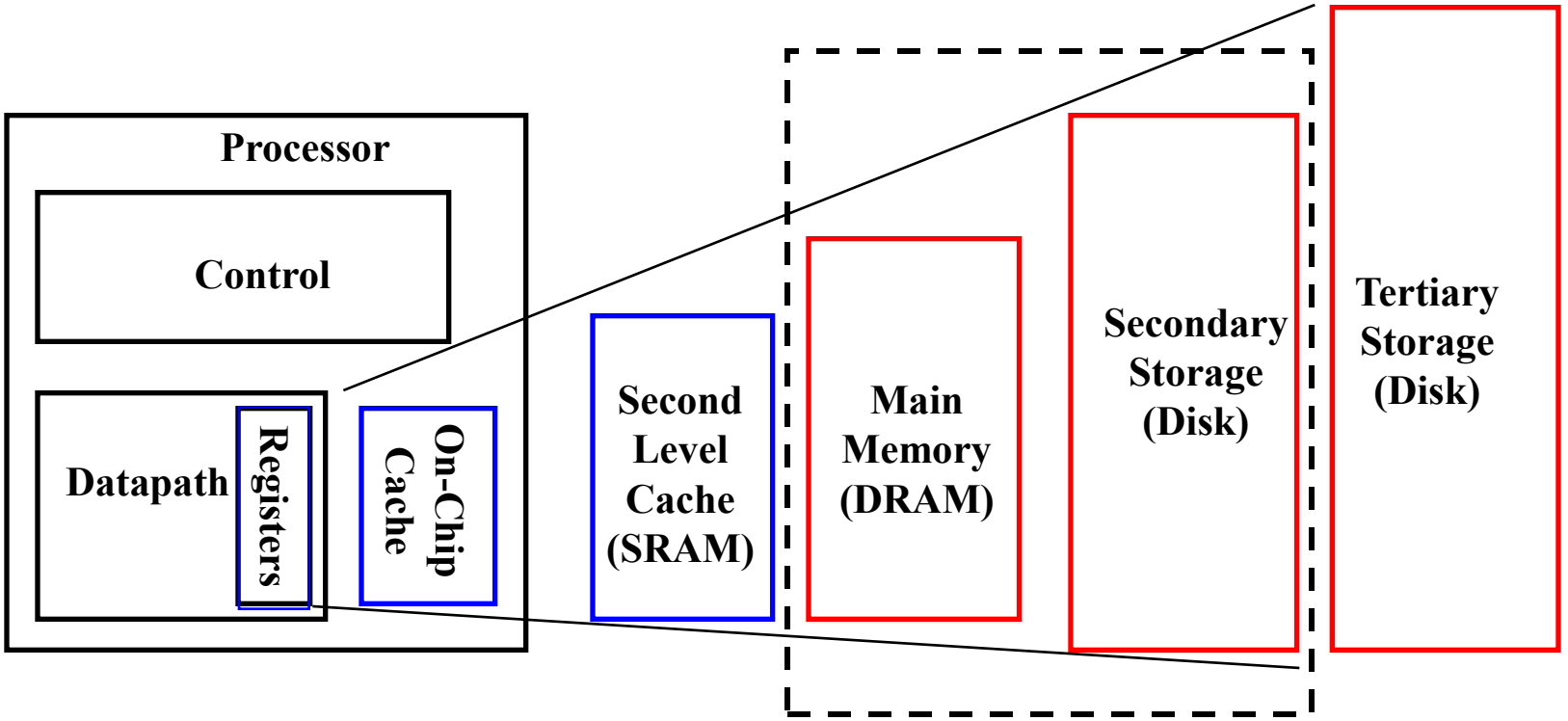
```
for (j=0; j<=2;j++)
{
  for(i=0;i<32;i++) {
      C[32*j + i] = min( A[32*j + i] , B[32*j + i] );
  }
  for(i=0;i<32;i++) {
      D[32*j + i] = max( A[32*j+ i], B[32*j + i] );
  }
}
```

State of the practice

- Cache friendly programming challenges
  - No global view of application
  - Different cache sizes

- Analyze programs after they're written
  - Find bad access patterns
  - Fix them
  - Rinse and repeat

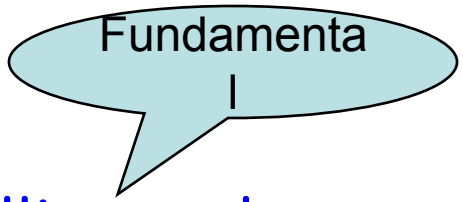
Virtual Memory



- Data movement between Disk and Main memory
- We know how layers of hierarchy interact
  - Cache and main memory
- Can we apply all the same techniques?
  - Similarities and differences

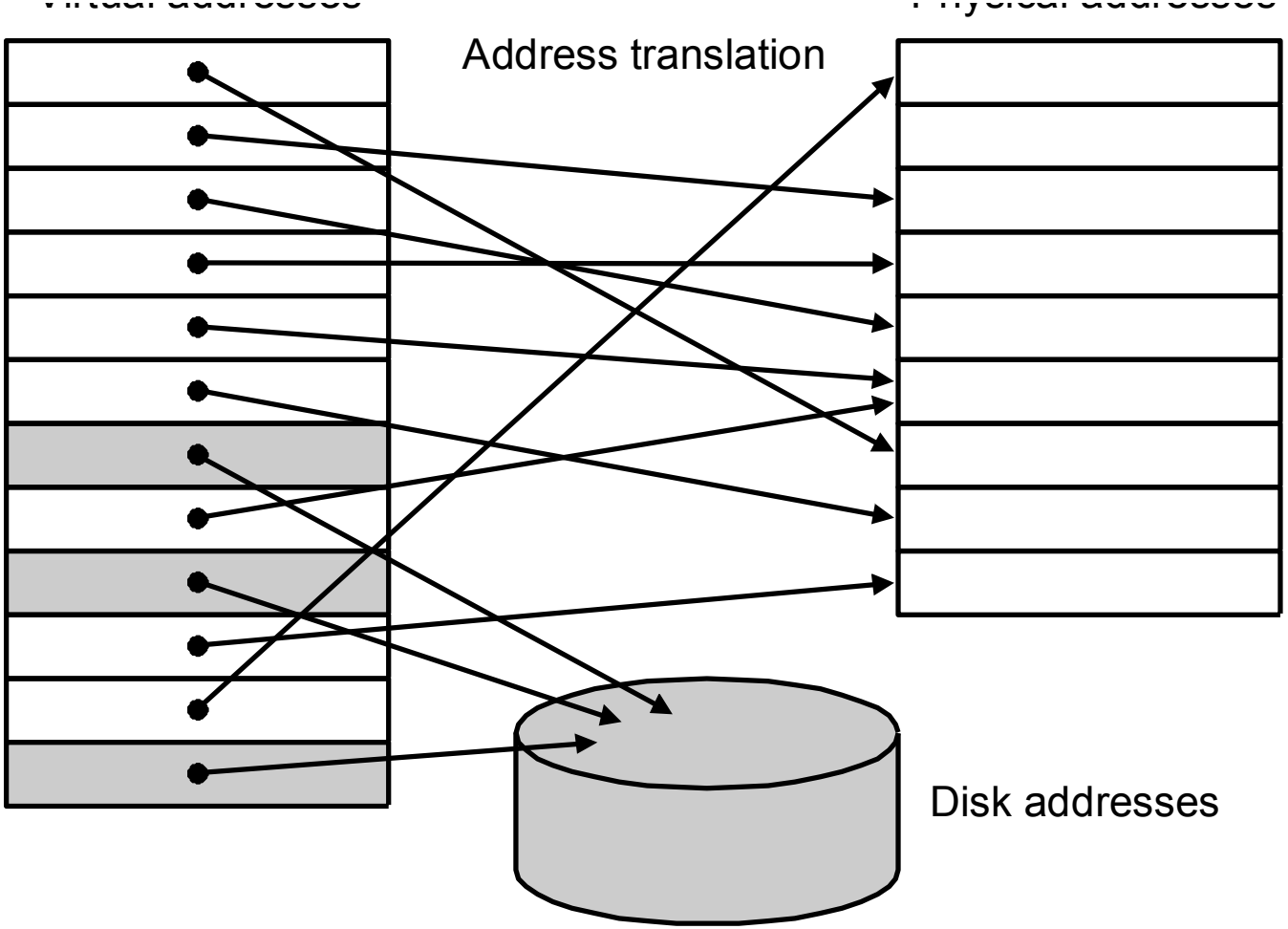
Virtual Memory

- Similarities:
  - Mapping a larger address space to a smaller space
  - Used for data movement between layers of the memory hierarchy



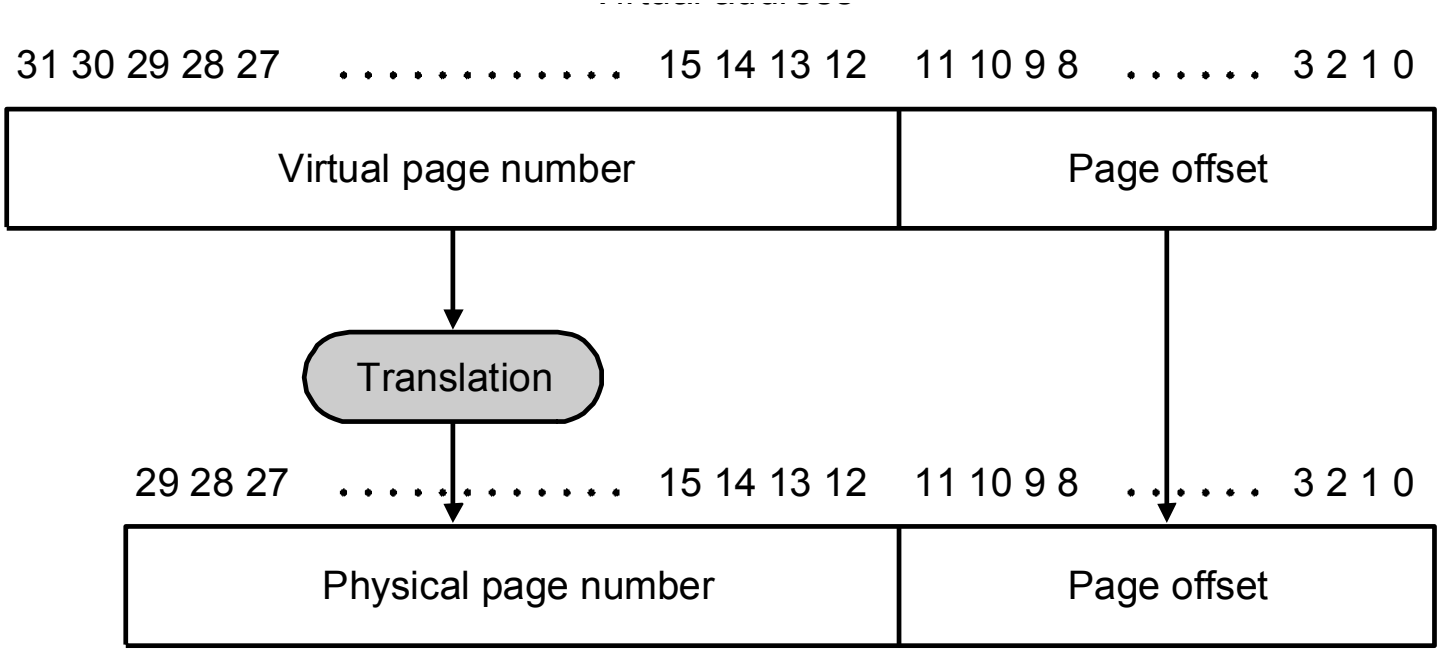
- Differences:
  - Miss-penalty : 10-100 cycles vs. ~1 million cycle
  - Block size : 16-64 bytes vs. 4 KB – 8 KB
  - Full associativity (But no associative search!)
  - Software handling

VM Operation



- Programs use virtual address
- The data/code resides elsewhere (physical address)

VM Operation



Physical address

- Assume 4KB pages
- 32-bit VA and 30-bit PA
- System responsible for translation
- E.g.
  - lw $r2, 0xffffc004
  - 0xffffc -> 0x20000 # VPN-> PPN translation
  - PA = 0x20000004

VM Advantages

- Application's view of memory
  - Large: ~4GB in Pentium (32b address)
  - Exclusive: Only program in memory
- System view
  - Smaller: 256MB-1GB
  - Multiple programs share memory
  - Run in a protected manner (memory is private **)
  - Address range is fixed (starts at 0x0)
  - Do not bring in entire program
    - Bring in relevant parts as needed
- VM reconciles these conflicting "views"
  - Not just the classical benefits of hierarchy
  - Illusion of
    - speed of expensive level
    - capacity and cost of cheaper level

VM Terminology

- Blocksize ~ 512B - 8KB+
  - Block : Page
- Miss : Page-fault
  - Fetch from disk
- Derivative properties:
  - Fundamental constraint: high access latency

Back to 4 questions

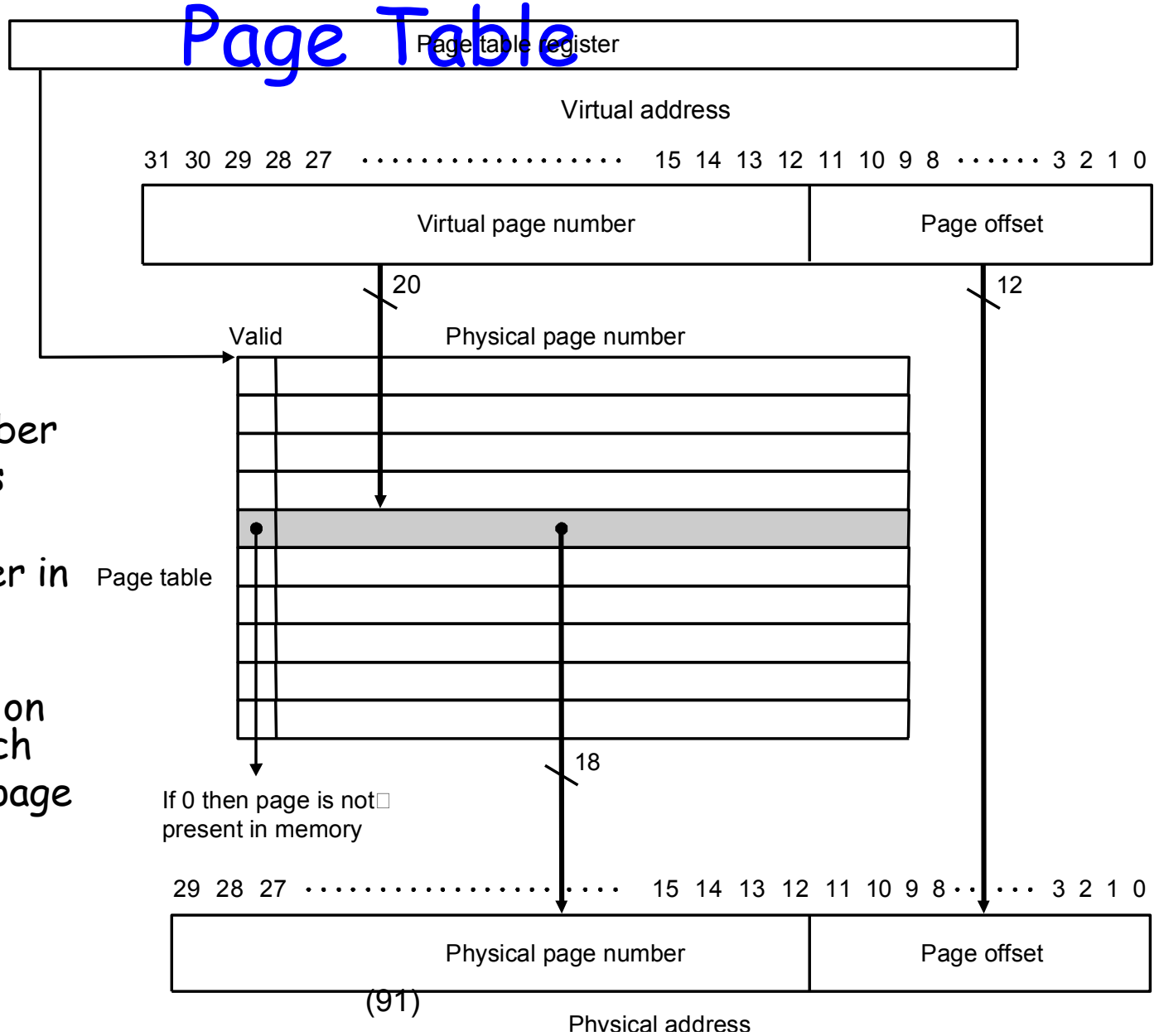
- Q1. Where does a block go?
  - Fully associative
    - Block offset and tag
    - NO INDEX
  - Why?
    - AMAT = access-time + miss-rate*miss-penalty
    - Miss-penalty = ~1 million cycles
    - Have to minimize miss-rate

Q2. Block Identification

- Q2. Block identification
  - Fully associative search??
    - 30-bit physical address (1GB)
    - 4 KB pages
    - Number of frames = 230/212 = 218
    - 512 K frames
    - Compare 512 K frames in parallel??!!
  - Reframe question:
    - Old: Is this VA in any given frame? => Parallel search
    - New: Where is this VA? => Table lookup
      - Page table

Page Table

- Virtual page number
  - Tag in caches
- Physical address
  - Frame-number in caches
- PTBR
  - Change value on context switch
  - Per-process page table



Page table

- Where does table reside?
  - Main memory
  - 100% overhead?
    - Each memory reference now generates two memory references
    - lw $r2,0xffff0004
      - Access page table entry for 0xffff0, get PPN
      - Access PPN:004
  - We want to minimize main memory access!
    - Page table entries can be cached like ordinary data
    - But wait: You need an address to access the cache ***!!
    - Special cache for Page table entries

Page Table Entries

- What does a page table entry contain
  - Physical page number (18 bits)
  - Access control (read/write permissions)
  - Valid bit (1 bit)
  - Misc
    - Use bit for replacement
    - Dirty bit for write-back
  - ~ 4 bytes (1 word)

Size of Page Table

- What is the size of the page table for a system with
  - 32 bit addresses
  - 256MB physical memory
  - 4KB pagesize

  - Virtual pages = $2^{32}/2^{12} = 2^{20}$
  - Physical pages = $2^{28}/2^{12} = 2^{16}$
  - PT size (per process) = (Entry size) * (# entries)
    = 4 bytes * 2^{20} = 2^{22} bytes = 4 MB

- # of processes? ~50 on my WinXP Pro machine
  - 200 MB for page tables?
  - "Big government" : 80% consumed for administration!!
  - Techniques to reduce overhead
    - Gradual growing
    - Inverted PT : entries per physical page

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

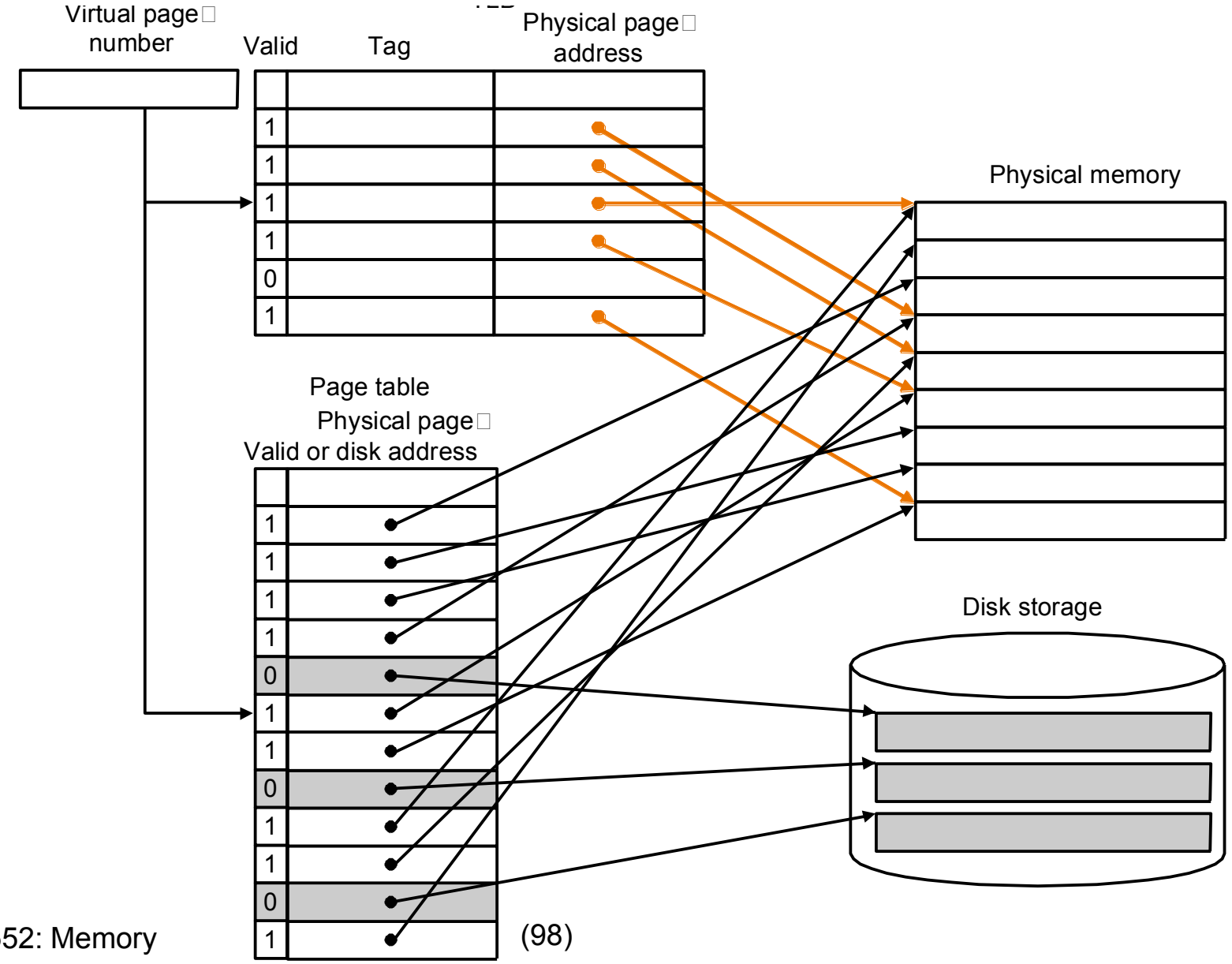
Write handling

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

Faster Translation

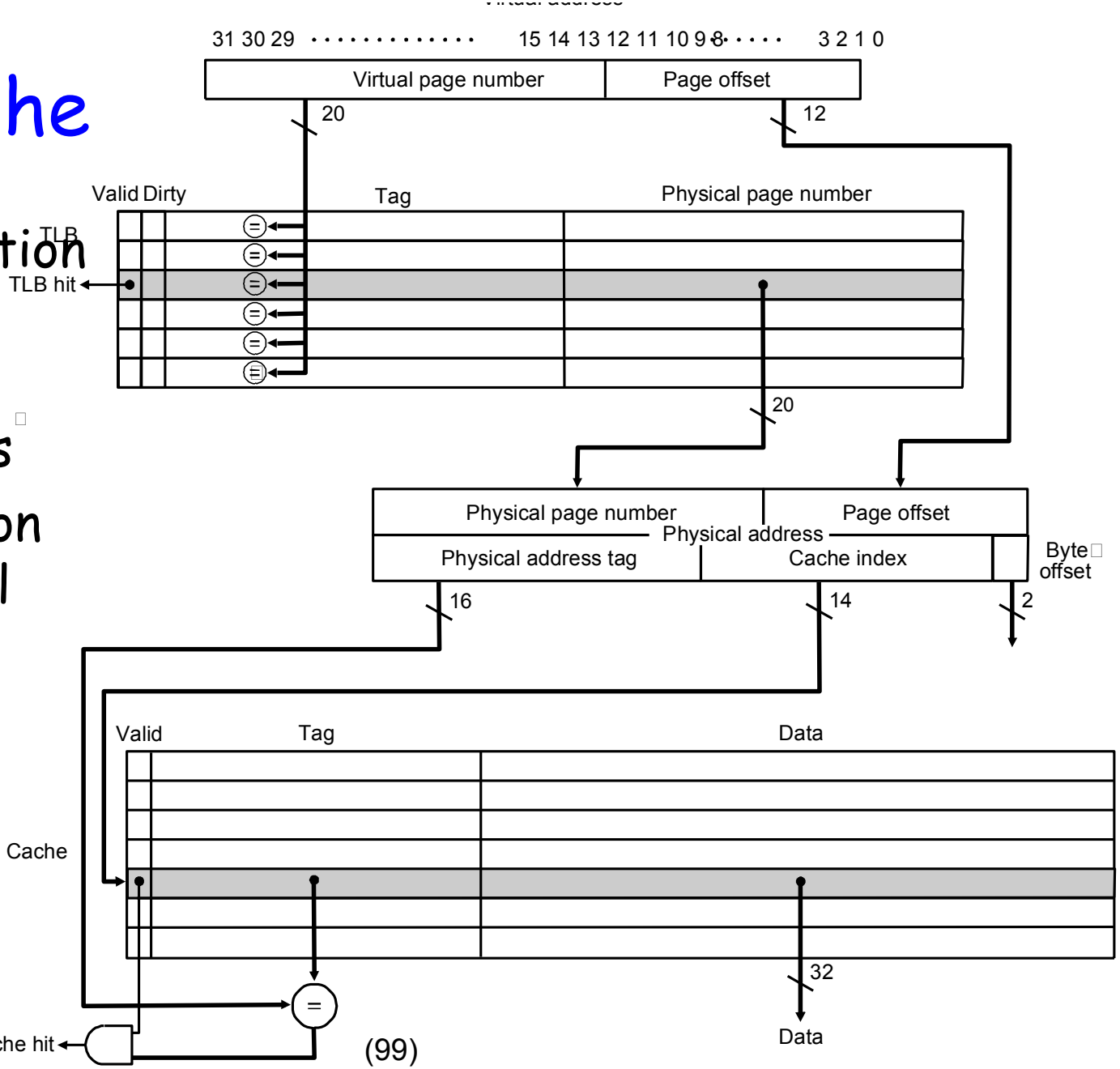
- Recall: 100% overhead with VM system
- Eliminate memory access for translation
  - Caching
  - Translation lookaside buffer (TLB)
    - Also DTB in some literature
    - A cache of translations
  - 64-128 entries
    - Covers 256 KB ~512 KB
  - Organization
    - 64 entry fully associative
    - 256 entry, 16-way set associative

Translation Lookaside Buffer



TLB+Cache

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



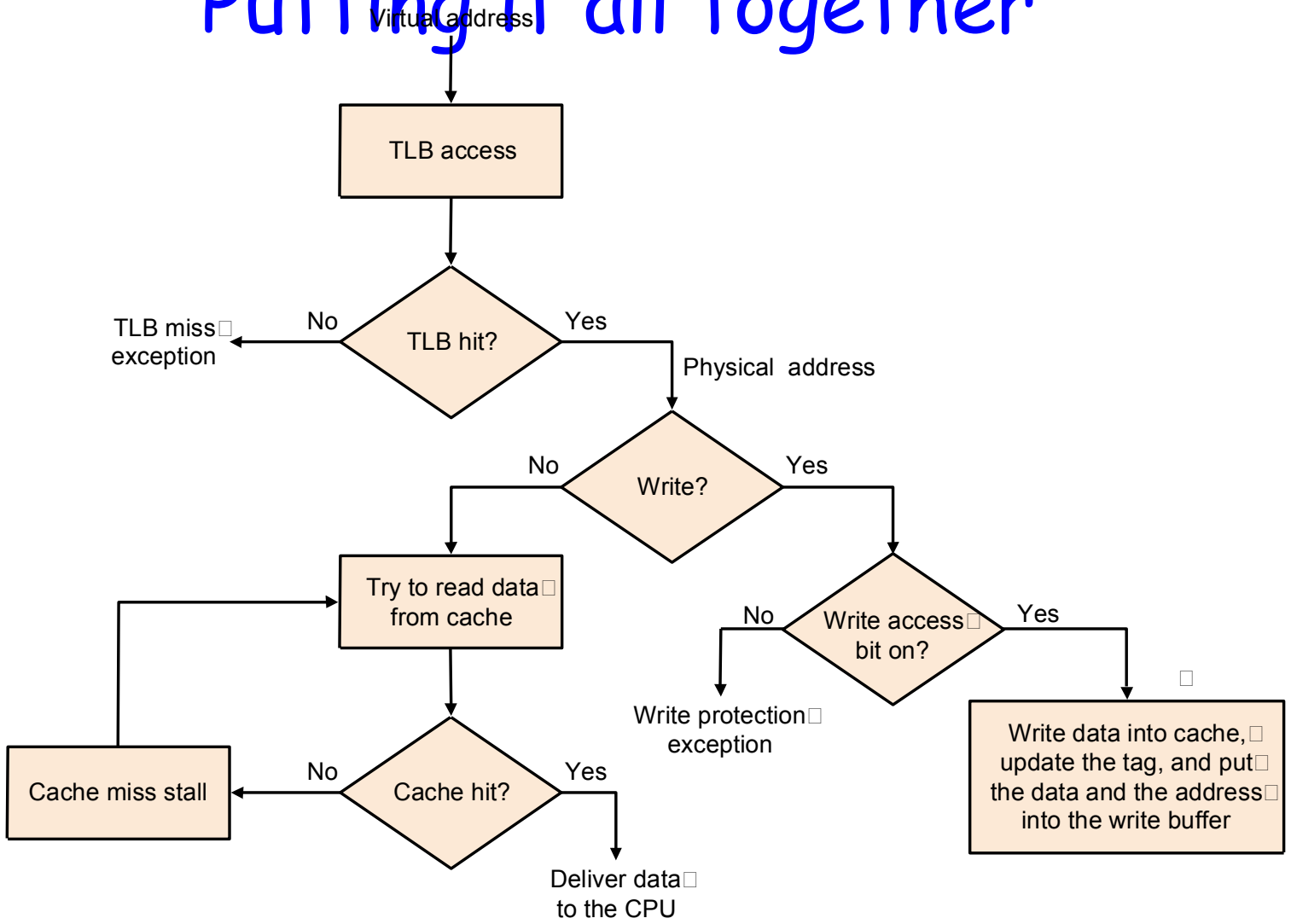
Memory Access Critical path

- Why use physical addresses?
  - Use virtual addresses
  - Faster : no translation
  - Block may have a different (still unique) tag and index
    - Who cares where the block resides in the cache?
- Synonyms
  - Two virtual pages map to same physical page
  - Should not be replicated in cache

Memory Access Critical Path

- Twist in the tale
  - Virtual-index
  - Physical tags
  - Indexing and translation proceeds in parallel
  - Tag comparison after translation

Putting it all together



- Memory access flowchart

Summary

- 4Q on VM
  - Placement : fully associative
  - Identification : Page Table lookup
  - Replacement : LRU / LRU-approx
  - Writes : Writeback
- 4Q on TLB
  - Placement: small and fully associative, larger and set-associative
  - Identification: Associative search (CAM)
  - Replacement : random
  - Writes : ?? Writes to TLB??

VM Miscellanea

- TLB: cache of VA-> translations
- Single TLB is a structural hazard too !
- On a context switch:
  - Change contents of PTBR for appropriate page table
  - What do we do with TLB contents?
    - Flush all entries
      - Simple, but inefficient
    - Associate Process ID with address
      - Flush required only when processor IDs are reused

VM Miscellanea

- Memory efficiency of page tables
  - Limit register
    - Only region between PTBR and Limit is valid
    - Grow as needed
  - Segmented
    - Two page tables and two limit registers (Stack and Heap)
  - Inverted Page table
    - Hashing to map VA to a number within PA range
      - Hash 32-bit VA to 28 bit PA
      - Lookup complications
      - Collision
  - Multilevel page tables
  - Paging page tables

Real Machines

- DEC Alpha 21364 (1.2 GHz)
  - L1: 64K, 2-way, I&D split
    - 3 cycles hit latency
    - 2 memops per cycle (upto 4 insts per cycle)
  - L2: 1.5M, 6-way
    - 12 cycle hit latency
  - System interface, 80 cycle latency
  - Multilevel page tables

Real Stuff

Characteristic	Pentium Pro	PowerPC
VA	32 bit	52 bit
PA	32 bit	32 bit
Page size	4KB, 4MB	4KB, selectable, 256 MB
TLB	Split I&D 4-way assoc Pseudo-random I-32, D-64 TLB miss H/W	Split I&D 2-way assoc LRU I-128, D-128 TLB miss H/W

Real Stuff

Characteristic	Pentium Pro	PowerPC
Cache	Split I & D	Split I & D
Size	8K + 8K	16K + 16K
Associativity	4-way	4-way
Replacement	Approx LRU	LRU
Block	32 bytes	32 bytes
Write	Write-back	Writeback or writethrough