CS 61C: Great Ideas in Computer Architecture

Lecture 14: *Caches, Part I*

Instructor: Sagar Karandikar

sagark@eecs.berkeley.edu

http://inst.eecs.berkeley.edu/~cs61c

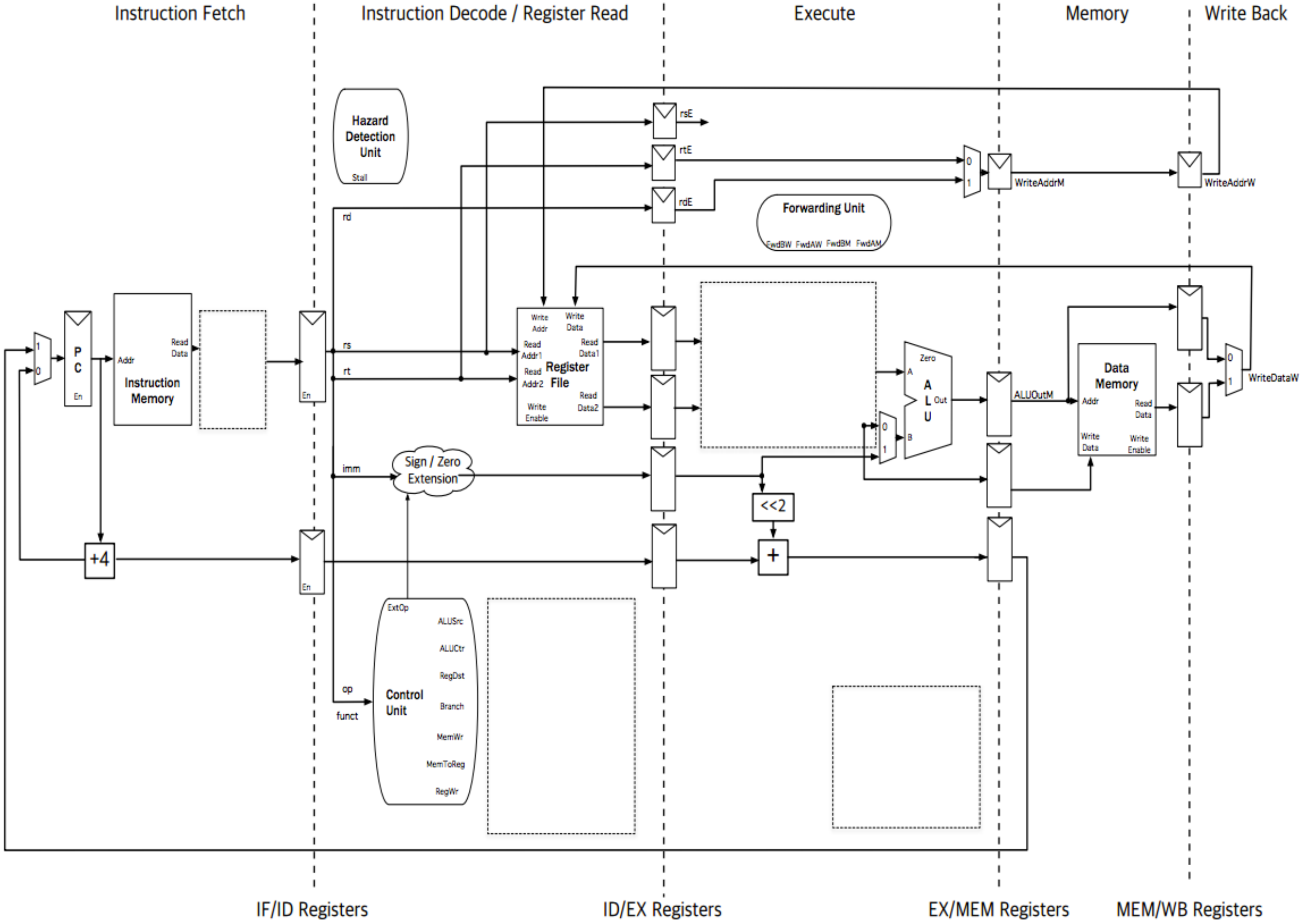
Review: Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock period is?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What can we do to improve clock rate?
- Will this improve performance as well?

Want increased clock rate to mean faster programs

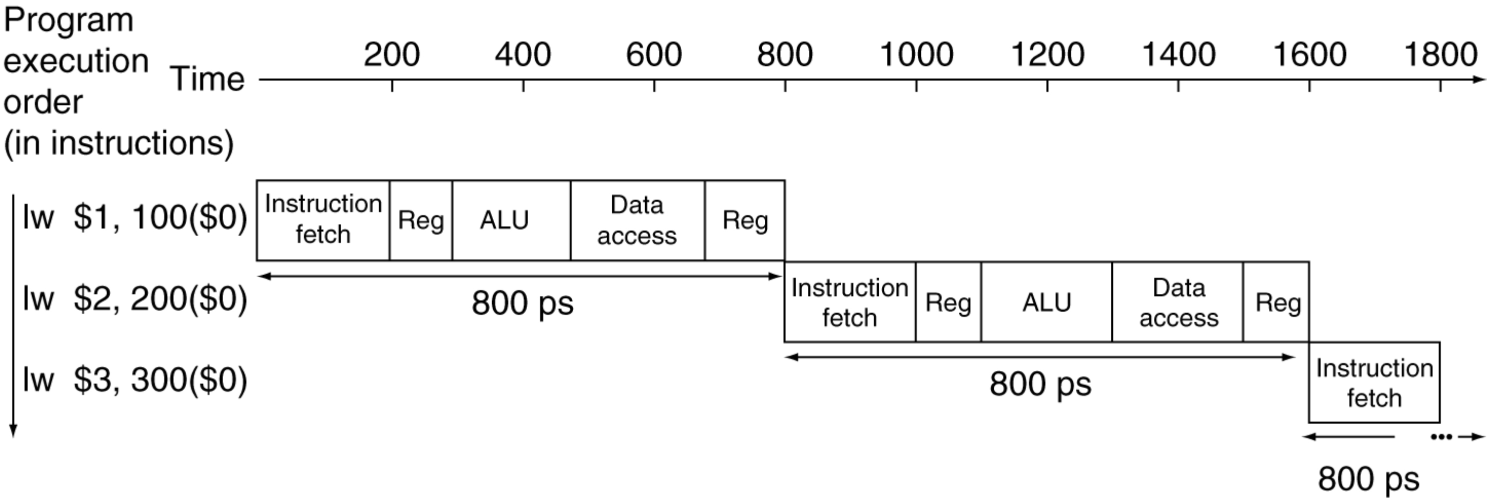

Instruction Fetch
Instruction Decode / Register Read
Execute
Memory
Write Back
Hazard Detection Unit
Stall
rsE
rtE
rdE
rd
0
1
WriteAddrM
WriteAddrW
Forwarding Unit
FwdBW
FwdAW
FwdBM
FwdAM
1
0
P C
En
Addr
Read Data
Instruction Memory
En
rs
rt
Write Addr
Write Data
Read Addr1
Read Data1
Register File
Read Addr2
Read Data2
Write Enable
Zero
A
A L U
Out
0
1
B
ALUOutM
Data Memory
Addr
Read Data
Write Data
Write Enable
0
1
WriteDataW
imm
Sign / Zero Extension
<<2
+4
+
En
ExtOp
ALUSrc
ALUCtr
RegDst
Branch
MemWr
MemToReg
RegWr
op
funct
Control Unit
IF/ID Registers
ID/EX Registers
EX/MEM Registers
MEM/WB Registers

Review: Pipelining Performance

Single-cycle

T_c = 800 ps

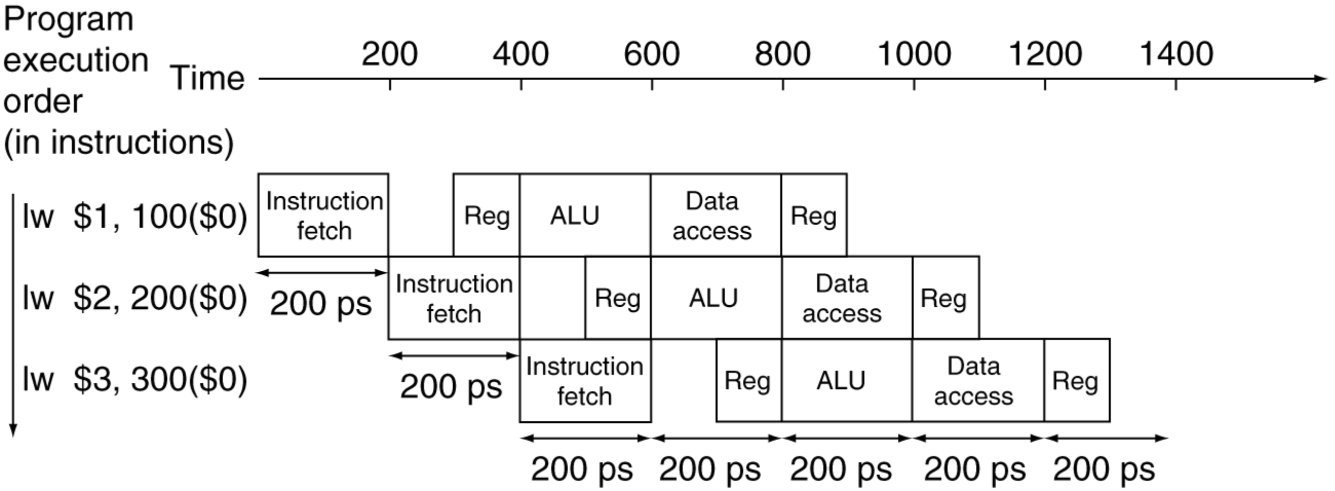
f = 1.25GHz



Pipelined

T_c = 200 ps

f = 5GHz



Review: Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) Structural hazard

- A required resource is busy
  (e.g. needed in multiple stages)

2) Data hazard

- Data dependency between instructions
- Need to wait for previous instruction to complete its data read/write

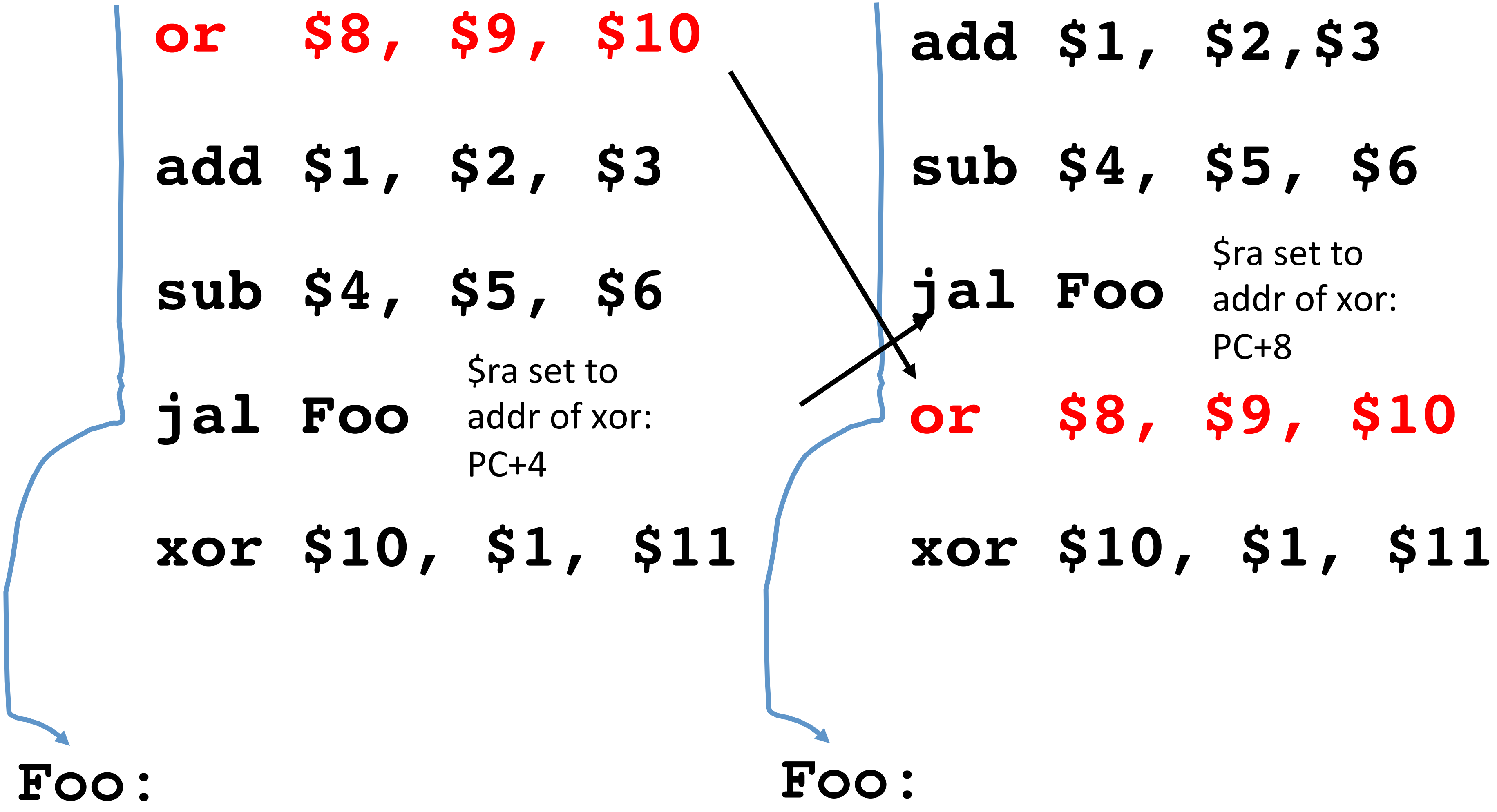
3) Control hazard

- Flow of execution depends on previous instruction

Review: Resolving Control Hazards

- Jumps would require inserting one bubble (since we don’t know the jump immediate until decode)
  - Jump delay slot lets us put a useful instruction here
- Branches would require inserting two bubbles (since we don’t do the equality comparison until the ALU stage)
  - Move equality comparison to decode stage, add branch delay slot

Example: Nondelayed vs. Delayed Jump



Nondelayed Jump

```
or   $8, $9, $10

add $1, $2, $3

sub $4, $5, $6

jal Foo

xor $10, $1, $11
```

$ra set to addr of xor: PC+4

```
Foo:
```

Delayed Jump

```
add $1, $2,$3

sub $4, $5, $6

jal Foo

or   $8, $9, $10

xor $10, $1, $11
```

$ra set to addr of xor: PC+8

```
Foo:
```

Review: Resolving (many) Data Hazards: Forwarding

- Forward result as soon as it is available
  - OK that it’s not stored in RegFile yet

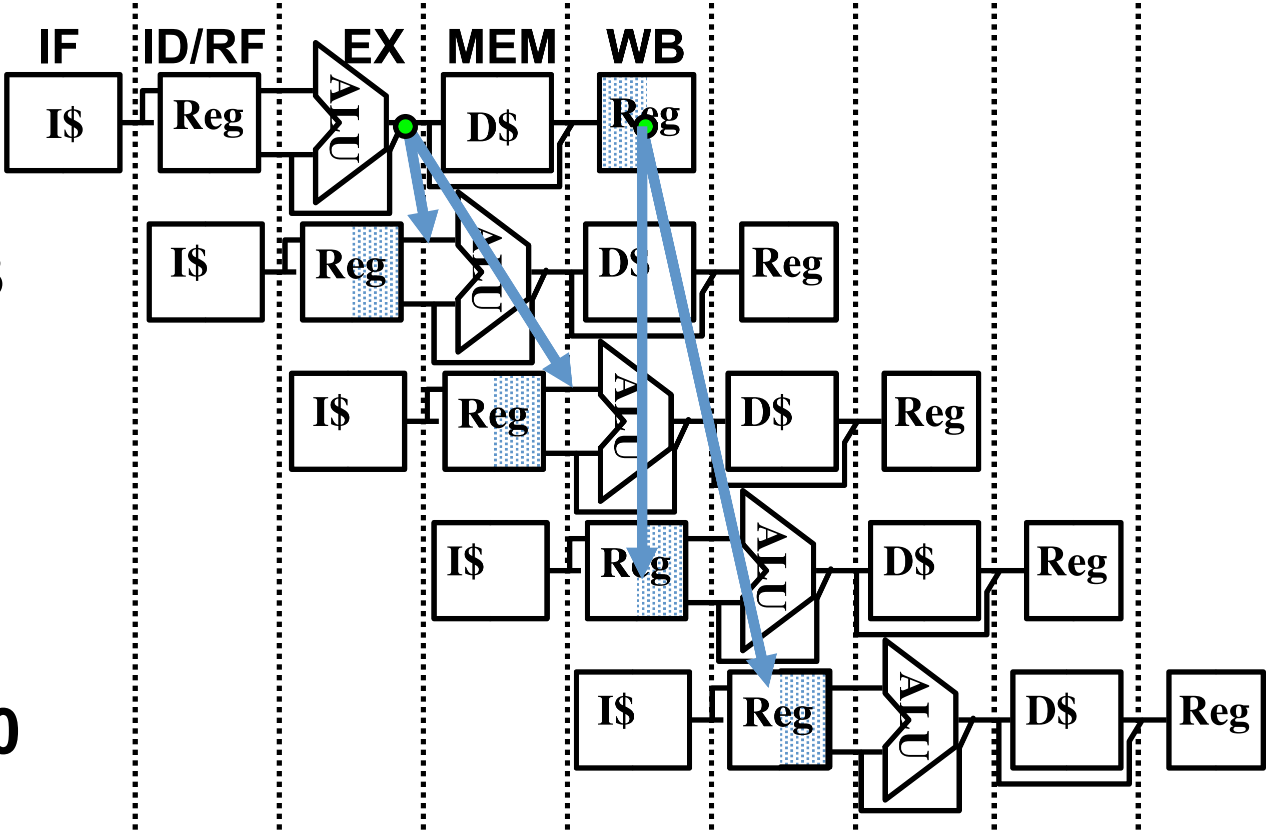
add $t0,$t1,$t2

sub $t4,$t0,$t3

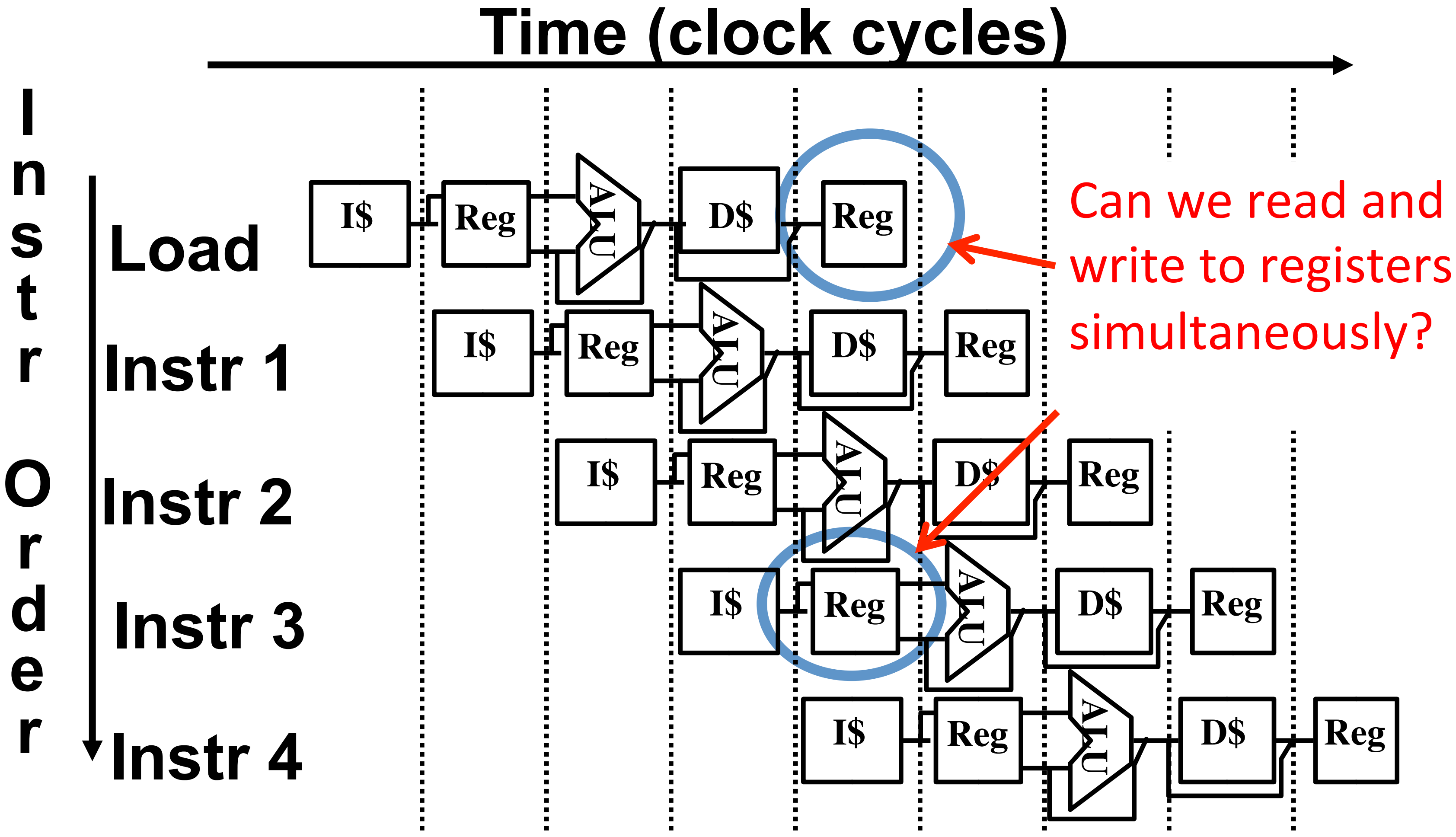
and $t5,$t0,$t6

or $t7,$t0,$t8

xor $t9,$t0,$t10

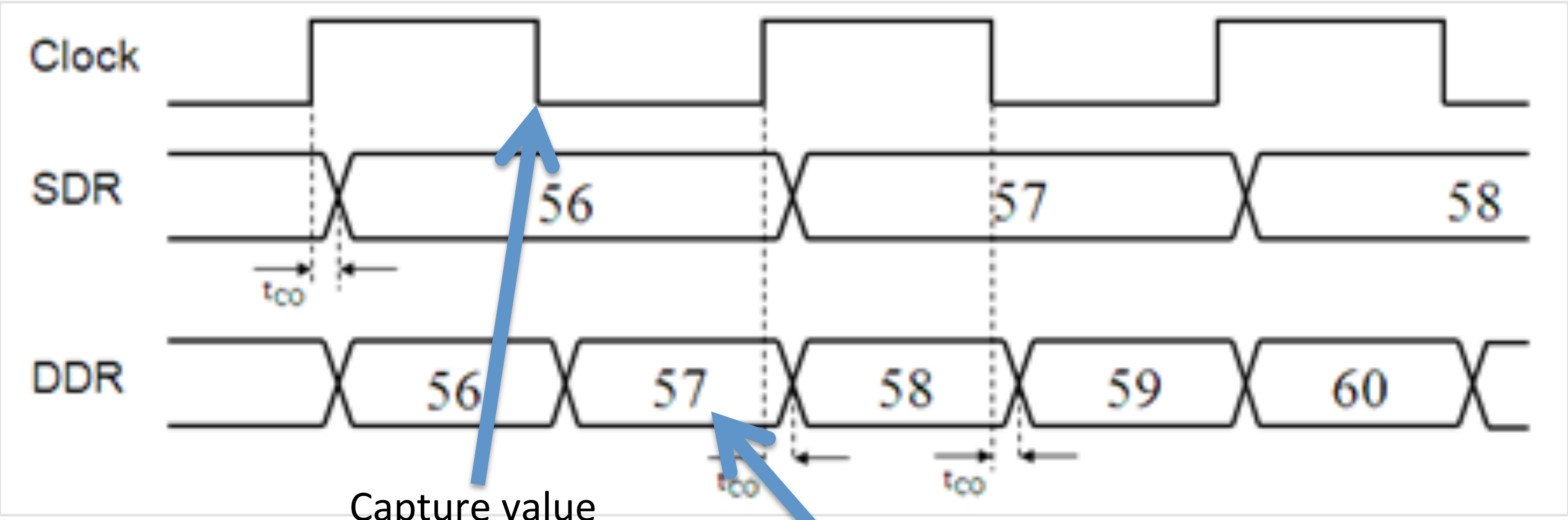


Review: Structural Hazard #1: Registers



Can we read and write to registers simultaneously?

Review: Solving Regfile Structural Hazard: Double Pumping (2/3)



Capture value at this (falling) edge

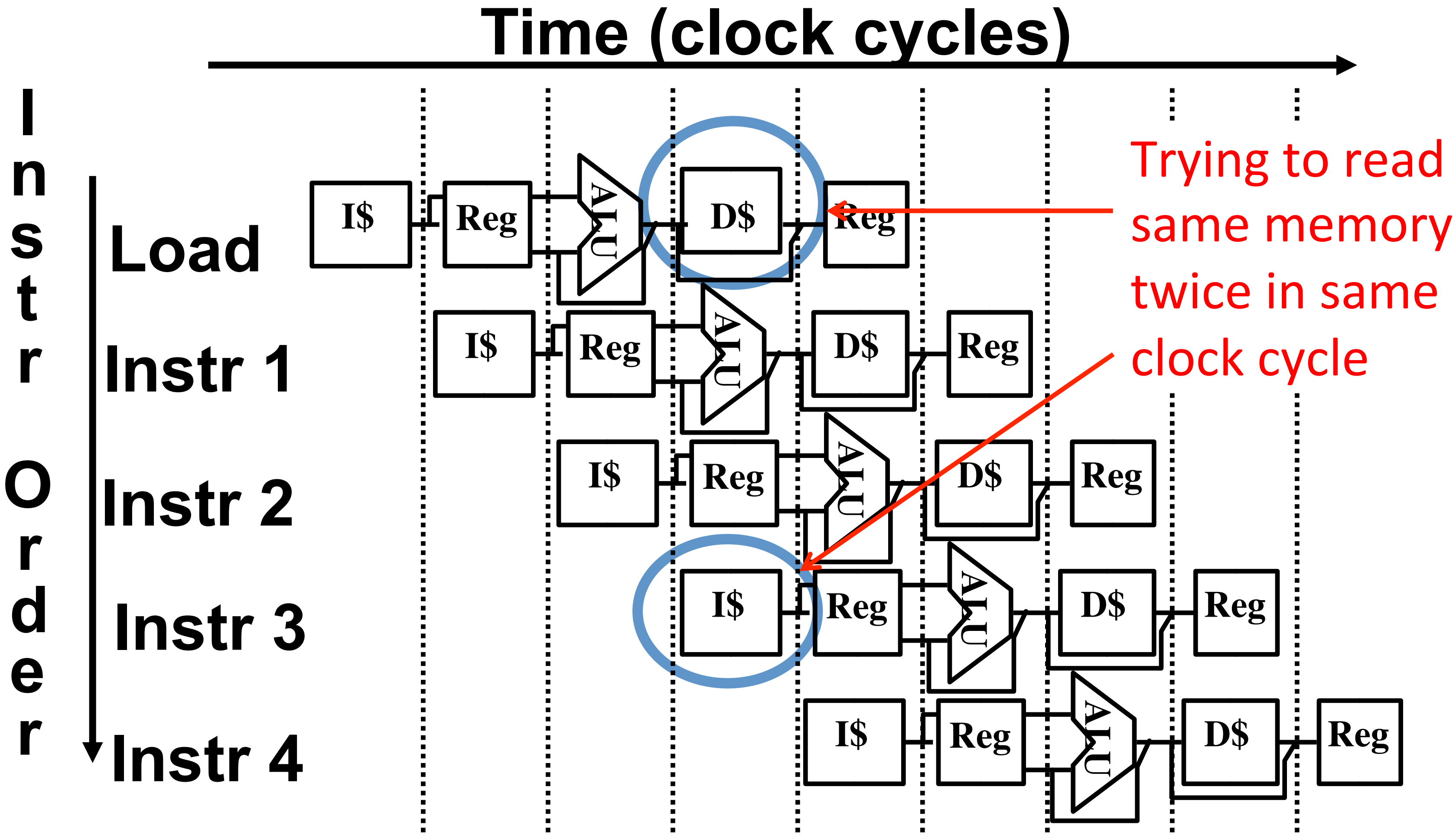
Can read the new value here

* ***<u>This is not how registers work in general!</u>***

Review: Structural Hazard #2: Single Memory

- MIPS pipeline with a single memory?
  - Load/Store requires memory access for data
  - Instruction fetch would have to *stall* for that cycle
    - Causes a pipeline "*bubble*"
- Hence, pipelined datapaths require separate instruction/data memories
  - Separate L1 I$ and L1 D$ take care of this

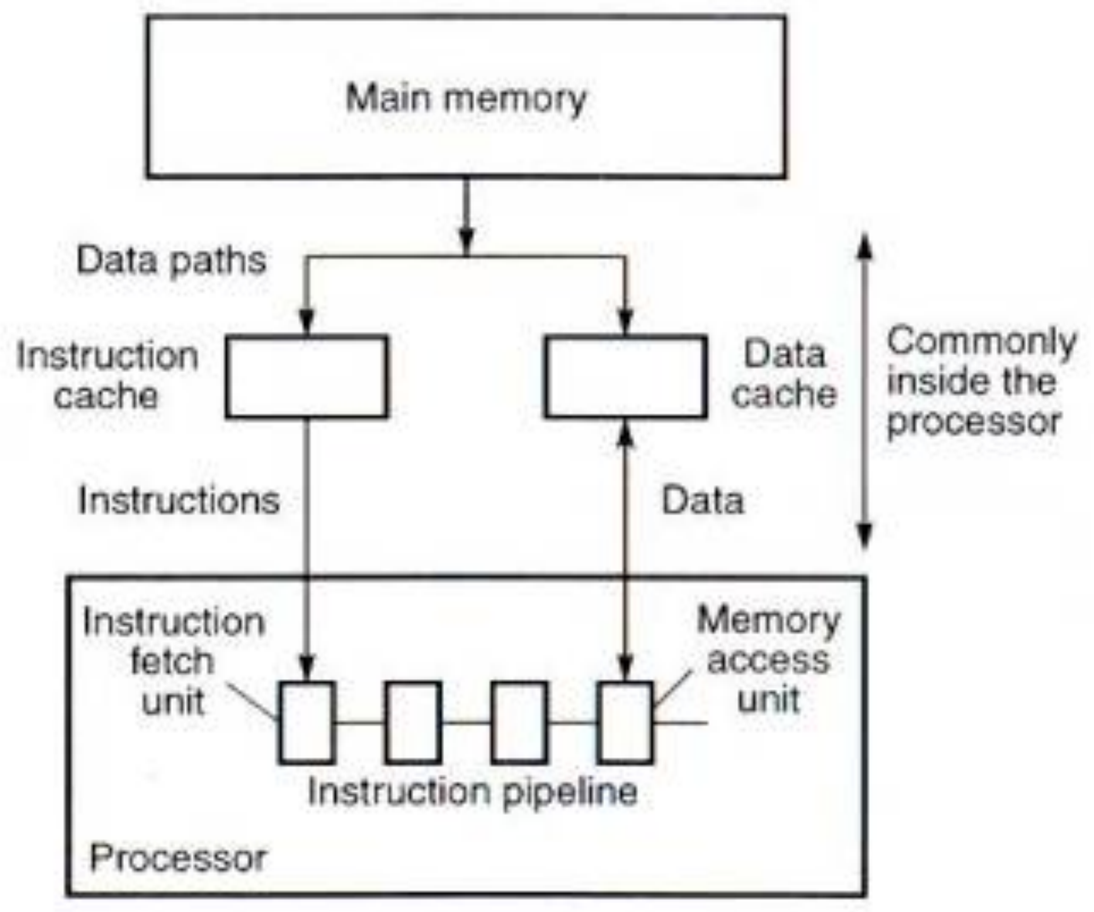
Review: Structural Hazard #2: Single Memory



Trying to read same memory twice in same clock cycle

Review: Solving Structural Hazard #2 with Caches

Today, we'll add caches

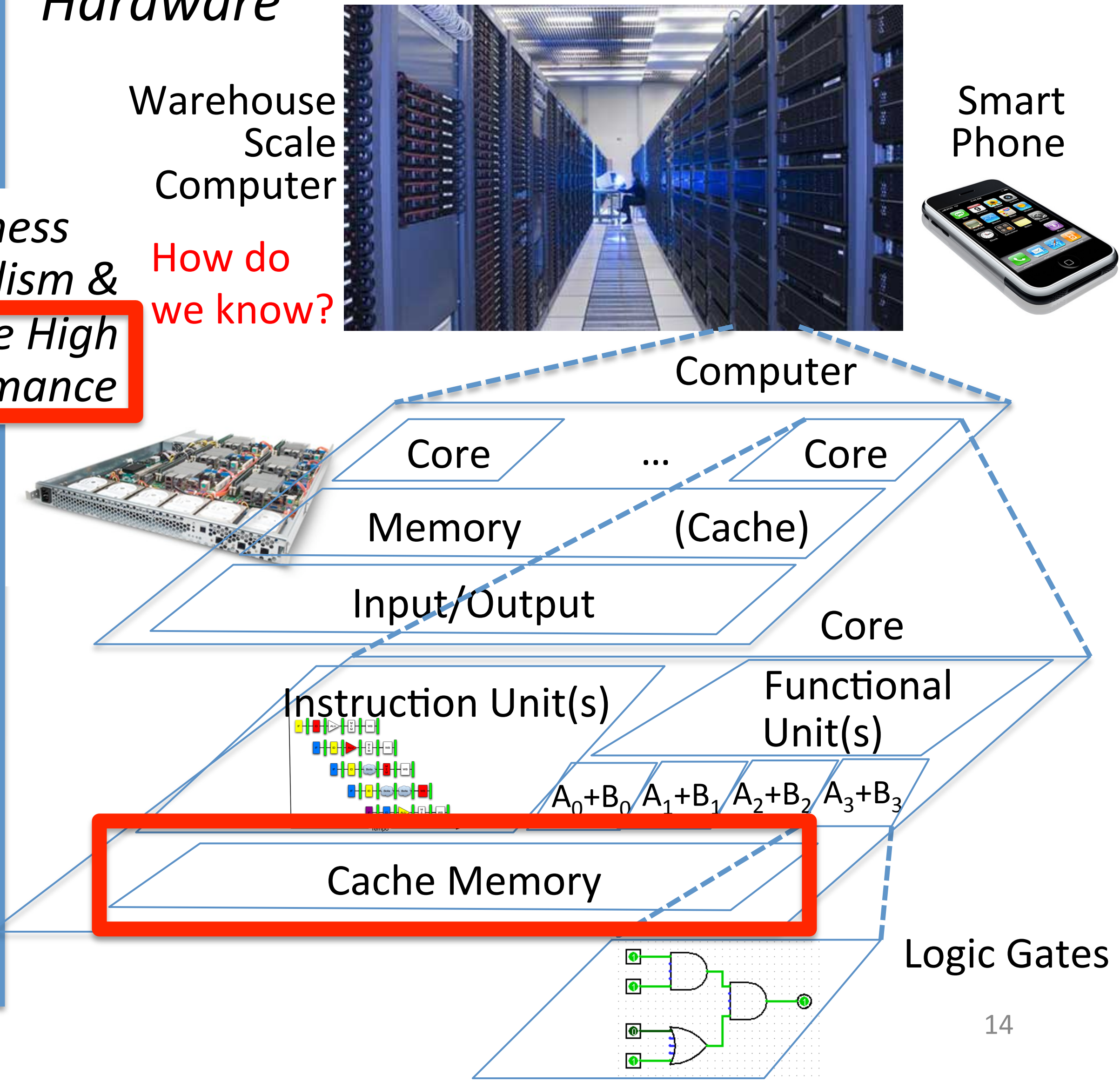


New-School Machine Structures (It’s a bit more complicated!)

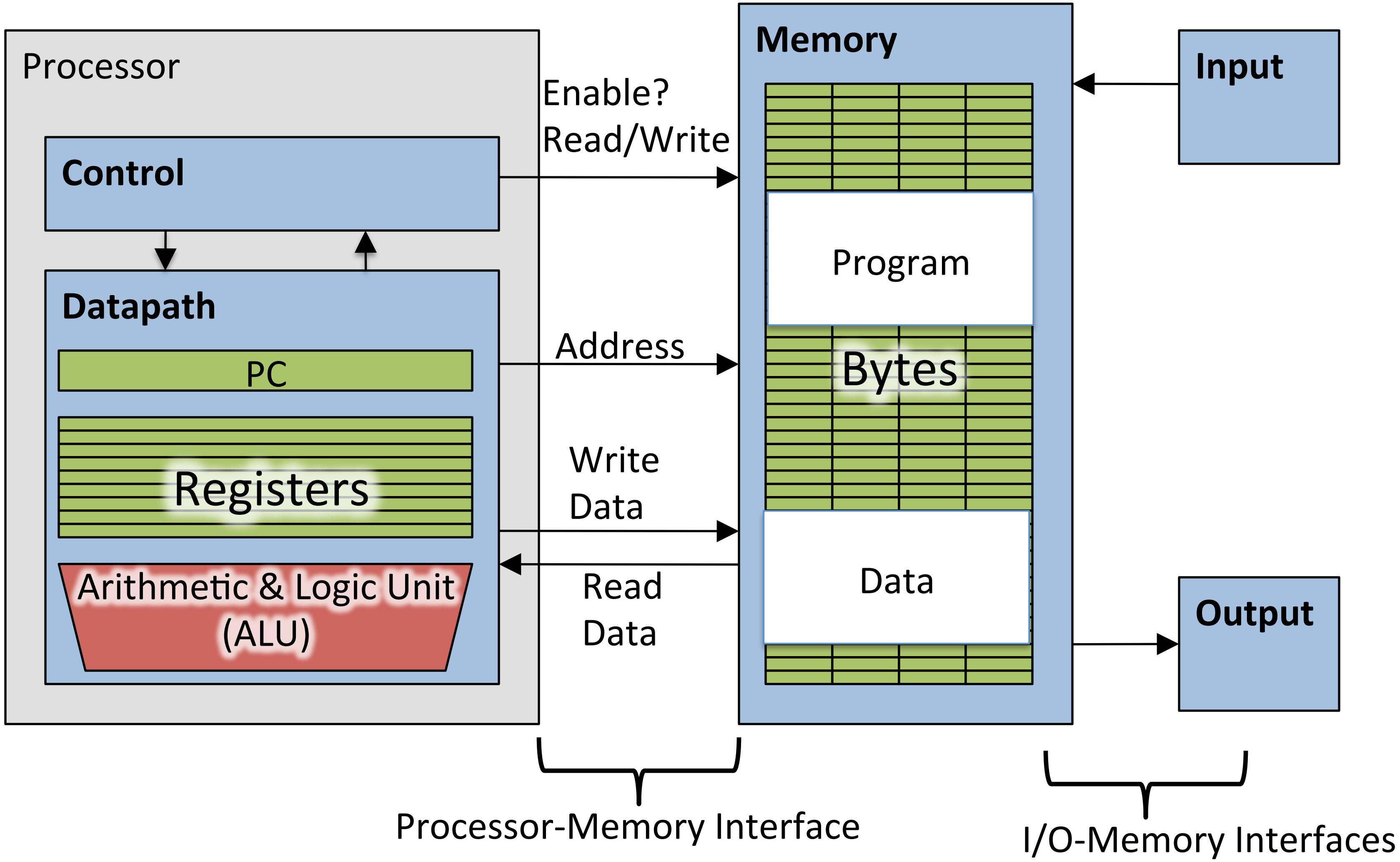
Software | *Hardware*

- 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

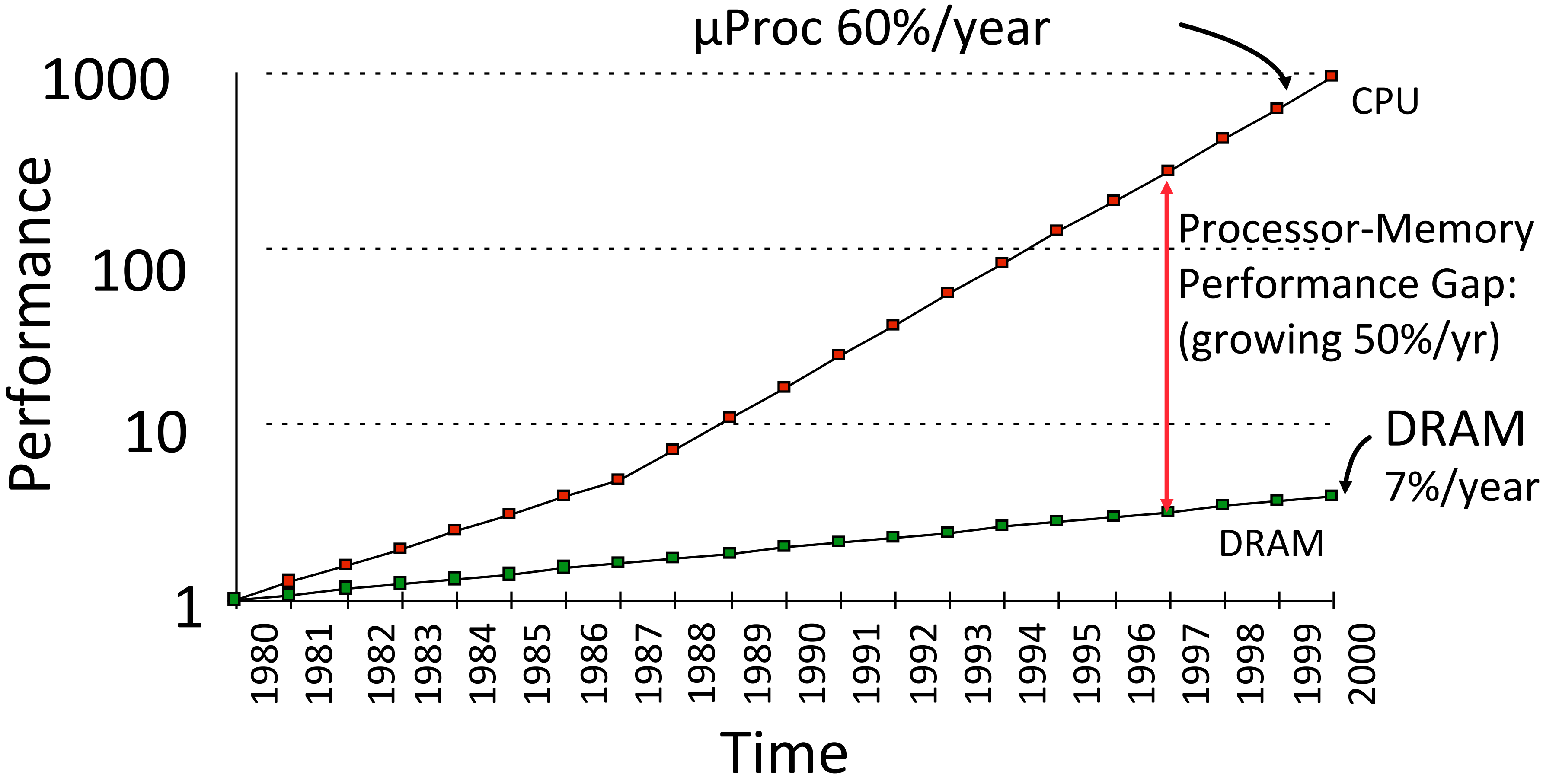
Harness Parallelism &
Achieve High Performance



Components of a Computer



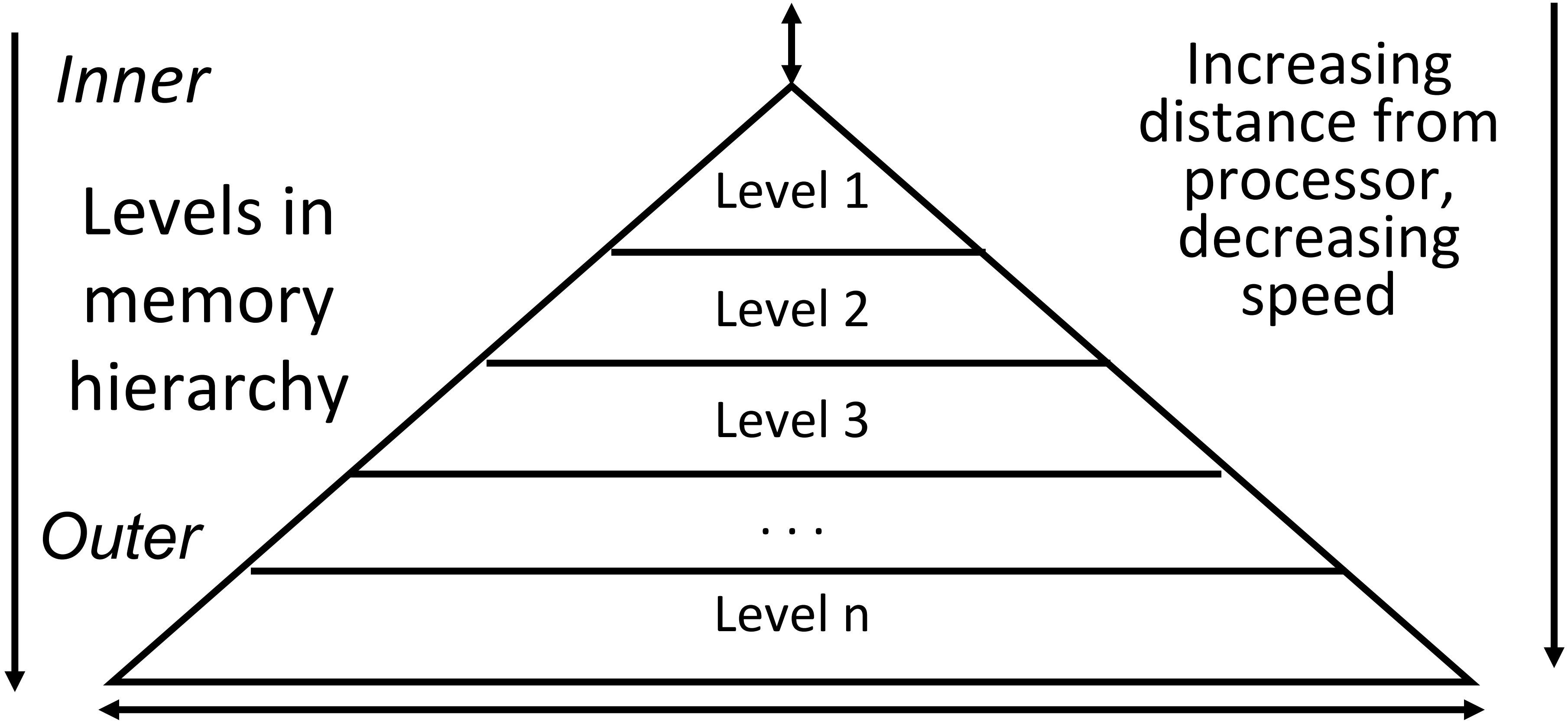
Processor-DRAM Gap (latency)



1980 microprocessor executes ~one instruction in same time as DRAM access
2015 microprocessor executes ~1000 instructions in same time as DRAM access

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. Why?

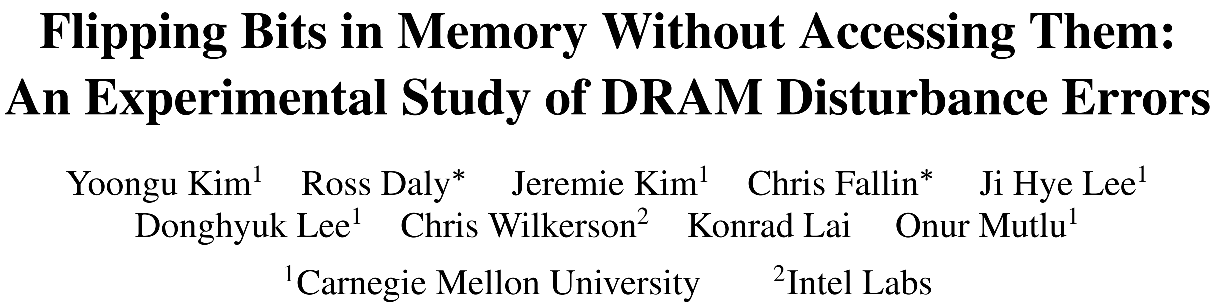
Library Analogy

- Writing a report based on books on reserve
  - E.g., works of J.D. Salinger
- Go to library to get reserved book 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 0.00001% of books in UC Berkeley libraries

Administrivia

- HW3 Out
- Proj2-2 Out
- Midterm regrades due today @ 23:59:59
- Register your Project 3 Teams

In the News: RowHammer Exploit



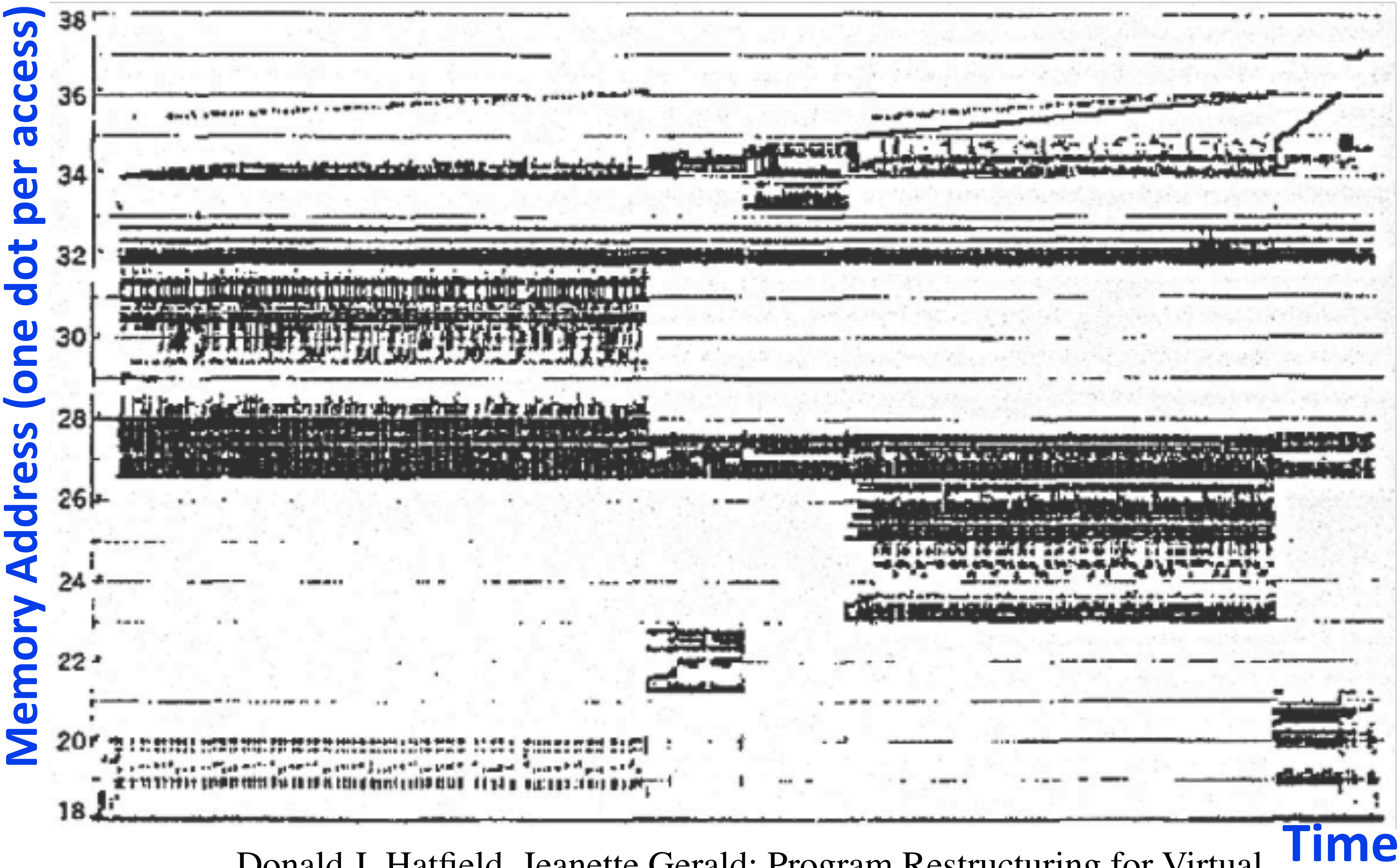
Flipping Bits in Memory Without Accessing Them:
An Experimental Study of DRAM Disturbance Errors

Yoongu Kim[1] Ross Daly* Jeremie Kim[1] Chris Fallin* Ji Hye Lee[1]
Donghyuk Lee[1] Chris Wilkerson[2] Konrad Lai Onur Mutlu[1]
[1]Carnegie Mellon University [2]Intel Labs

- CMU + Intel researchers found commercial DRAM chips susceptible to neighboring bits flipping if one row of memory accessed frequently
- Google Engineers figured out how to use this to gain root access on a machine! Almost all laptops susceptible, but server ECC memory helps reduce impact.

Break

Real Memory Reference Patterns

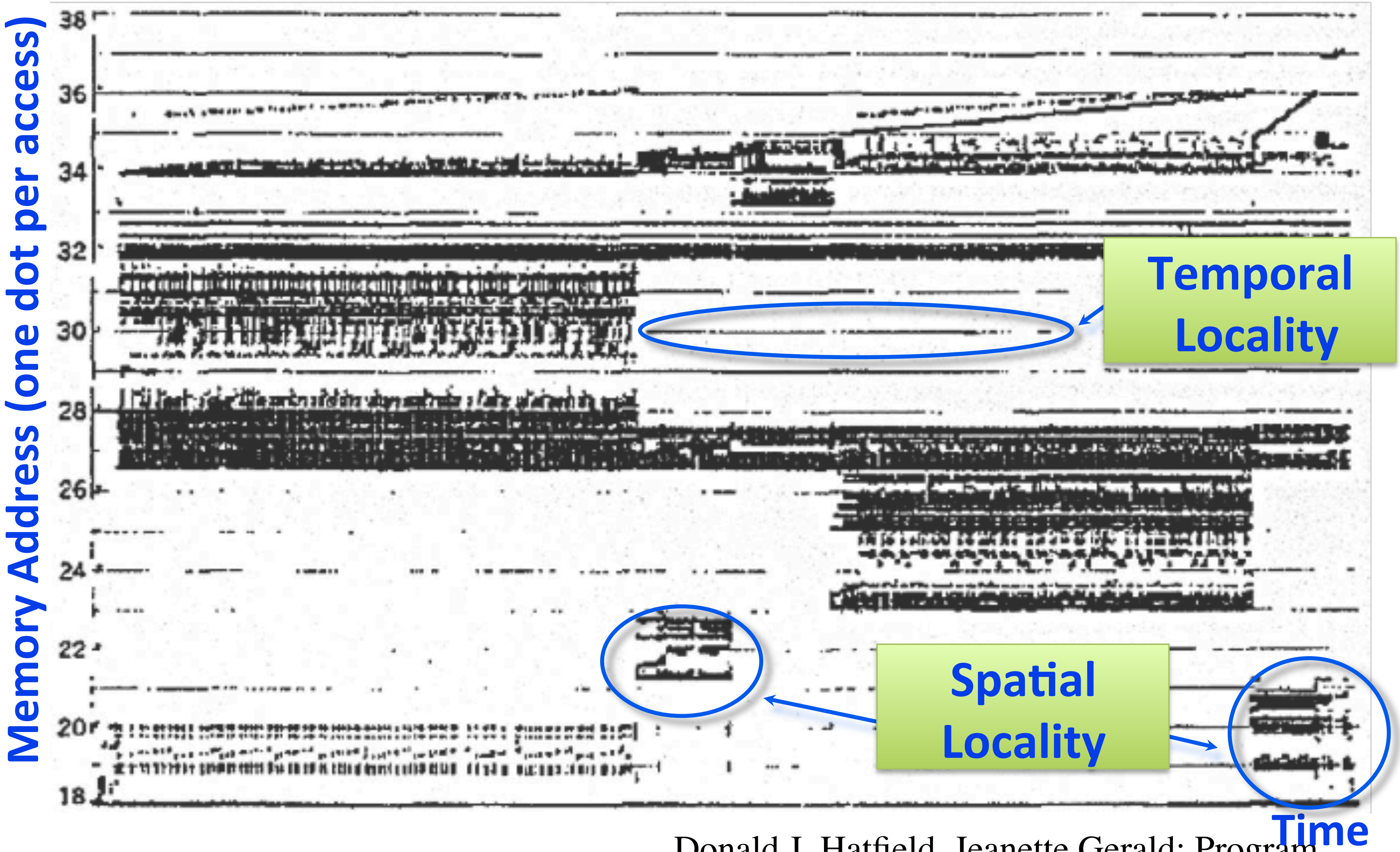


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Big Idea: Locality

- *Temporal Locality* (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Memory Reference Patterns



Temporal Locality

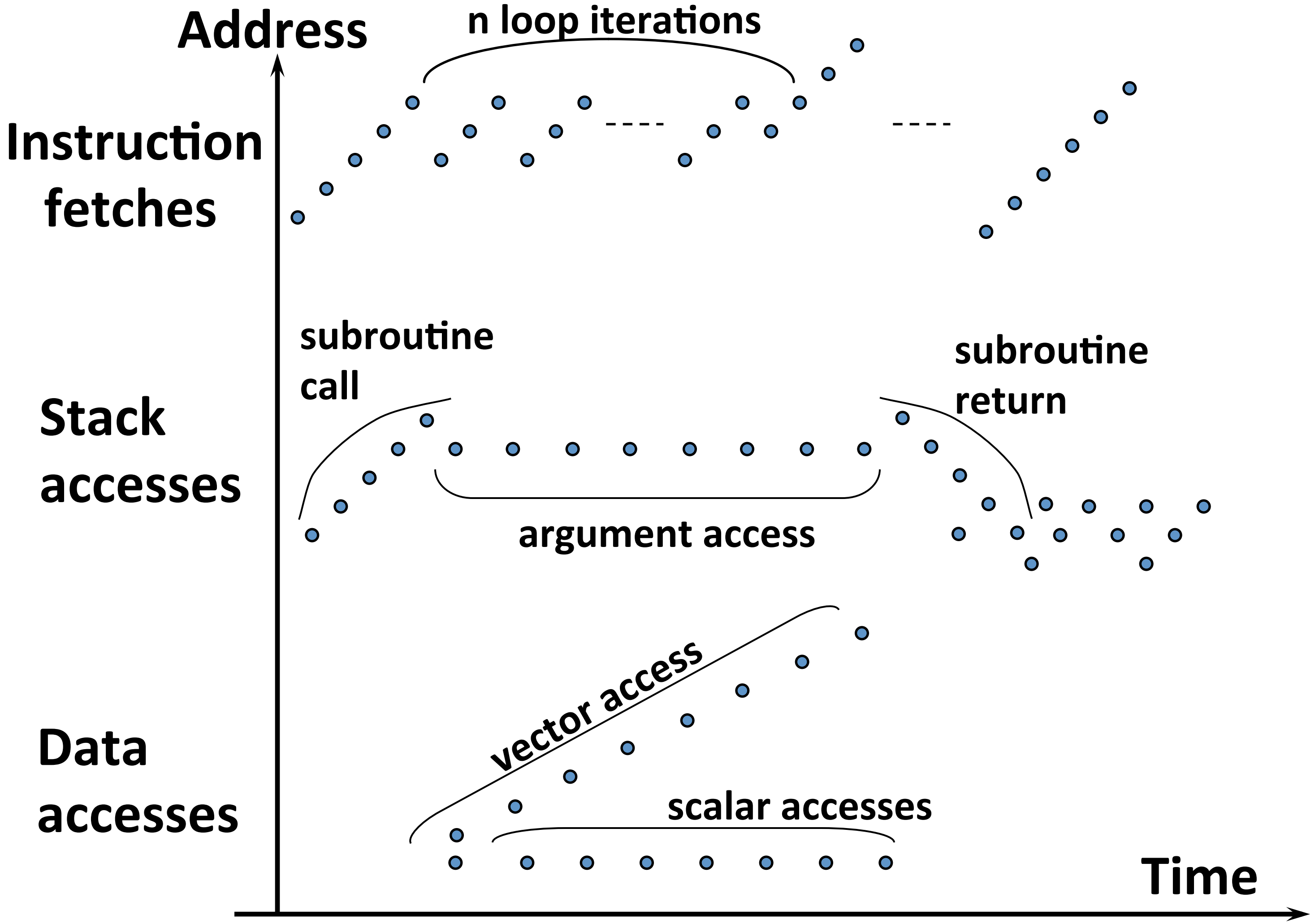
Spatial Locality

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
- What program structures lead to temporal and spatial locality in instruction accesses?
- In data accesses?

Memory Reference Patterns



Cache Philosophy

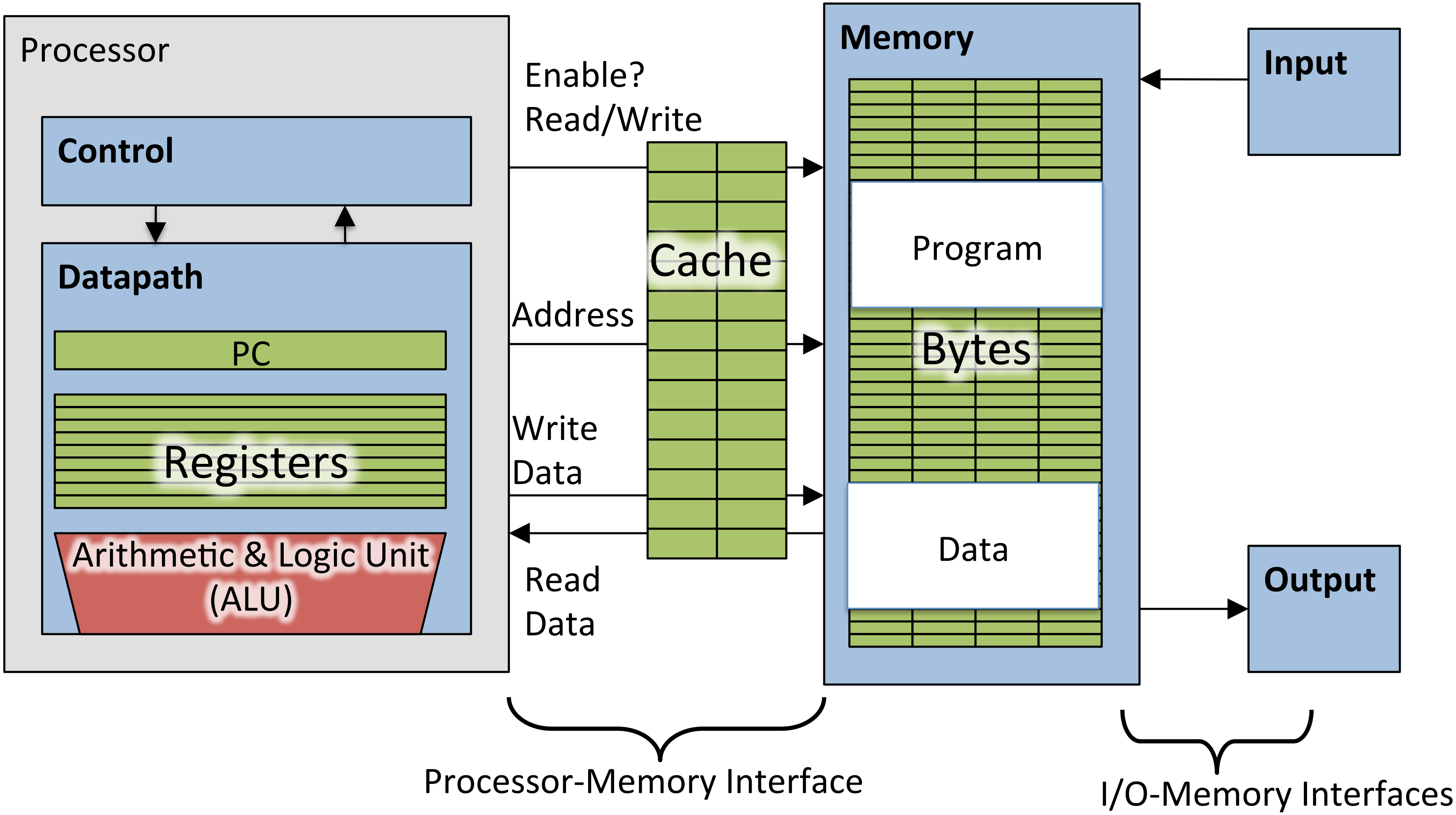
- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
  - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache
  - We’ll do this in Project 4

Memory Access without Cache

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

1. Processor issues address 1022$_{\text{ten}}$ to Memory
2. Memory reads word at address 1022$_{\text{ten}}$ (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 (similar to a hash)
  1. Processor issues address 1022_{ten} to Cache
  2. 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
  3. Processor loads 99 into register $t0

Cache “Tags”

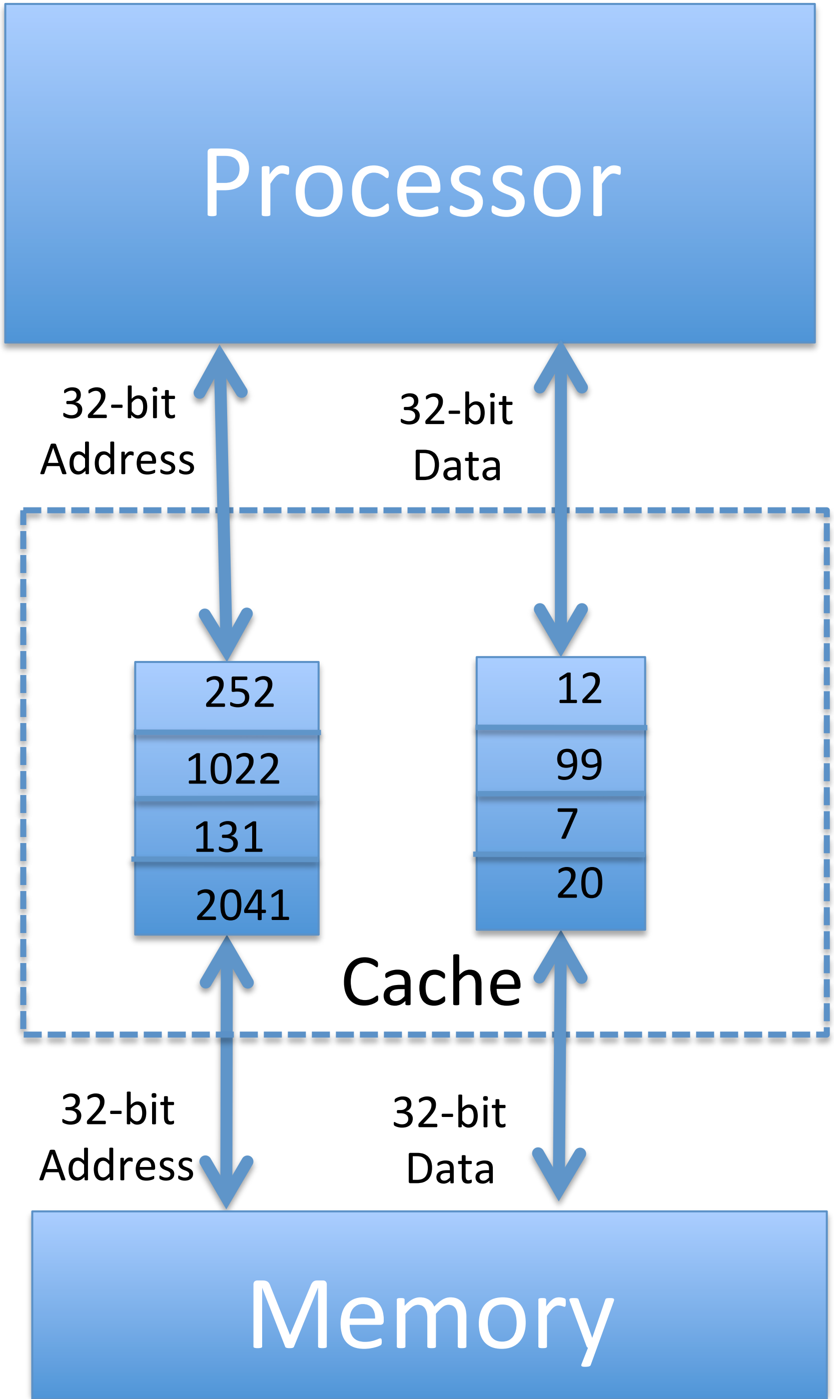
- Need way to tell if have copy of location in memory so that can decide on hit or miss
- On cache miss, put memory address of block in “tag address” of cache block
  - 1022 placed in tag next to data from memory (99)

Tag	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

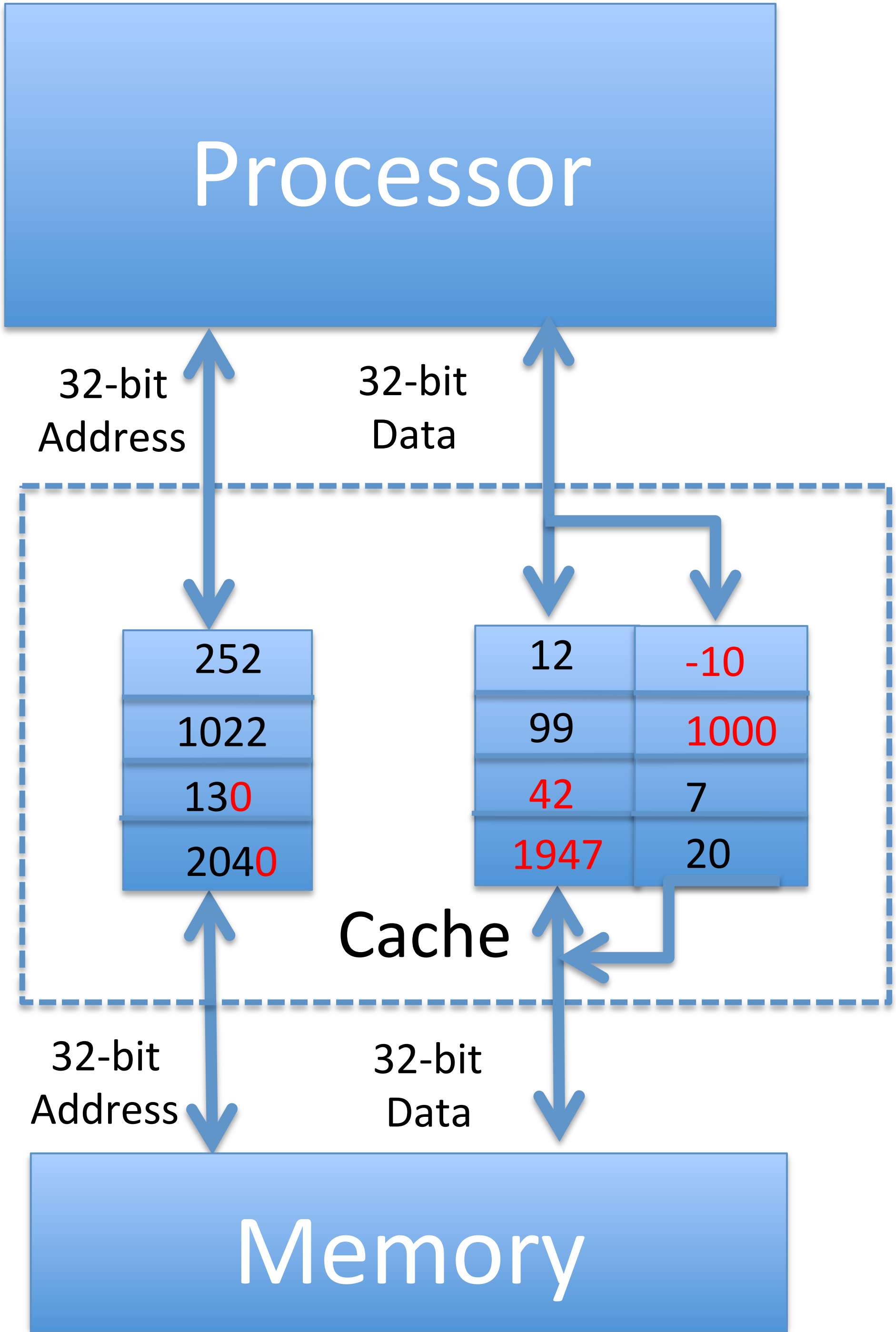
- Blocks must be aligned in pairs, otherwise could get same word twice in cache

⇒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



Processor

32-bit Address

32-bit Data

252	12	-10
1022	99	1000
130	42	7
2040	1947	20

Cache

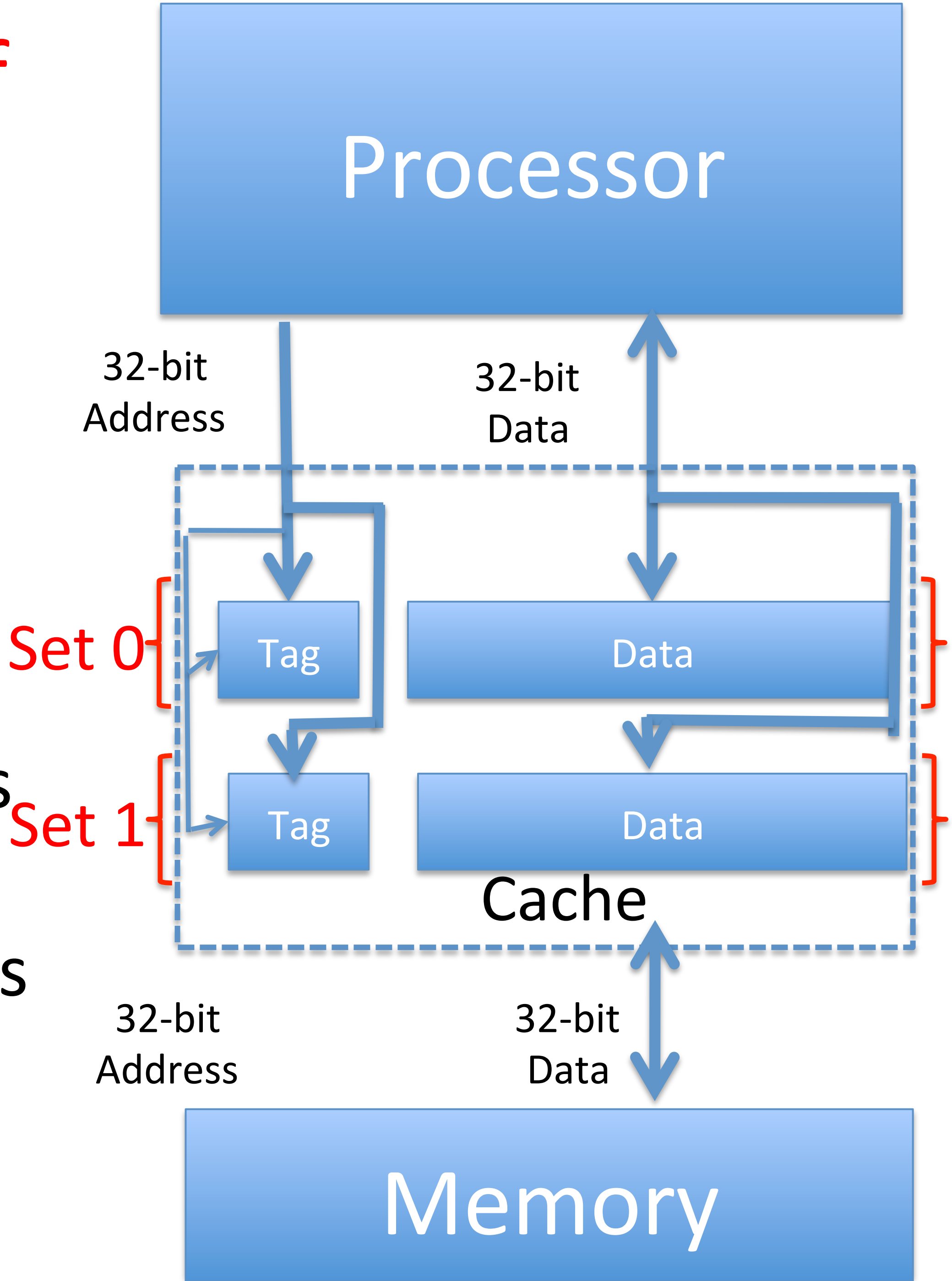
32-bit Address

32-bit Data

Memory

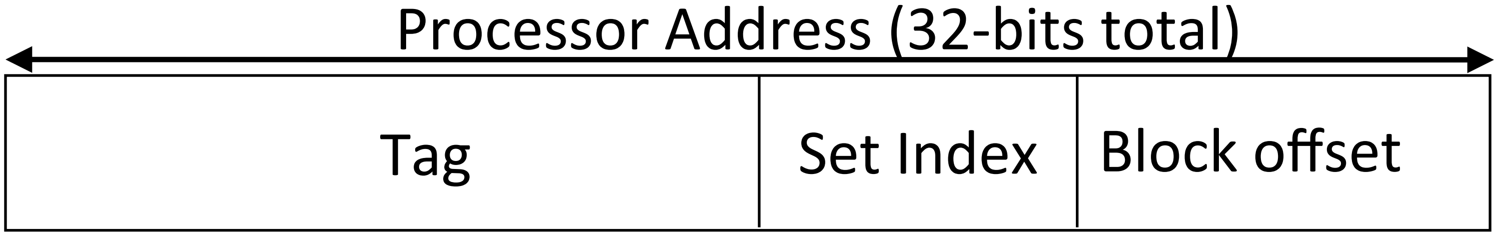
Hardware Cost of Cache

- Need to compare every tag to the Processor address
- Comparators are expensive
- Optimization: 2 sets => ½ comparators
- 1 Address bit selects which set



Processor Address Fields used by Cache Controller

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



Processor Address (32-bits total)

Tag

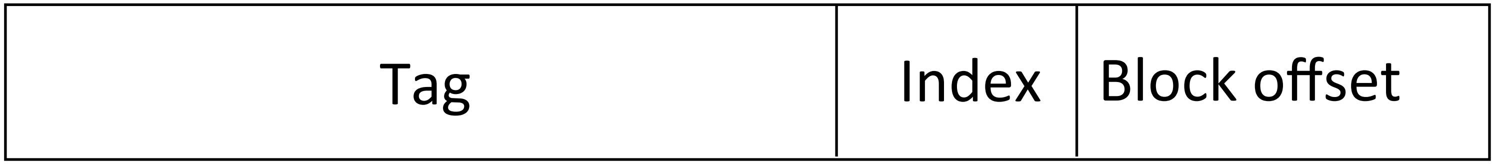
Set Index

Block offset

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

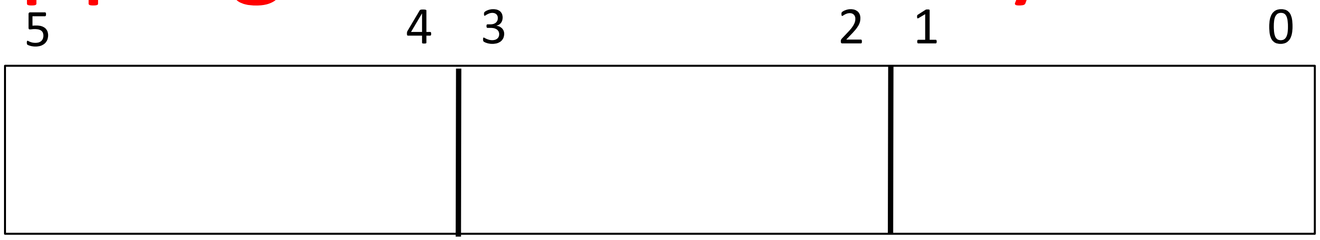
What is limit to number of sets?

- Can save more comparators if have more than 2 sets
- Limit: As Many Sets as Cache Blocks – only needs one comparator!
- Called “Direct-Mapped” Design



Tag	Index	Block offset

Mapping a 6-bit Memory Address



Mem Block Within $ Block *Tag*

Block Within $ *Index*

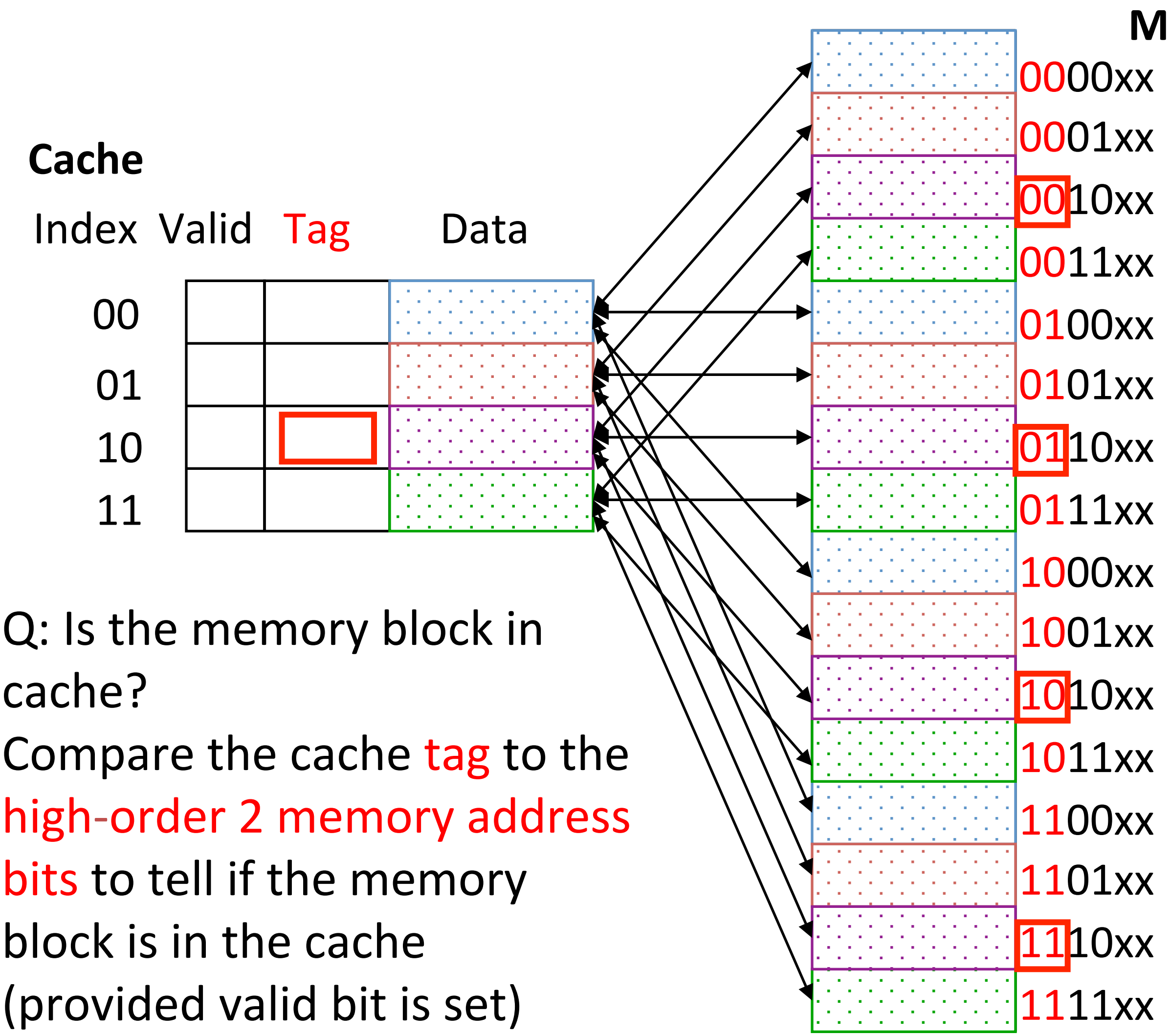
Byte Offset Within Block (e.g., Word)

- In example, block size is 4 bytes/1 word (it could be multi-word)
- Memory and cache blocks are 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
- Byte within block: low order two bits, ignore! (nothing smaller than a 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



Cache

Index	Valid	Tag	Data
00			
01			
10			
11			

Main Memory

0000xx
0001xx
0010xx
0011xx
0100xx
0101xx
0110xx
0111xx
1000xx
1001xx
1010xx
1011xx
1100xx
1101xx
1110xx
1111xx

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 next 2 low-order memory address bits – the index – to determine which cache block (i.e., modulo the number of blocks in the cache)

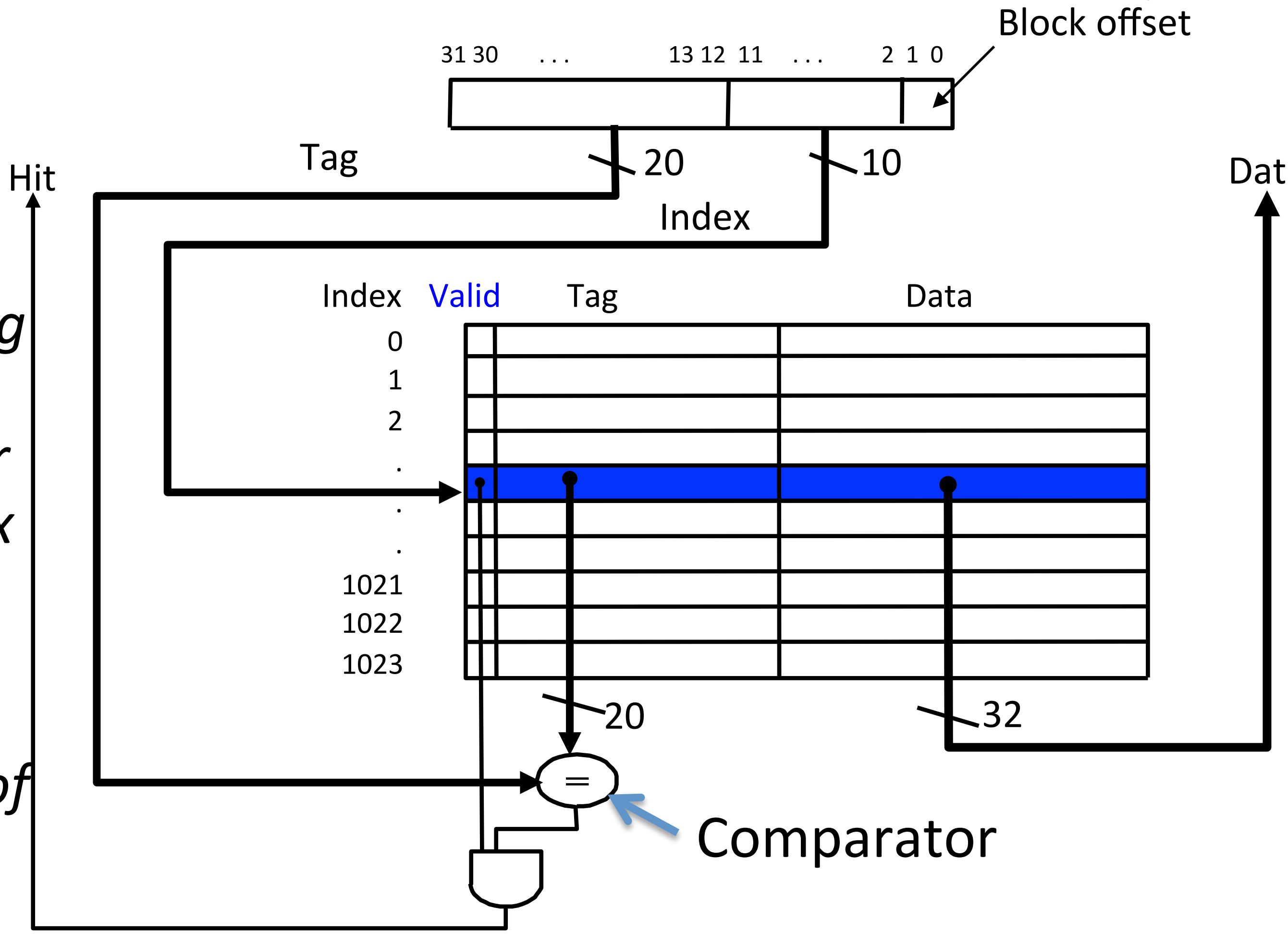
Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

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

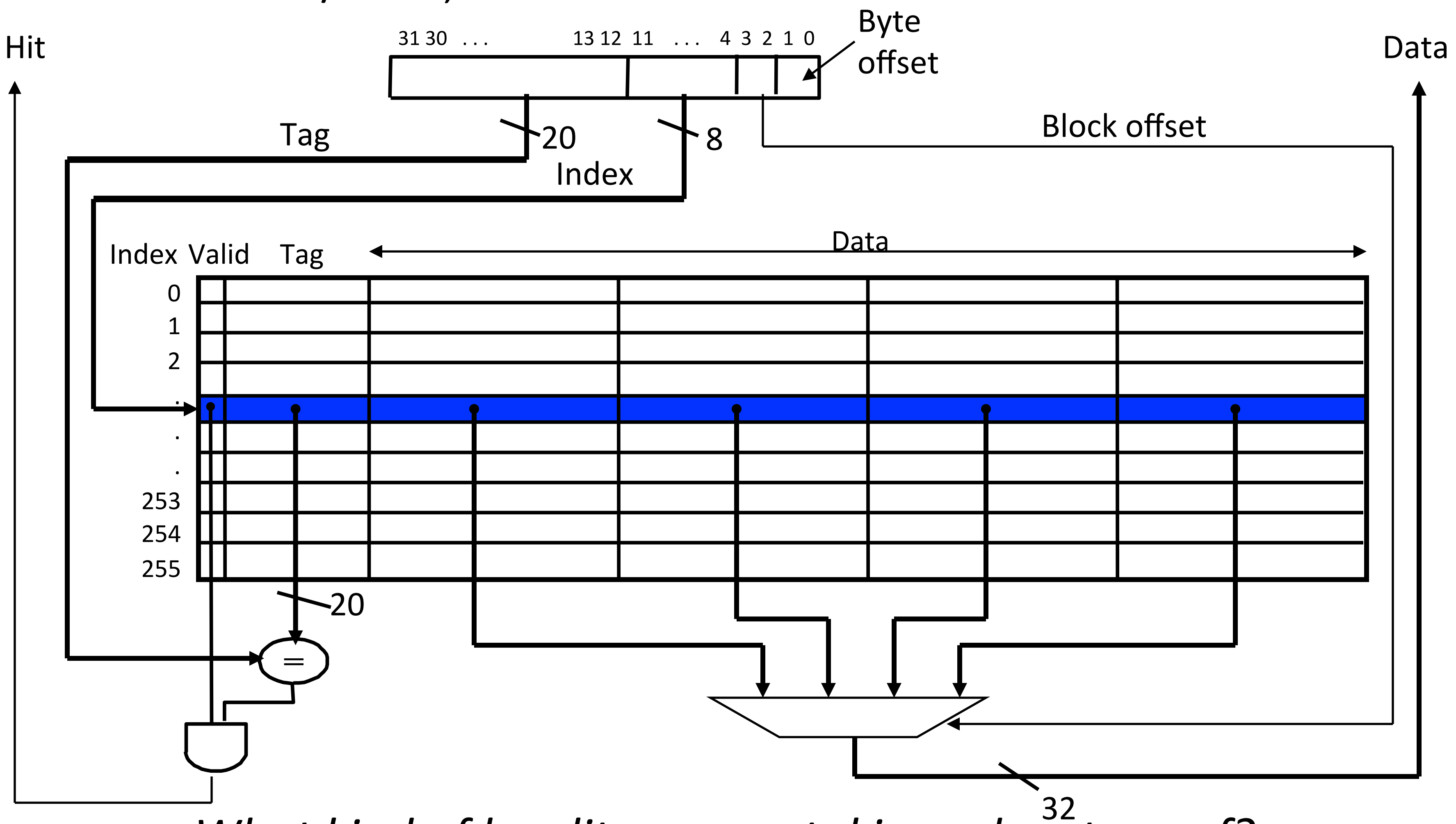


Read data from cache instead of memory if a Hit

What kind of locality are we taking advantage of?

Multiword-Block Direct-Mapped Cache

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



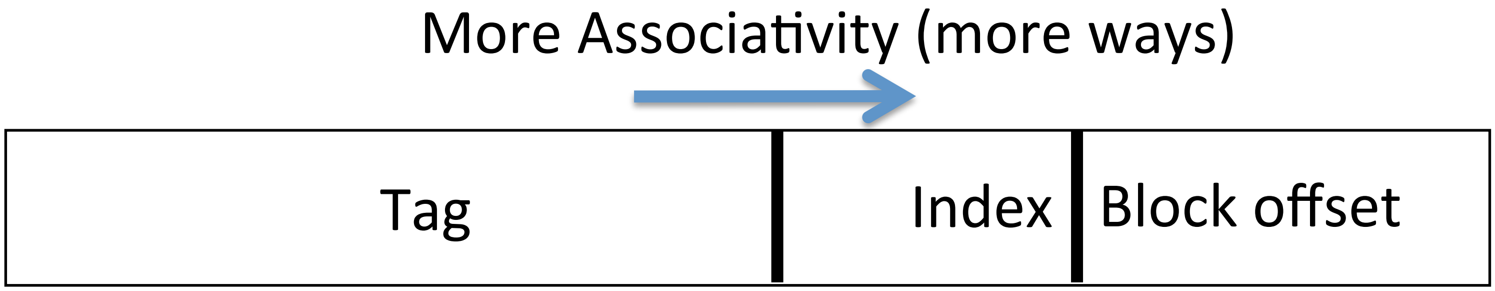
What kind of locality are we taking advantage of?

Cache Names for Each Organization

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

Range of Set-Associative Caches

- For a fixed-size cache, 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



More Associativity (more ways)

Tag | Index | Block offset

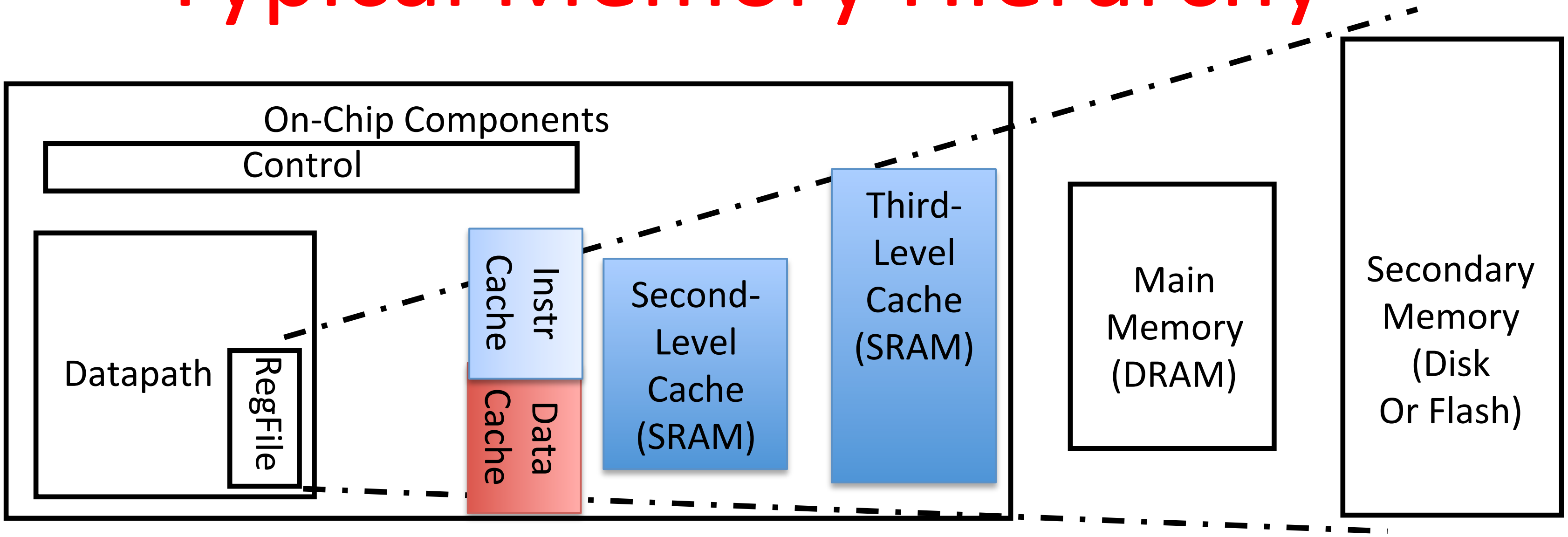
Note: IBM persists in calling sets "ways" and ways "sets". They're wrong.

Clickers/Peer Instruction

- For a cache with constant total capacity, if we increase the number of ways by a factor of 2, which statement is false:
- A: The number of sets could be doubled
- B: The tag width could decrease
- C: The number of tags could stay the same
- D: The block size could be halved
- E: Tag width must increase

Break

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

Handling Stores with Write-Through

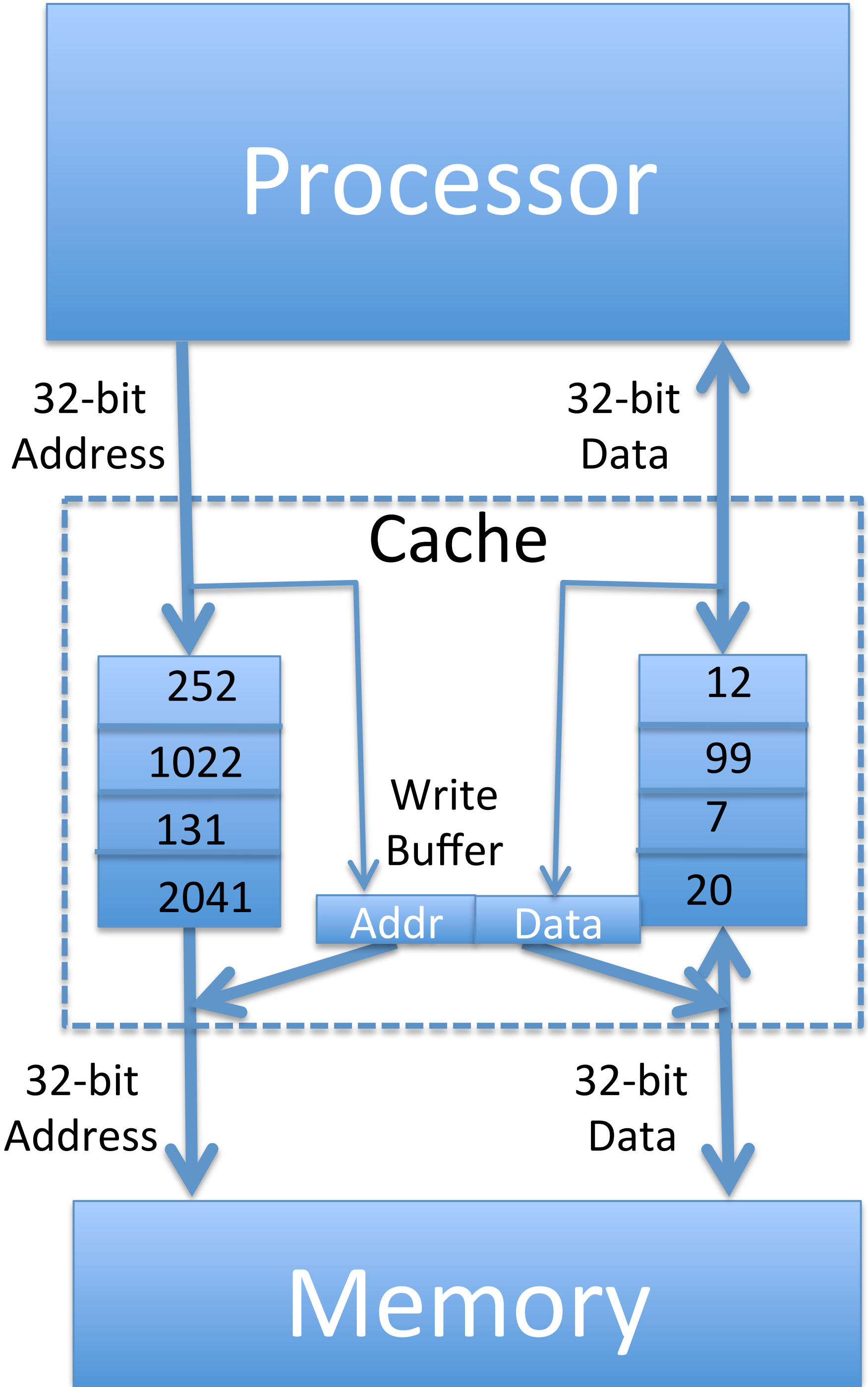
- Store instructions write to memory, changing values
- Need to make sure cache and memory have same values on writes: 2 policies

1) Write-Through Policy: write cache and write *through* the cache to memory

  - Every write eventually gets to memory
  - Too slow, so include Write Buffer to allow processor to continue once data in Buffer
  - Buffer updates memory in parallel to processor

Write-Through Cache

- Write both values in cache and in memory
- Write buffer stops CPU from stalling if memory cannot keep up
- Write buffer may have multiple entries to absorb bursts of writes
- What if store misses in cache?



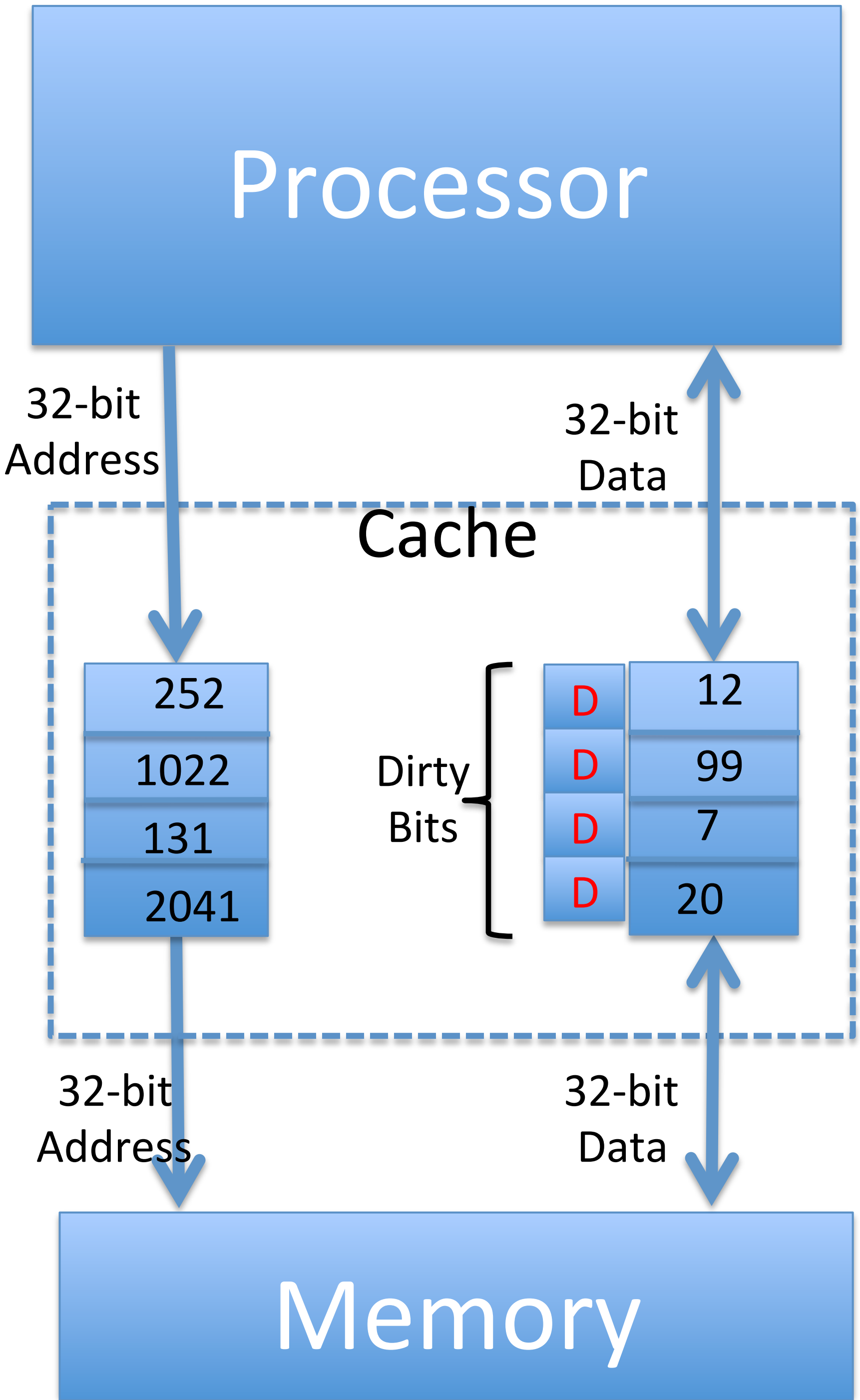
Handling Stores with Write-Back

2) Write-Back Policy: write only to cache and then write cache block *back* to memory when evict block from cache

- Writes collected in cache, only single write to memory per block
- Include bit to see if wrote to block or not, and then only write back if bit is set
  - Called "Dirty" bit (writing makes it "dirty")

Write-Back Cache

- Store/cache hit, write data in cache *only* & set dirty bit
  - Memory has stale value
- Store/cache miss, read data from memory, then update and set dirty bit
  - “Write-allocate” policy
- Load/cache hit, use value from cache
- On any miss, write back evicted block, only if dirty. Update cache with new block and clear dirty bit.



Write-Through vs. Write-Back

- Write-Through:
  - Simpler control logic
  - More predictable timing simplifies processor control logic
  - Easier to make reliable, since memory always has copy of data

- Write-Back
  - More complex control logic
  - More variable timing (0,1,2 memory accesses per cache access)
  - Usually reduces write traffic
  - Harder to make reliable, sometimes cache has only copy of data

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
- Cache design choice:
  - Write-Through vs. Write-Back