CS 110
Computer Architecture
Lecture 14:
Caches Part I

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Admin

- P 1.2 due today!
  - Can use slip days…
- Project 2.1 will be published very soon.
- Midterm: Hopefully on-site later in the semester
  - Contents: More or less everything up to then

Cache Agenda

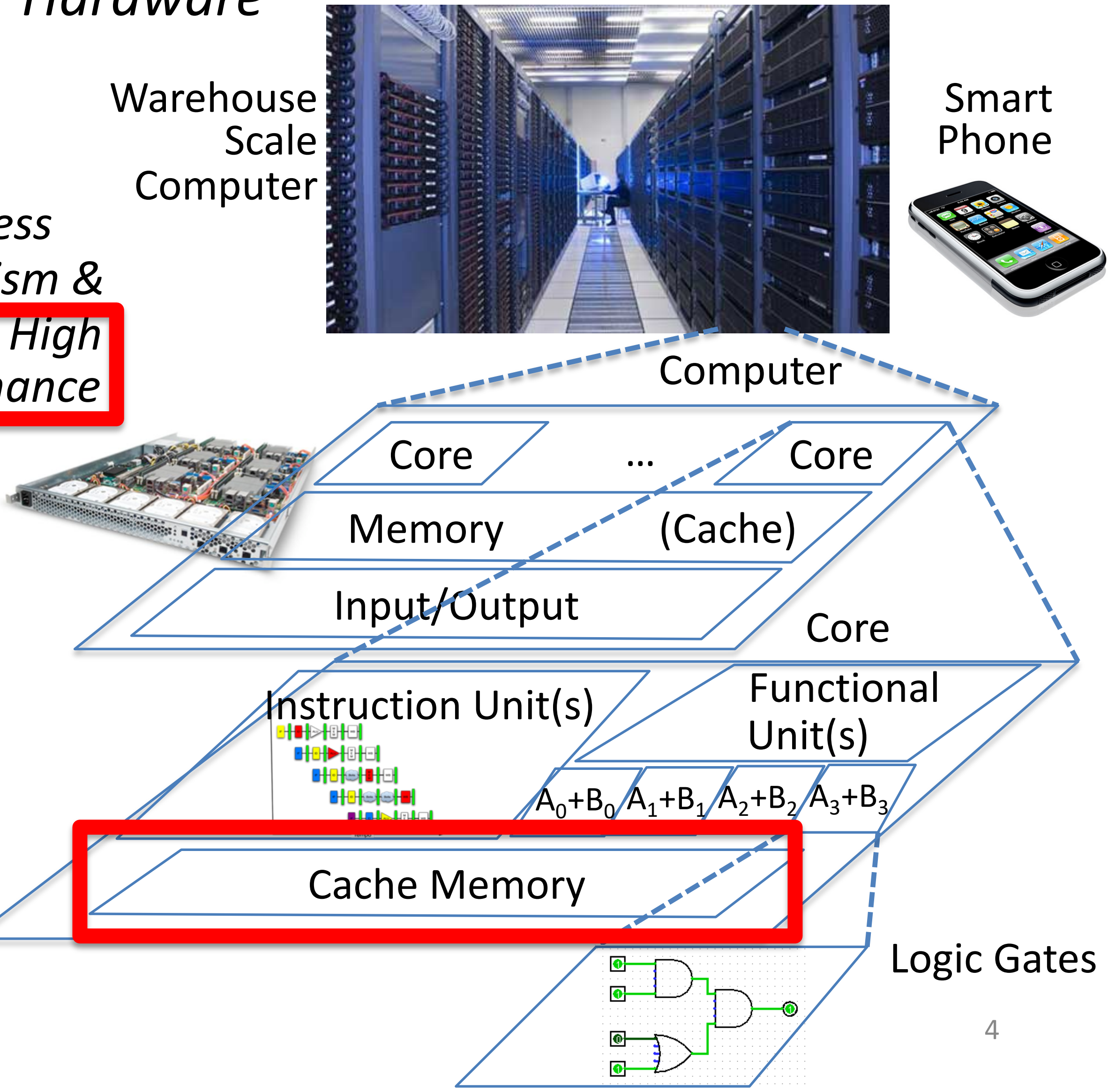
- Cache Lecture I
  - Caches Introduction
  - Principle of Locality
  - Simple Cache
  - Direct Mapped & Set-Associative Caches
- Cache Lecture II
  - Stores to Caches
  - Cache Performance
  - Cache Misses
- Cache Lecture III
  - Multi-Level Caches
  - Cache Configurations
  - Cache Examples
- …
- Lecture: Cache Coherence (Caches for multi-core computers)
- Lecture: Advanced Caches

New-School Machine Structures (It's a bit more complicated!)

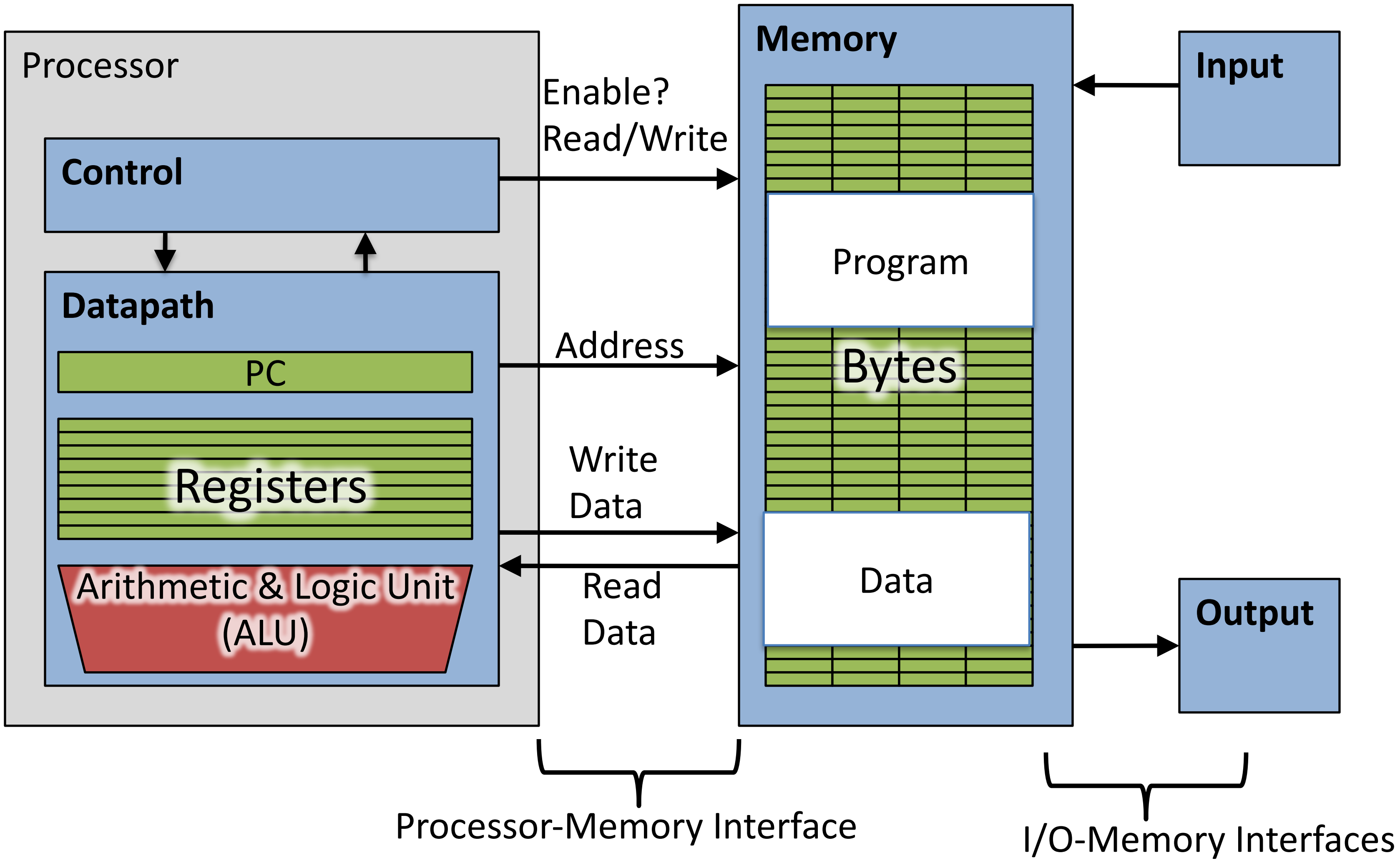
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search "Katz"
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance



Components of a Computer

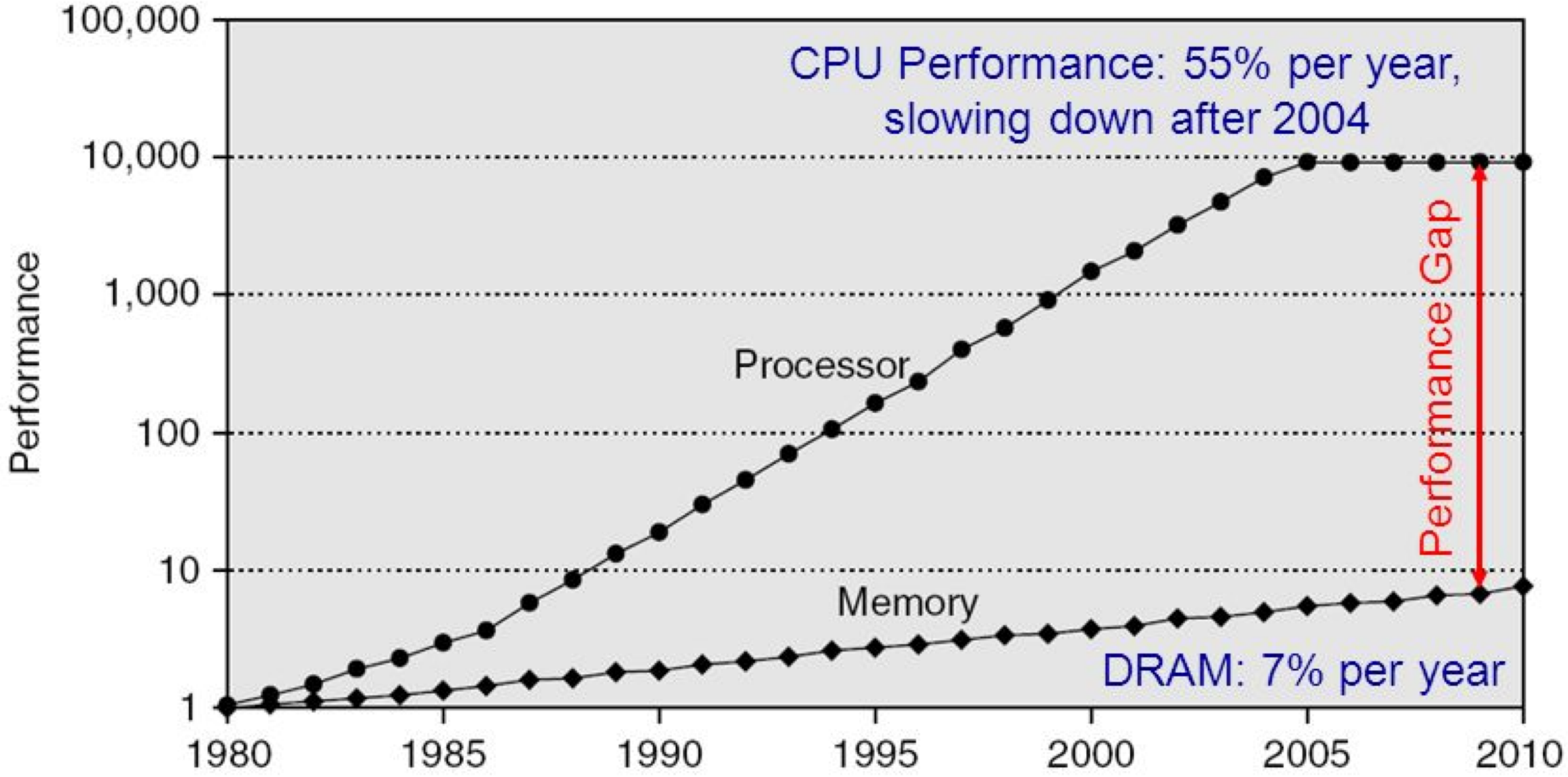


Problem: Large memories slow? Library Analogy

- Finding a book in a large library takes time
  - Takes time to search a large card catalog – (mapping title/author to index number)
  - Round-trip time to walk to the stacks and retrieve the desired book.
- Larger libraries makes both delays worse
- Electronic memories have the same issue, *plus* the technologies that we use to store an individual bit get slower as we increase density (SRAM versus DRAM versus Magnetic Disk)

However what we want is a large yet fast memory!

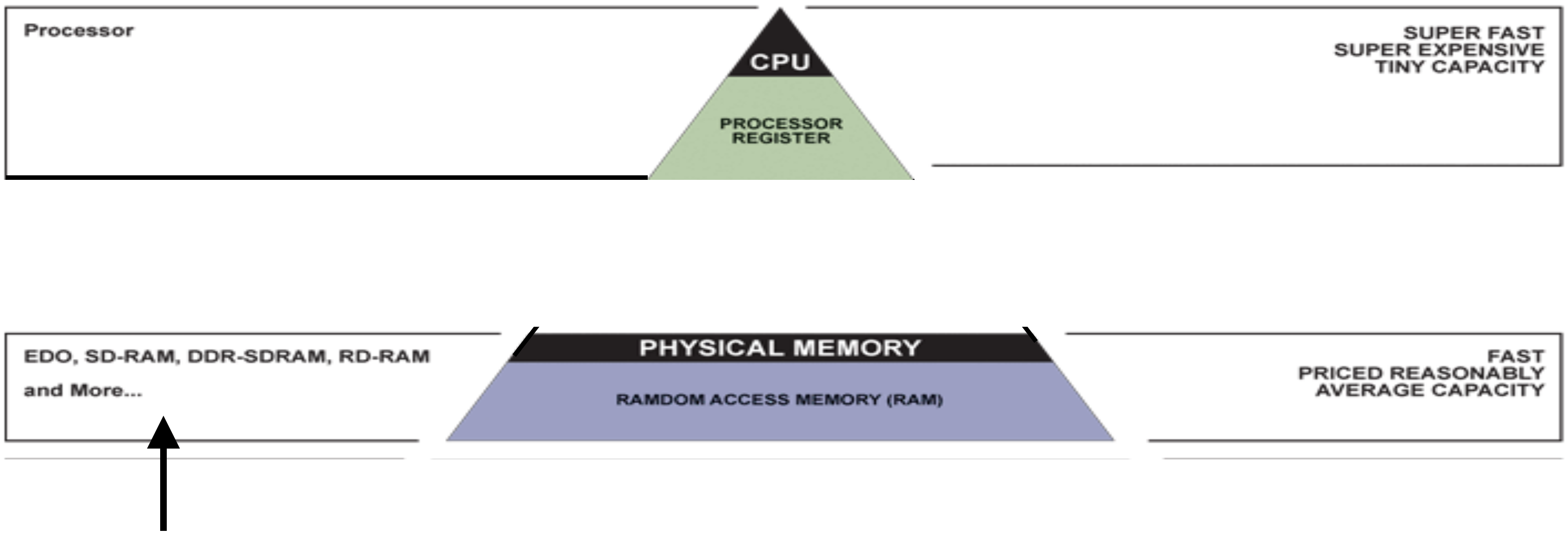
Processor-DRAM Gap (Latency)



1980 microprocessor executes **~one instruction** in same time as DRAM access
2017 microprocessor executes **~1000 instructions** in same time as DRAM access

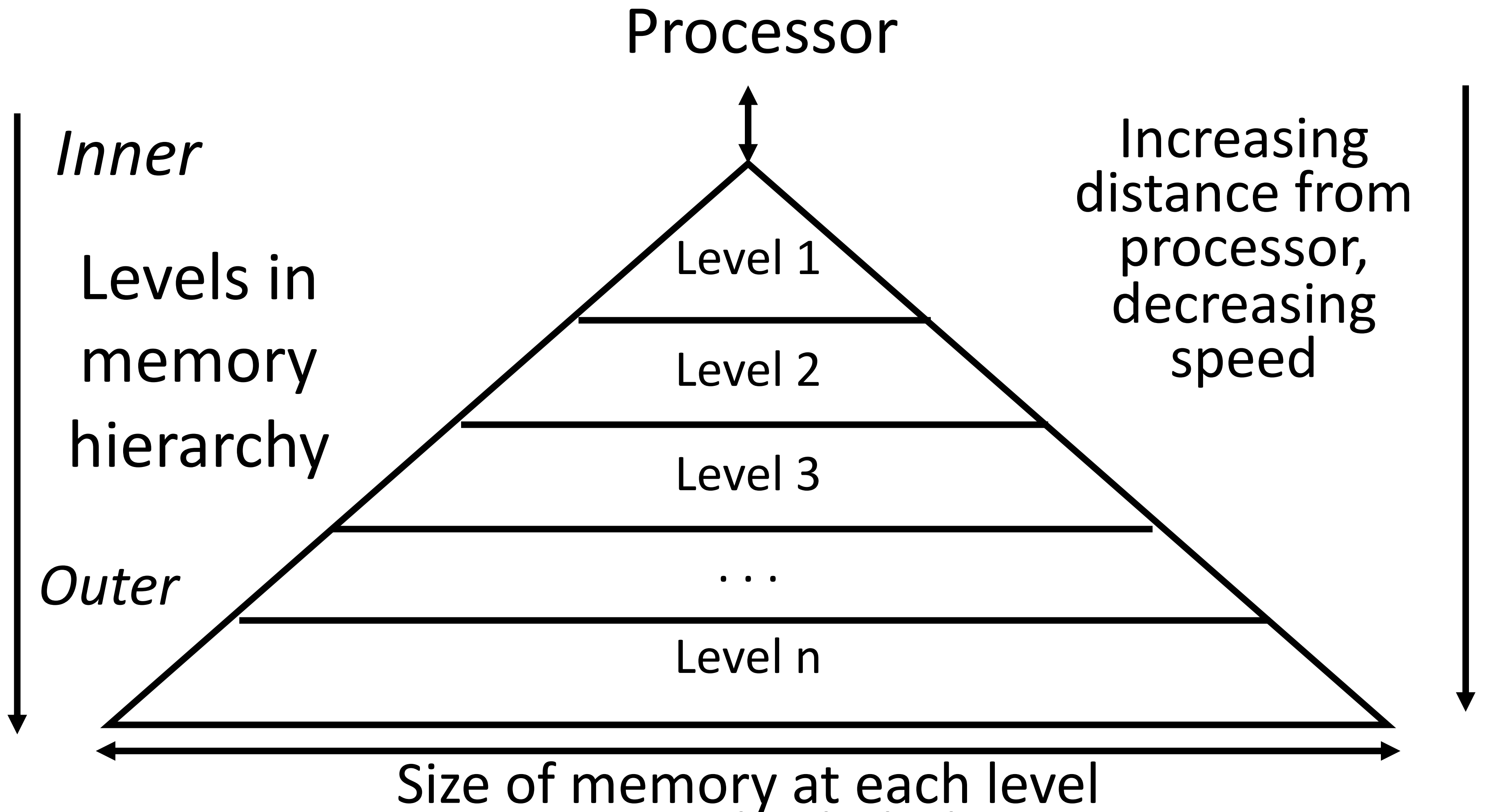
Slow DRAM access has disastrous impact on CPU performance!

Great Idea #3: Principle of Locality / Memory Hierarchy



Note: These names are a bit dated

Big Idea: Memory Hierarchy

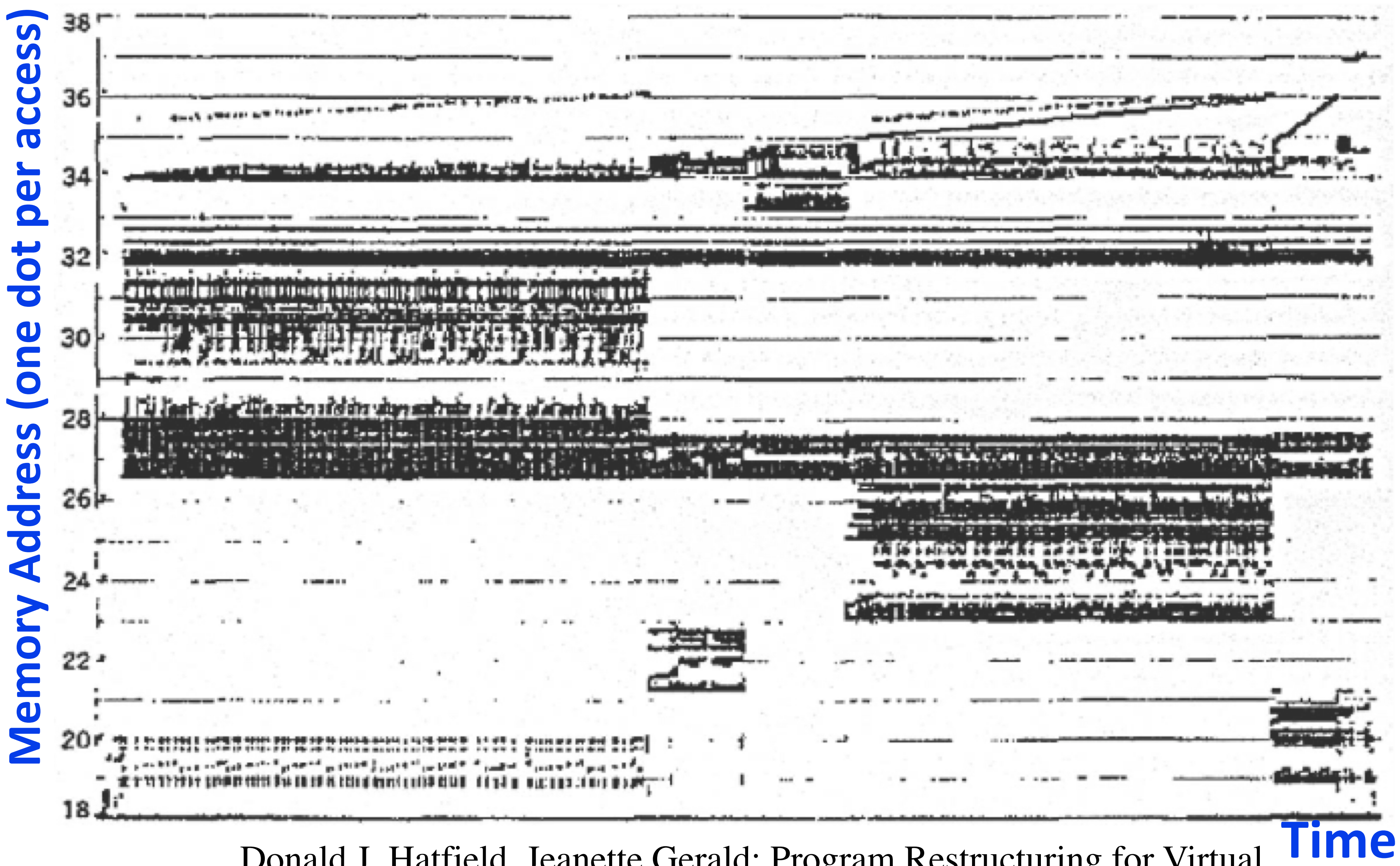

Processor
Inner
Levels in memory hierarchy
Outer
Level 1
Level 2
Level 3
. . .
Level n
Increasing distance from processor, decreasing speed
Size of memory at each level

As we move to outer levels the latency goes up and price per bit goes down.

What to do: Library Analogy

- Want to write a report using library books
- Go to library, look up relevant books, fetch from stacks, and place on desk in library
- If need more, check them out and keep on desk
  - But don’t return earlier books since might need them
- You hope this collection of ~10 books on desk enough to write report, despite 10 being only a tiny fraction of books available

Real Memory Reference Patterns



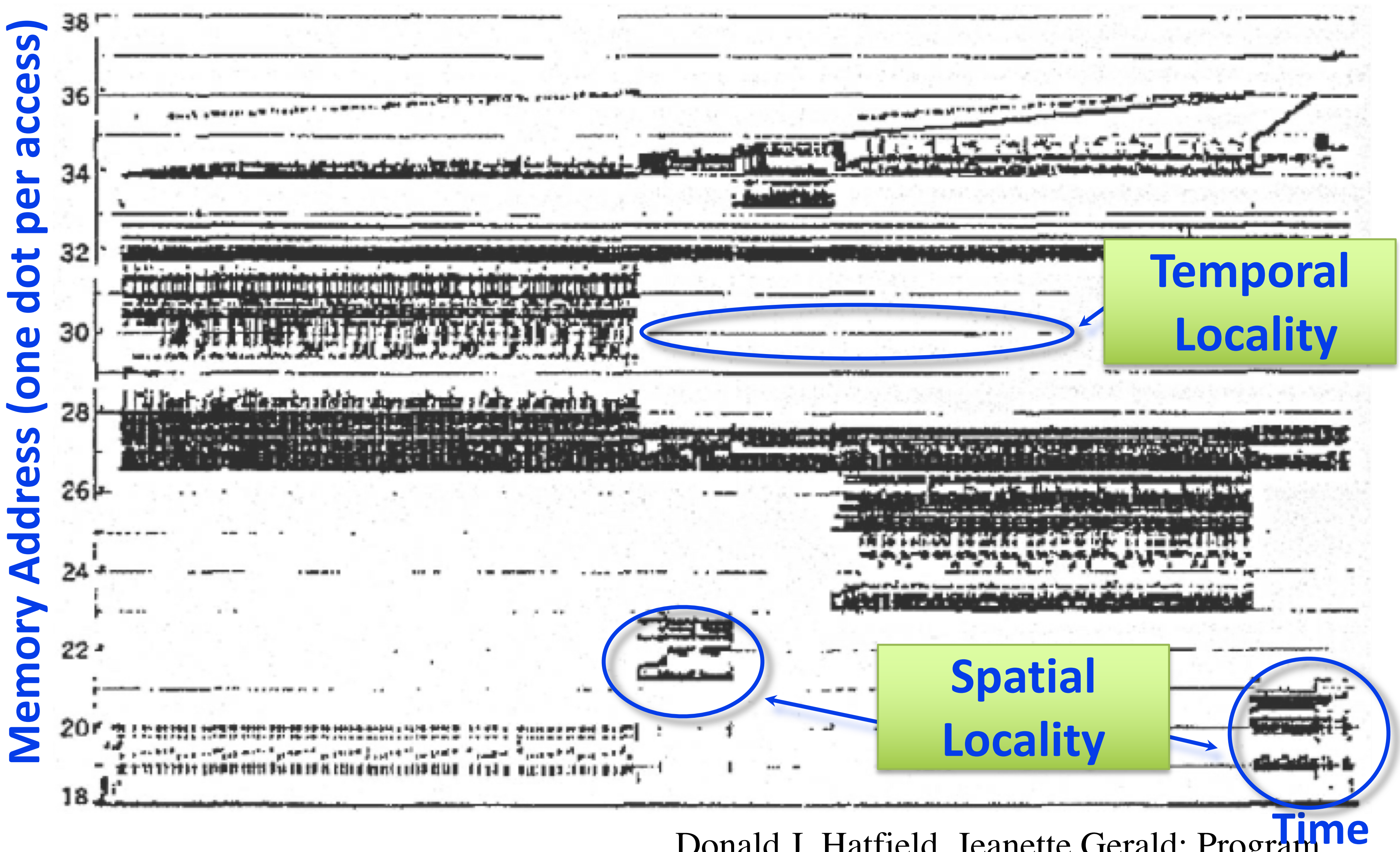
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Big Idea: Locality

- *Temporal Locality* (locality in time)
  - If a memory location is referenced, then it will tend to be referenced again soon

- *Spatial Locality* (locality in space)
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Memory Reference Patterns

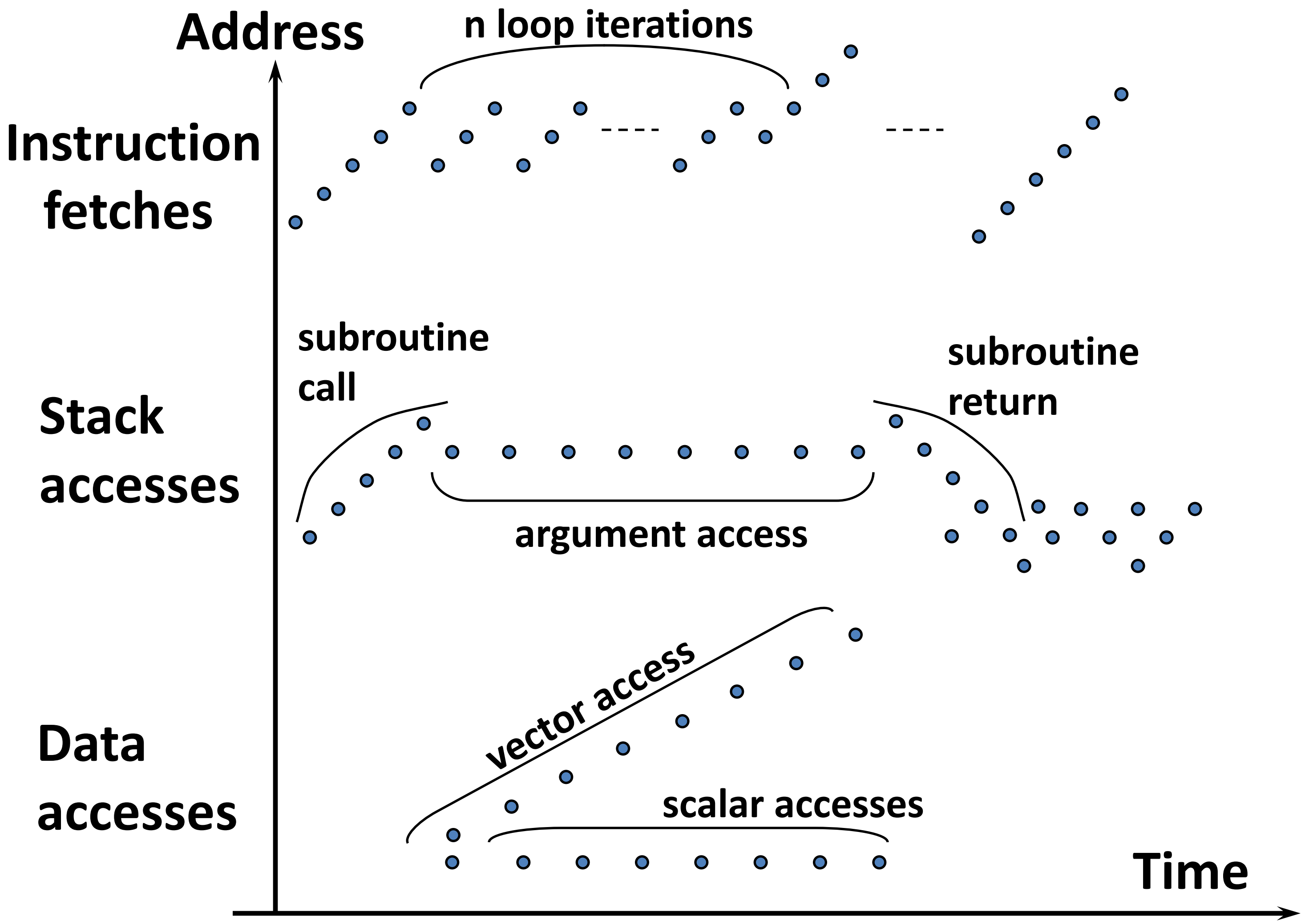


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- What program structures lead to temporal and spatial locality in **instruction** accesses?
- In **data** accesses?

Memory Reference Patterns



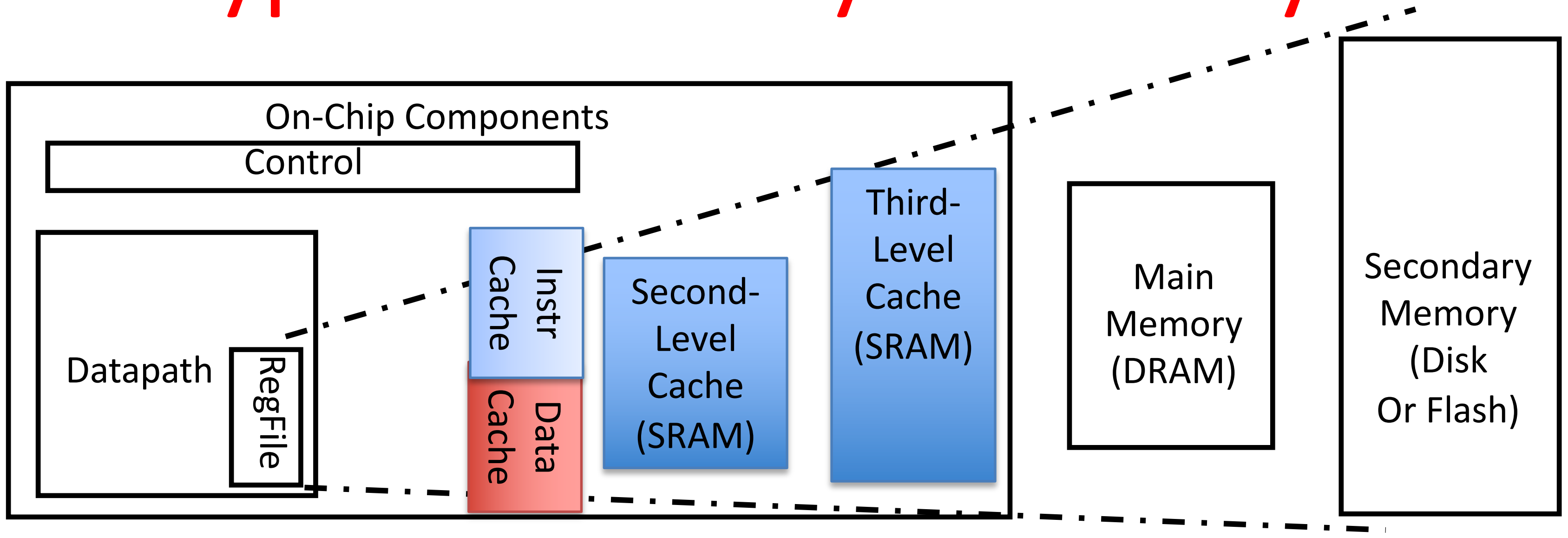
And the Bane of Locality: Pointer Chasing...

- We all love linked lists, trees, etc...
  - Easy to append onto and manipulate...
- But they have ***horrid*** locality preferences
  - Every time you follow a pointer it is to an unrelated location: No spacial reuse from previous pointers
  - And if you don't chase the pointers again you don't get temporal reuse either
- Why modern languages tend to do things a bit differently. For example, ***go*** has "slices" and "maps":
  - Slice, easy to append to to array
    - Only copies on append when you overwhelm the size
  - Map, a hash table implementation
    - But without nearly so much pointer chasing

Cache Philosophy

- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
  - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache
  - And modern programming languages try to provide storage abstractions that provide flexibility while still caching well
- Does have limits: When you overwhelm the cache your performance may drop off a cliff...

Typical Memory Hierarchy



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest ⟷				lowest

- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

How is the Hierarchy Managed?

- registers ↔ memory
  - By compiler (or assembly level programmer)
- cache ↔ main memory
  - By the cache controller hardware
- main memory ↔ disks (secondary storage)
  - By the operating system (virtual memory)
  - Virtual to physical address mapping assisted by the hardware ('translation lookaside buffer' or TLB)
  - By the programmer (files)

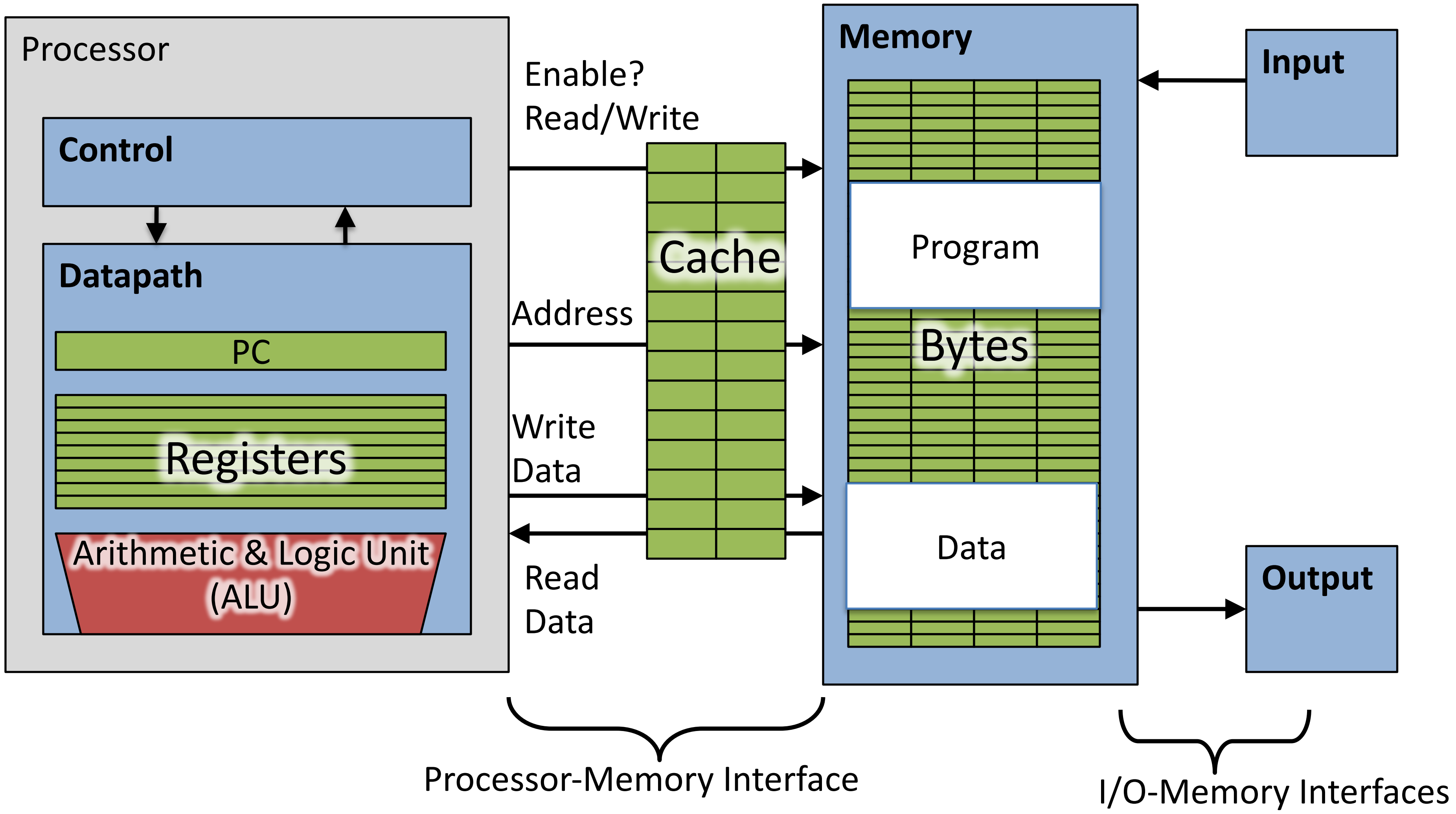
↑ **Also a type of cache**

Memory Access without Cache

- Load word instruction: `lw t0,0(t1)`
- t1 contains $1022_{ten,}$ Memory[1022] = 99

1. Processor issues address 1022_{ten} to Memory
2. Memory reads word at address 1022_{ten} (99)
3. Memory sends 99 to Processor
4. Processor loads 99 into register t0

Adding Cache to Computer



Memory Access with Cache

- Load word instruction: `lw t0,0(t1)`
- t1 contains 1022_{ten}, Memory[1022] = 99
- With cache: Processor issues address 1022_{ten} to Cache
  1. Cache checks to see if has copy of data at address 1022_{ten}

     2a. If finds a match (Hit): cache reads 99, sends to processor

     2b. No match (Miss): cache sends address 1022 to Memory

     I. Memory reads 99 at address 1022_{ten}

     II. Memory sends 99 to Cache

     III. Cache replaces word with new 99

     IV. Cache sends 99 to processor
  2. Processor loads 99 into register t0

TA Discussion

Mengying Wu

Q & A

Quiz

Quiz

Piazza: "Online Lecture 14 $ Poll"

- Select statements that are true:

A. The assembly programmer/ compiler has to use the cache correctly to ensure the correct execution of the program.
B. The assembly programmer/ compiler has to use the main memory correctly to ensure the correct execution of the program.
C. Random accesses to memory will benefit very little from caches.
D. We use a hierarchy of caches to give the programmer the illusion of having memory as big as the biggest memory with almost the speed of the fastest memory.

CS 110
Computer Architecture
Lecture 14:
Caches Part I
Video 2: Cache Details

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Memory Access with Cache

- Load word instruction: `lw t0,0(t1)`
- t1 contains $1022_{ten,}$ Memory[1022] = 99
- With cache: Processor issues address 1022_{ten} to Cache

1. Cache checks to see if has copy of data at address 1022_{ten}

   2a. If finds a match (Hit): cache reads 99, sends to processor

   2b. No match (Miss): cache sends address 1022 to Memory

      I. Memory reads 99 at address 1022_{ten}

      II. Memory sends 99 to Cache

      III. Cache replaces word with new 99

      IV. Cache sends 99 to processor

2. Processor loads 99 into register t0

Cache "Tags"

- Need way to tell if have copy of location in memory so that can decide on hit or miss
- On cache miss, put memory address of block in "tag address" of cache block

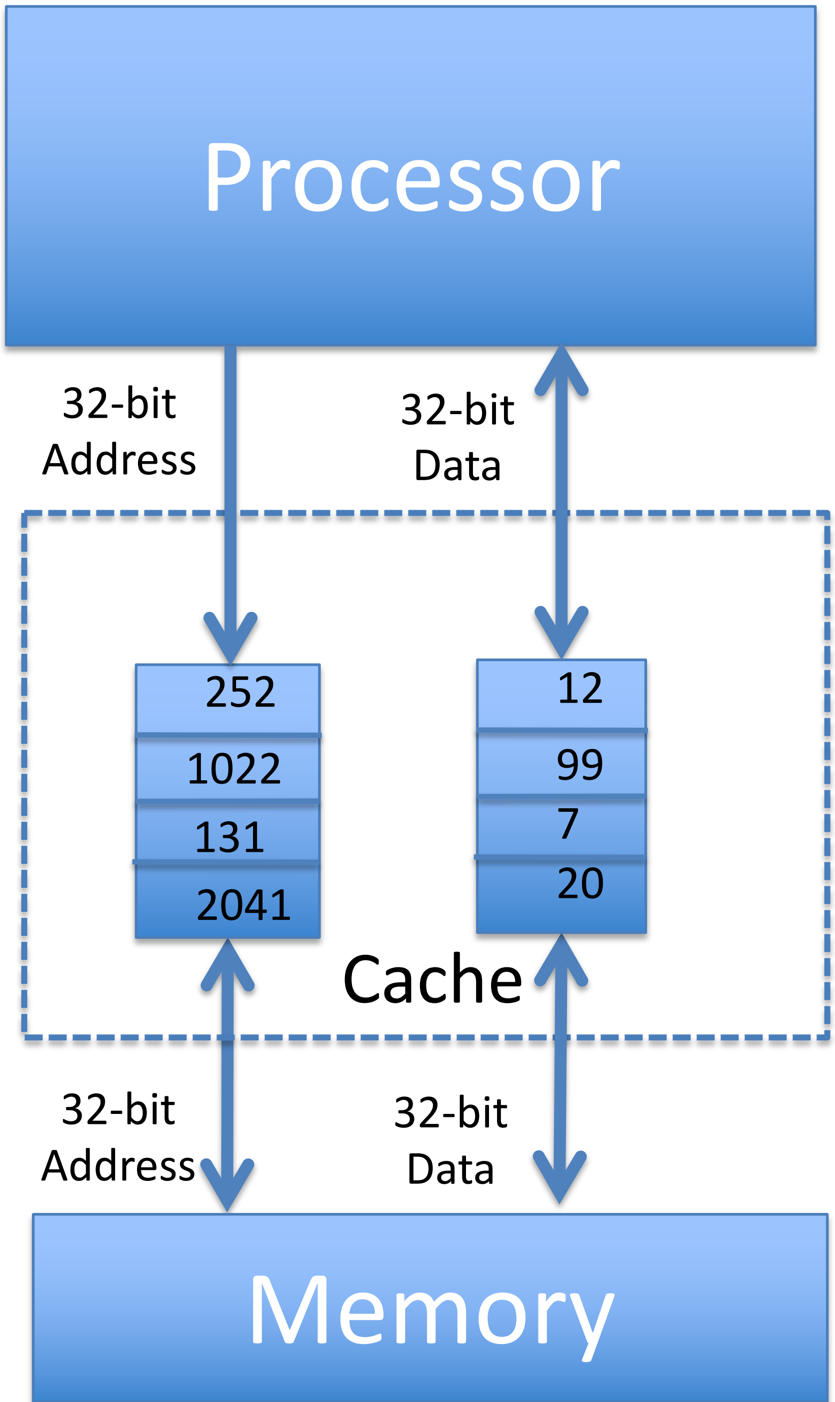
  1022 placed in tag next to data from memory (99)

Tag (= Address in this simple example)	Data
252	12
1022	99
131	7
2041	20

From earlier instructions

Anatomy of a 16 Byte Cache, 4 Byte Block

- Operations:
  1. Cache Hit
  2. Cache Miss
  3. Refill cache from memory
- Cache needs Address Tags to decide if Processor Address is a Cache Hit or Cache Miss
  - Compares all 4 tags



Cache Replacement

- Suppose processor now requests location 511, which contains 11?
- Doesn't match any cache block, so must "evict" one resident block to make room
  - Which block to evict?
- Replace "victim" with new memory block at address 511

Tag	Data
252	12
1022	99
511	11
2041	20

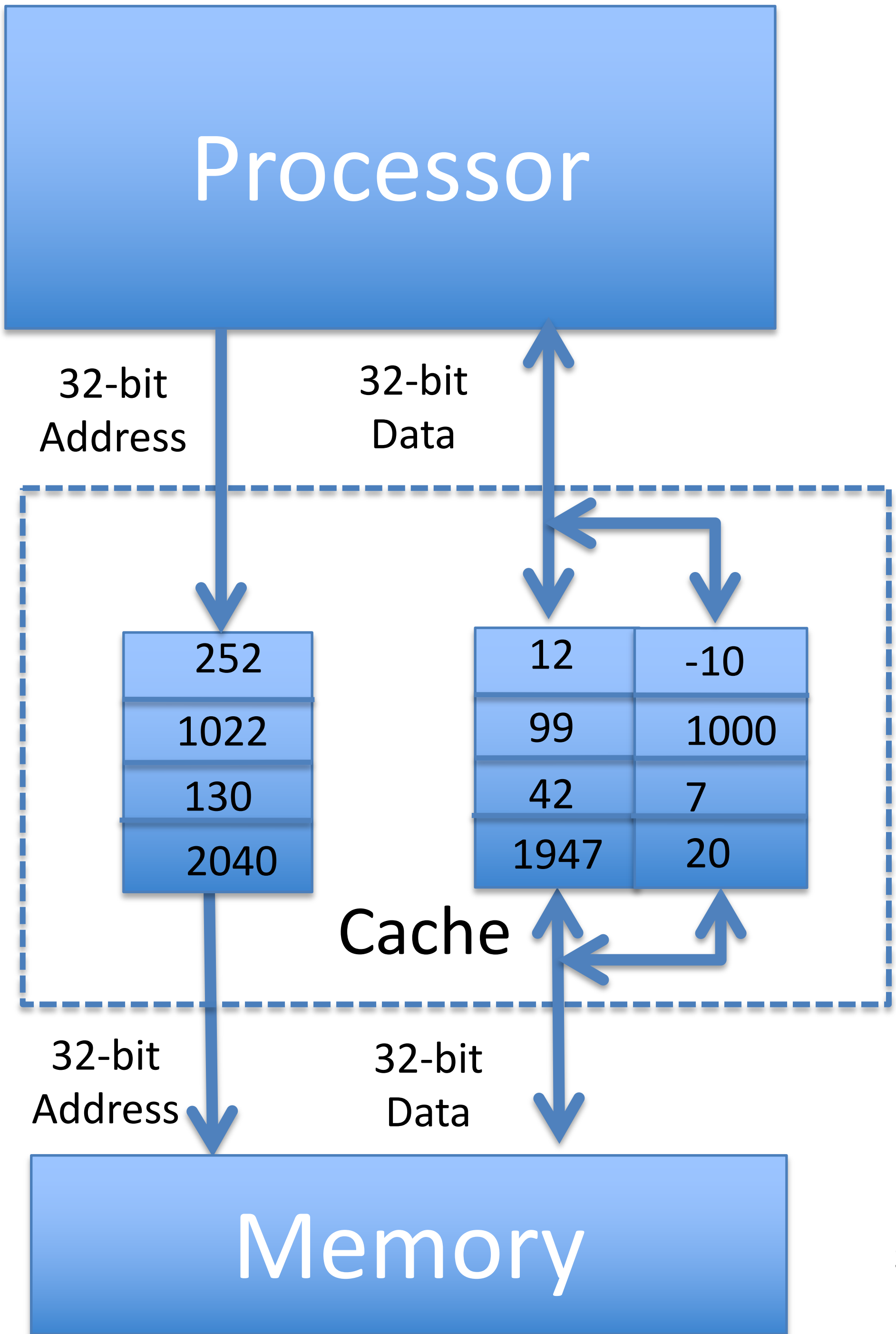
Block Must be Aligned in Memory

- Word blocks are aligned, so binary address of all words in cache always ends in 00_{two}
- How to take advantage of this to save hardware and energy?
- Don't need to compare last 2 bits of 32-bit byte address (comparator can be narrower)

=> Don't need to store last 2 bits of 32-bit byte address in Cache Tag (Tag can be narrower)

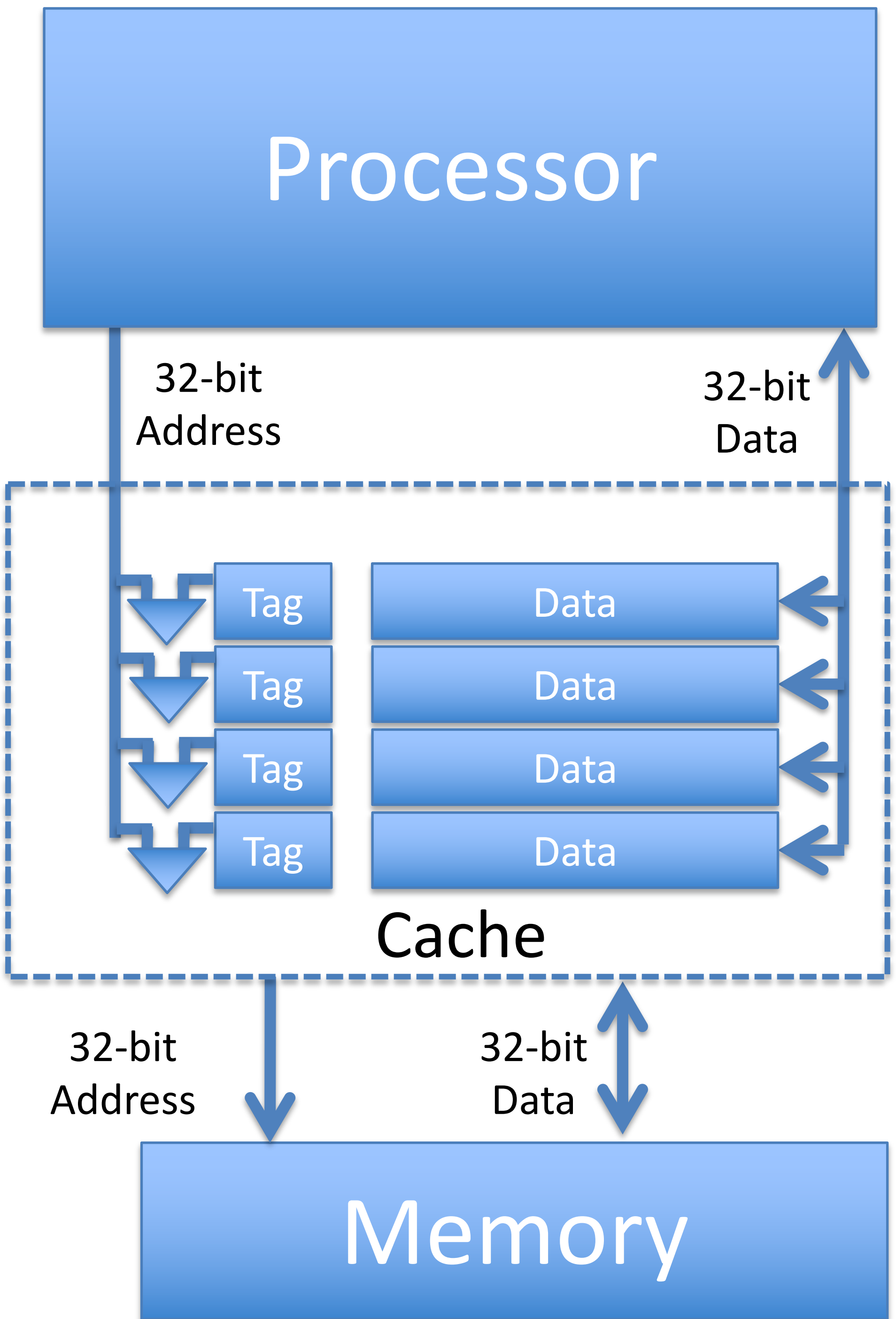
Anatomy of a 32B Cache, 8B Block

- Blocks must be aligned in pairs, otherwise could get same word twice in cache
  - ➢ Tags only have even-numbered words
  - ➢ Last 3 bits of address always 000_{two}
  - ➢ Tags, comparators can be narrower
- Can get hit for either word in block



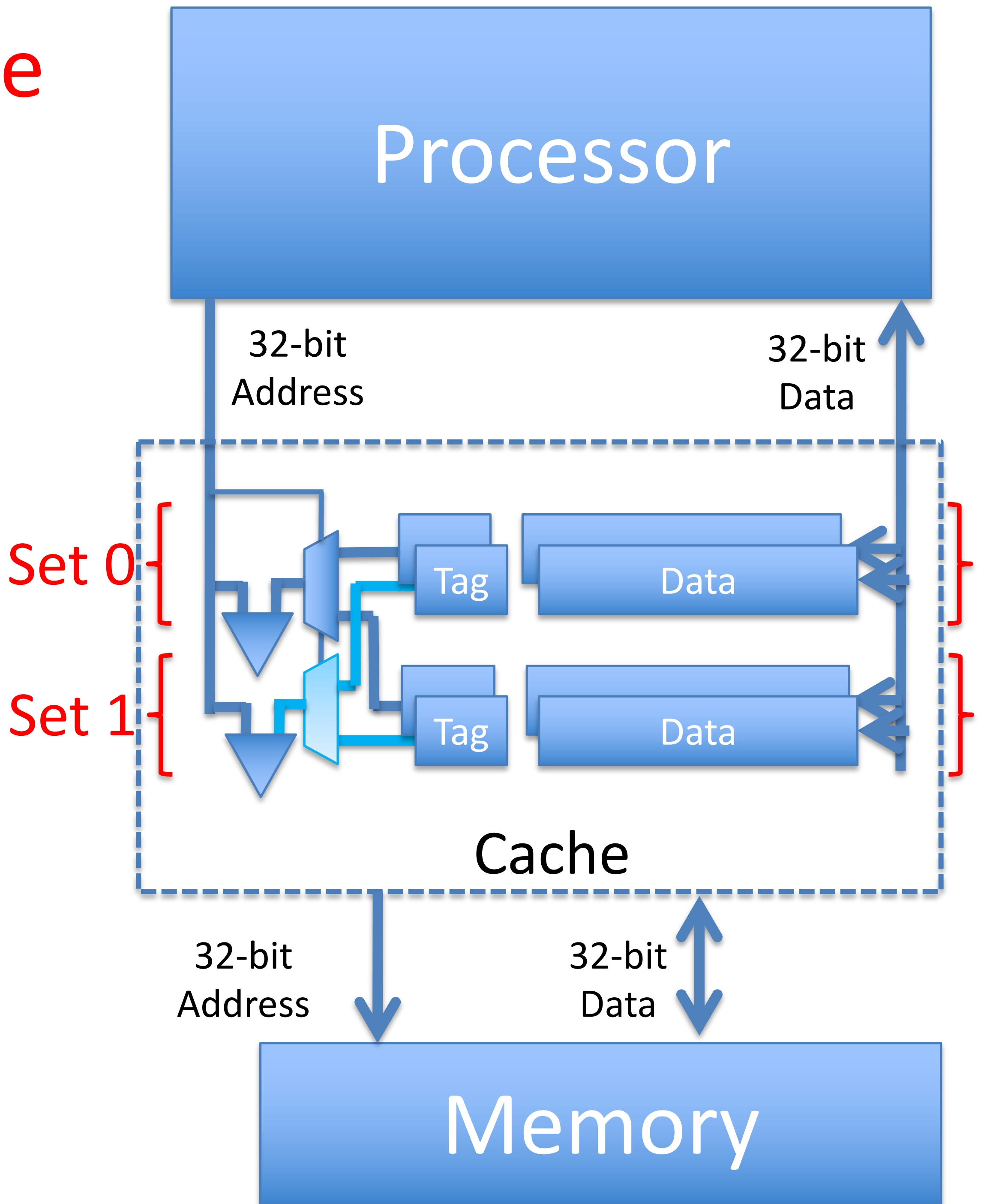
Hardware Cost of Cache

- Need to compare every tag to the Processor address
- Comparators are expensive



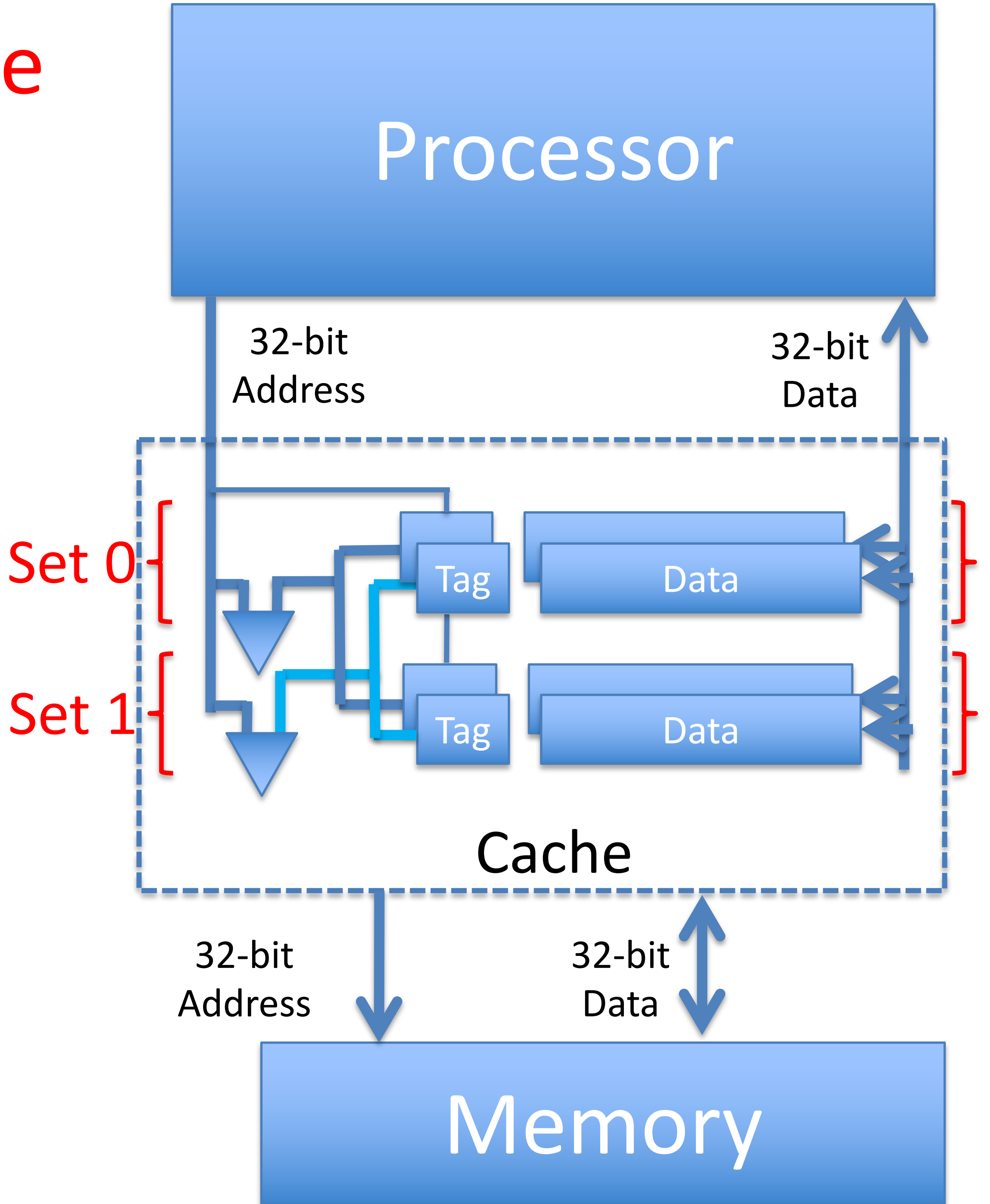
Set Associative Cache

- Optimization: use 2 “sets” => ½ comparators
- 1 Address bit selects which set
- Compare only tags from selected set
- Generalize to more sets:
  - Need as many comparitors as tags in a set



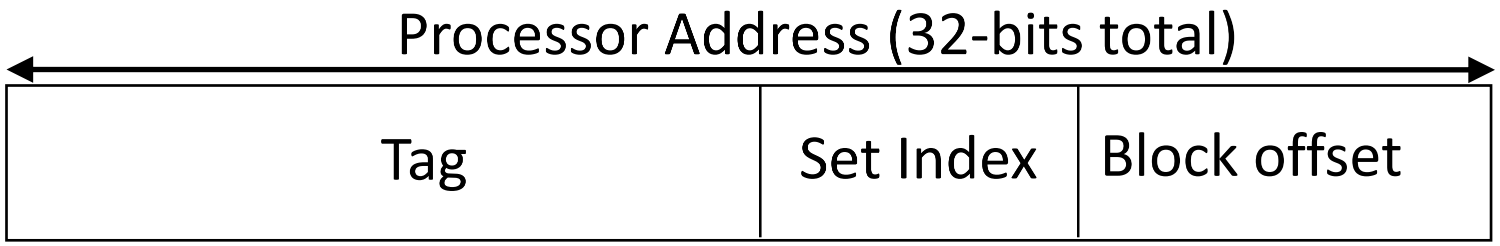
Set Associative Cache

- Optimization: use 2 “sets” => ½ comparators
- 1 Address bit selects which set
- Compare only tags from selected set
- Generalize to more sets:
  - Need as many comparitors as tags in a set
  - Don’t need extra mux per comparitor – tags and data are memory – have mux inside!



Processor Address Fields used by Cache Controller

- Block Offset: Byte address within block
- Set Index: Selects which set
- Tag: Remaining portion of processor address



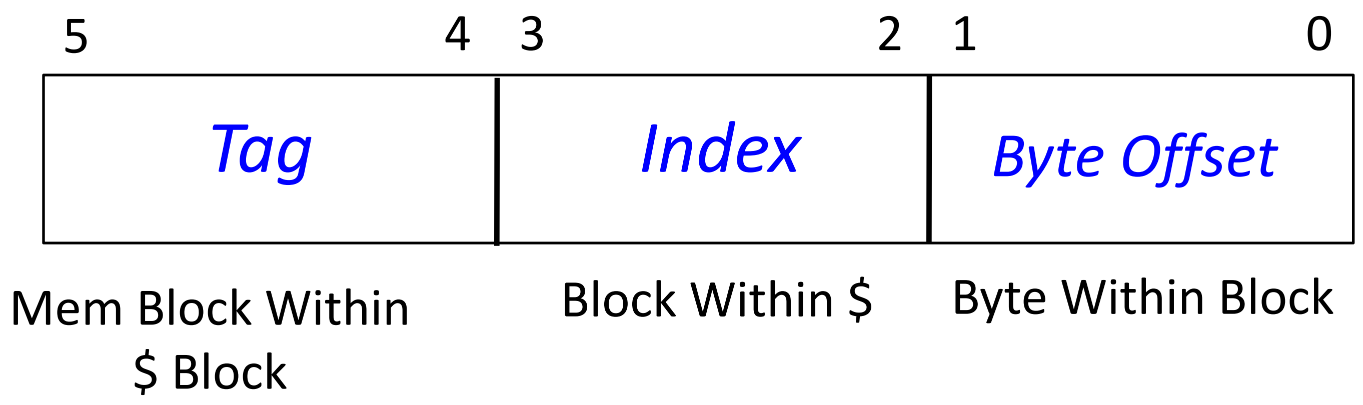
- Size of Index = log2 (number of sets)
- Size of Tag = Address size – Size of Index – log2 (number of bytes/block)

What is limit to number of sets?

- For a given total number of blocks, we can save more comparators if have more than 2 sets
- Limit: As Many Sets as Cache Blocks => only one block per set – only needs one comparator!
- Called “Direct-Mapped” Design

Tag	Index	Block offset

Direct Mapped Cache Ex: Mapping a 6-bit Memory Address



- In example, block size is 4 bytes/1 word
- Memory and cache blocks always the same size, unit of transfer between memory and cache
- # Memory blocks >> # Cache blocks
  - 16 Memory blocks = 16 words = 64 bytes => 6 bits to address all bytes
  - 4 Cache blocks, 4 bytes (1 word) per block
  - 4 Memory blocks map to each cache block
- Memory block to cache block, aka *index*: middle two bits
- Which memory block is in a given cache block, aka *tag*: top two bits

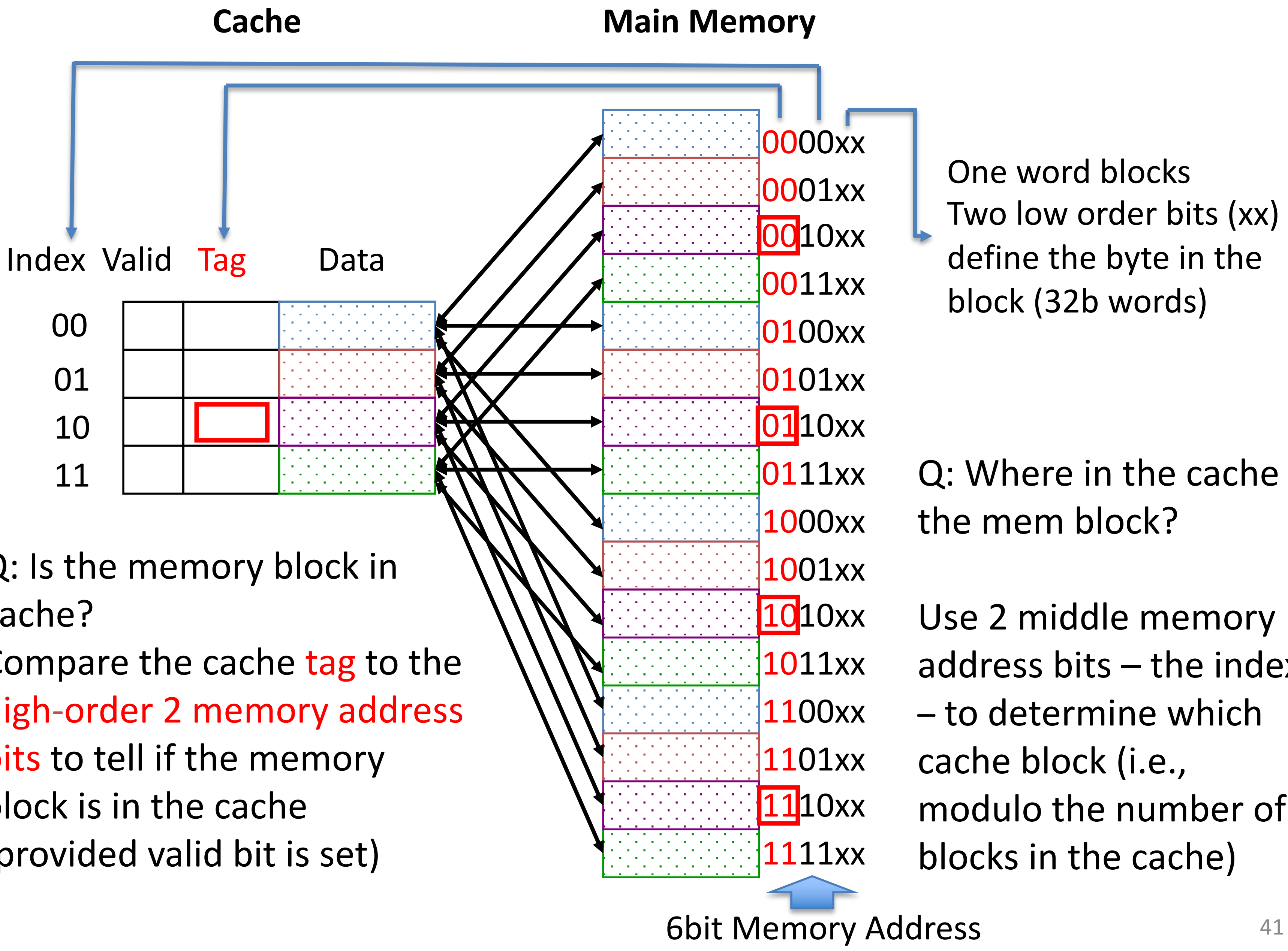
One More Detail: Valid Bit

- When start a new program, cache does not have valid information for this program
- Need an indicator whether this tag entry is valid for this program
- Add a “valid bit” to the cache tag entry

  0 => cache miss, even if by chance, address = tag

  1 => cache hit, if processor address = tag

Caching: A Simple First Example



One word blocks
Two low order bits (xx) define the byte in the block (32b words)

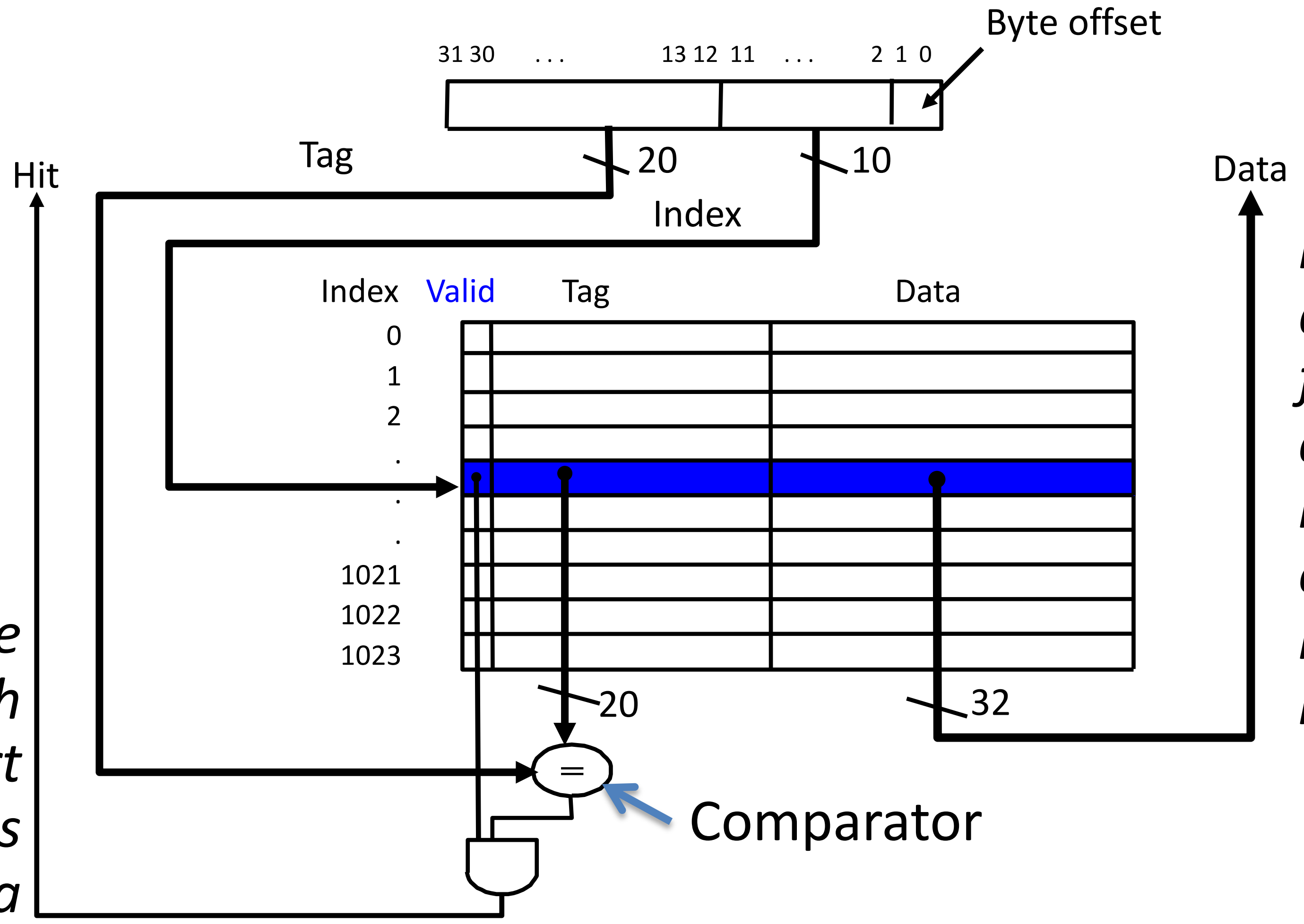
Q: Where in the cache is the mem block?

Use 2 middle memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

Direct-Mapped Cache Example

- One word blocks, cache size = 1K words (or 4KB)



Valid bit ensures something useful in cache for this index

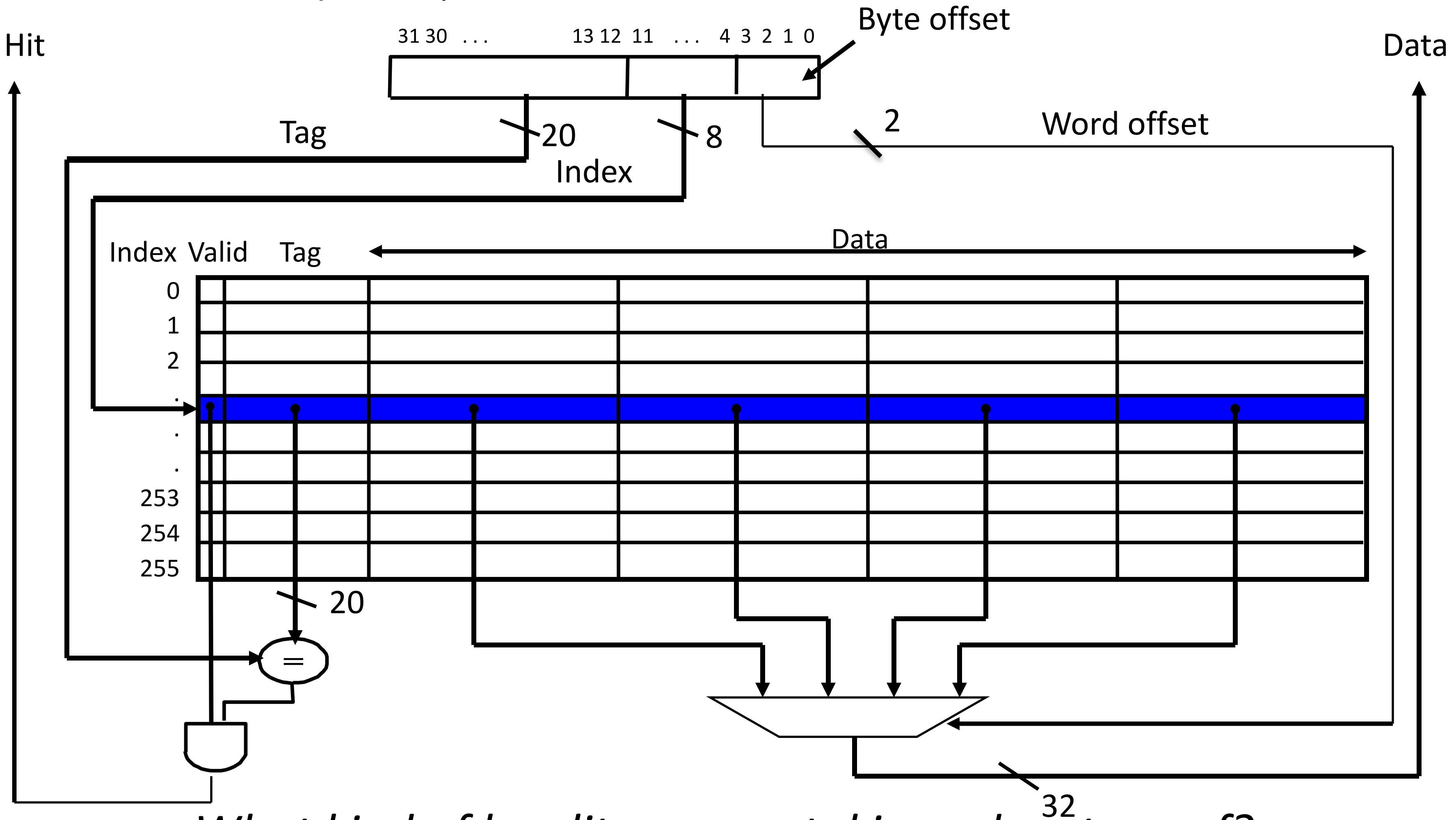
Compare Tag with upper part of Address to see if a Hit

Read data from cache instead of memory if a Hit

What kind of locality are we taking advantage of?

Multiword-Block Direct-Mapped Cache

- Four words/block, cache size = 1K words



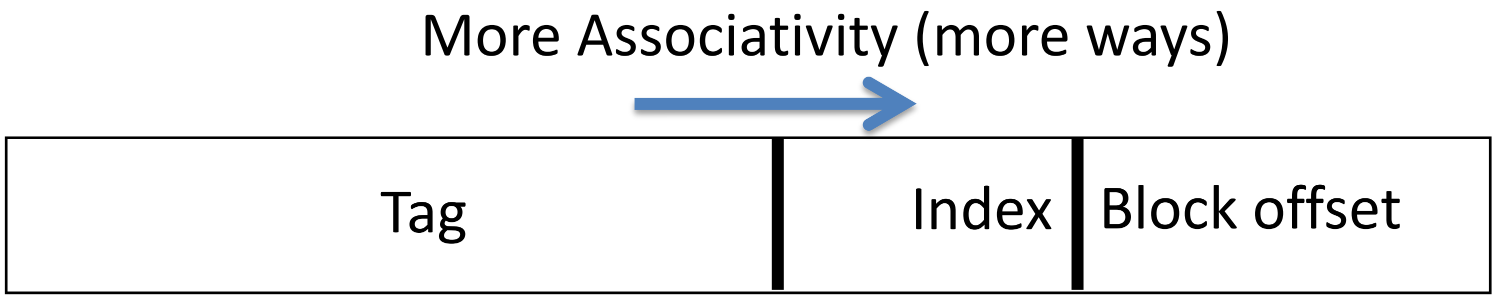
What kind of locality are we taking advantage of?

Cache Names for Each Organization

- "Fully Associative": Line can go anywhere
  - First design in lecture
  - Note: No Index field, but 1 comparator/ line
- "Direct Mapped": Line goes one place
  - Note: Only 1 comparator
  - Number of sets = number blocks
- "N-way Set Associative": N places for a line
  - Number of sets = number of lines/ N
  - N comparators
  - ***Fully Associative: N = number of lines***
  - ***Direct Mapped: N = 1***

Range of Set-Associative Caches

- For a fixed-size cache, and a given block size, each increase by a factor of 2 in associativity doubles the number of blocks per set (i.e., the number of "ways") and halves the number of sets –
  - decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Total Cash Capacity =

Associativity * # of sets * block_size

$Bytes = blocks/set * sets * Bytes/block$

$C = N * S * B$

Tag	*Index*	*Byte Offset*

address_size = tag_size + index_size + offset_size
= tag_size + log2(S) + log2(B)

And In Conclusion, …

- Principle of Locality for Libraries /Computer Memory
- Hierarchy of Memories (speed/size/cost per bit) to Exploit Locality
- Cache – copy of data lower level in memory hierarchy
- Direct Mapped to find block in cache using Tag field and Valid bit for Hit

Quiz

Piazza: “Video Lecture 14 $ Poll”

- For a cache with constant total capacity, if we increase the number of ways by a factor of 2, what are the FALSE statement(s)?:

A: The number of sets could be doubled

B: The tag width could decrease

C: The block size could stay the same

D: The block size could be halved

E: Tag width must increase