CS 110
Computer Architecture

Caches Part 1

Instructor:
Sören Schwertfeger

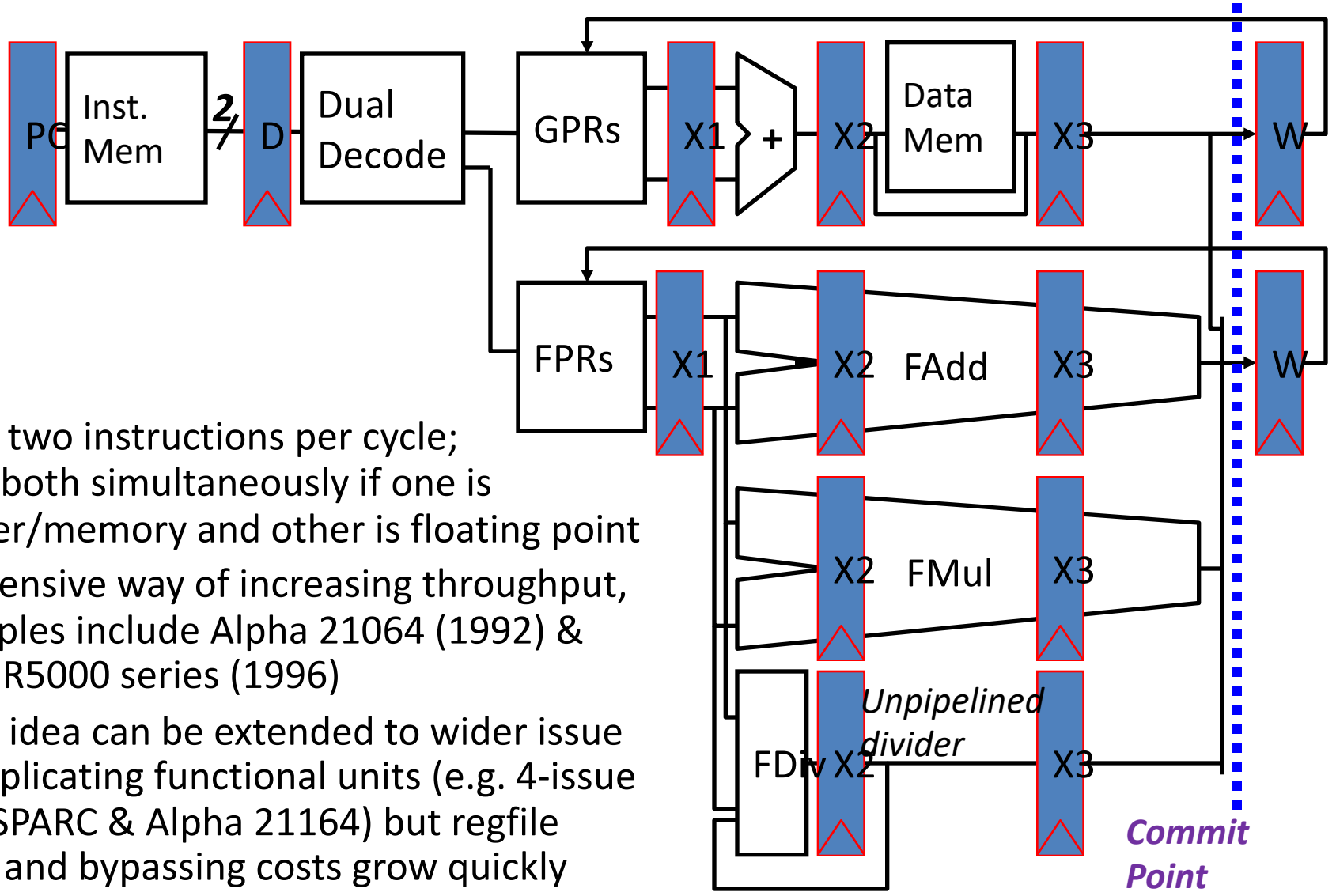
https://robotics.shanghaitech.edu.cn/courses/ca

School of Information Science and Technology SIST

ShanghaiTech University

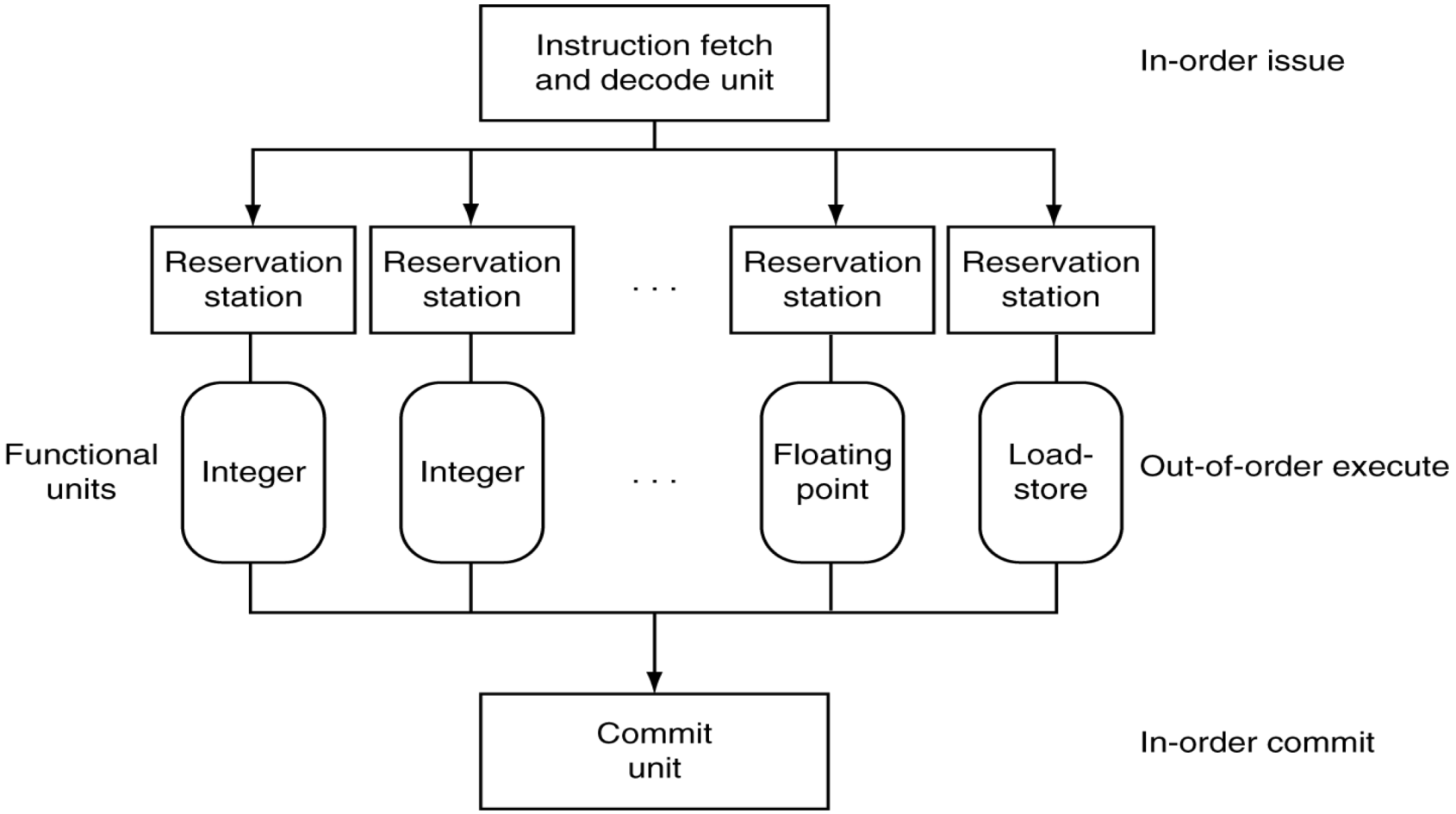
Slides based on UC Berkley's CS61C

In-Order Superscalar Pipeline

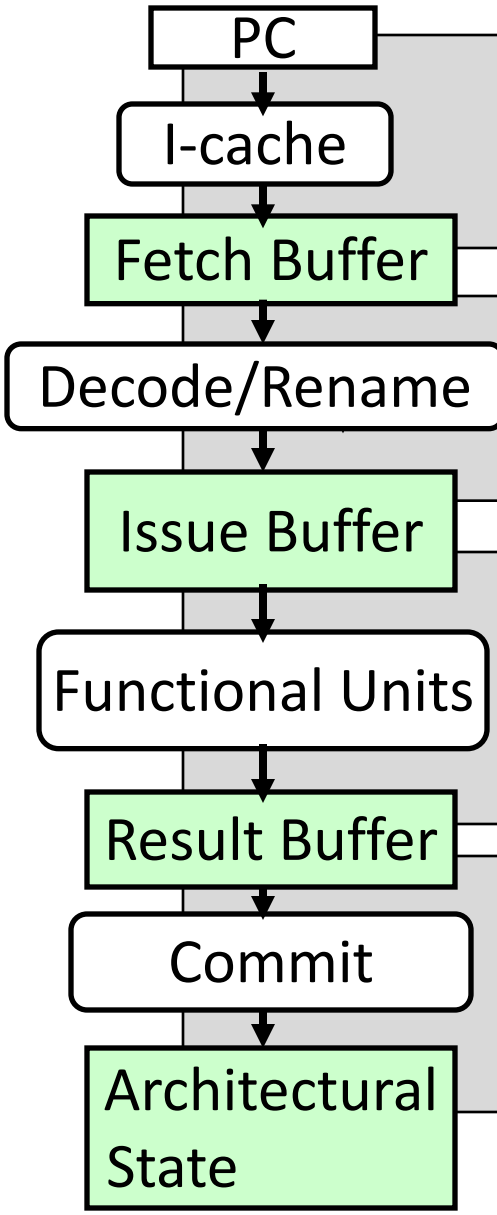


- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly

Superscalar Processor



Phases of Instruction Execution



Fetch: *Instruction bits retrieved from instruction cache.*

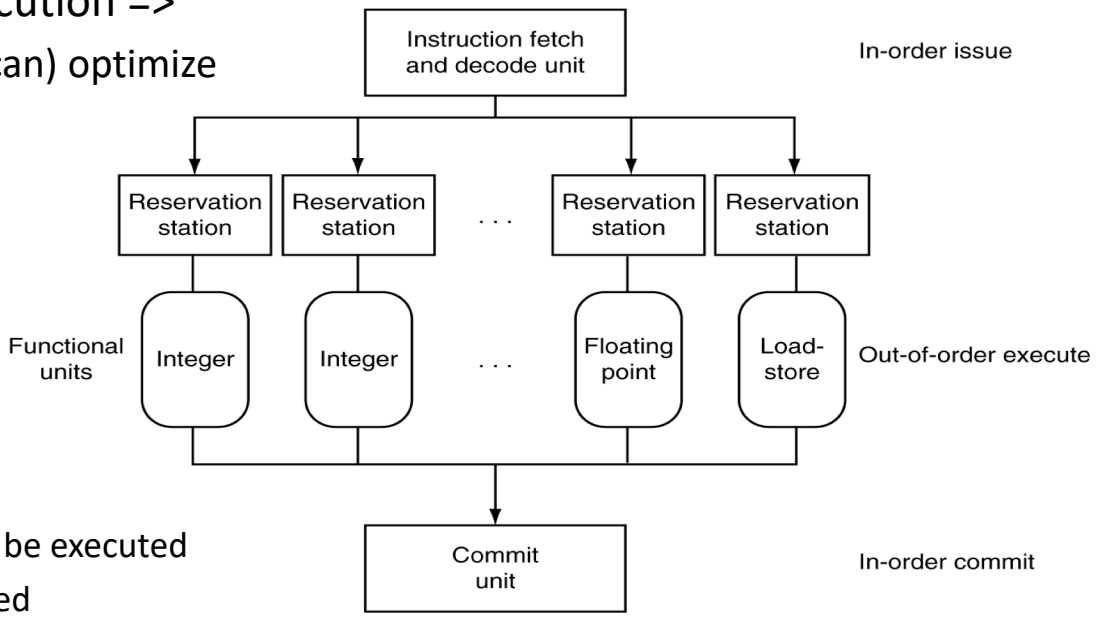
Decode: *Instructions dispatched to appropriate issue buffer*

Execute: *Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.*

Commit: *Instruction irrevocably updates architectural state (aka "graduation"), or takes precise trap/interrupt.*

Superscalar: Dynamic Multiple Issue

- Hardware guarantees correct execution =>
  - Compiler does not need to (but can) optimize



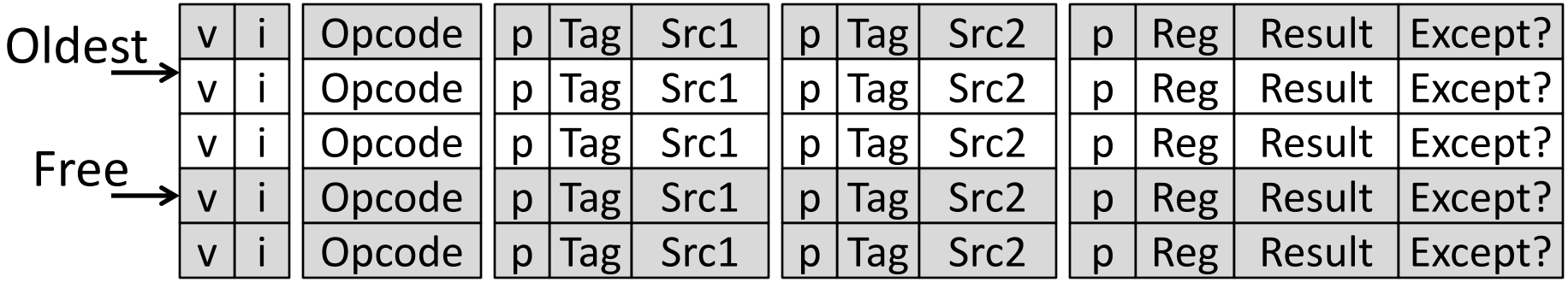
- Dynamic pipeline scheduling:
  - Instructions execute based on:
    - What functional units are free
    - Avoiding of data hazards
  - Reservation Station
    - Buffer of instructions waiting to be executed
    - With operands (Registers) needed
    - Once all operands are available: execute!
  - Commit Unit (Reorder buffer): supply the operands to reservation station; write to register; in original order
  - Unified Physical Register File :
    Registers are renamed for use in reservation station and commit unit

Separating Completion from Commit

- Re-order buffer holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)

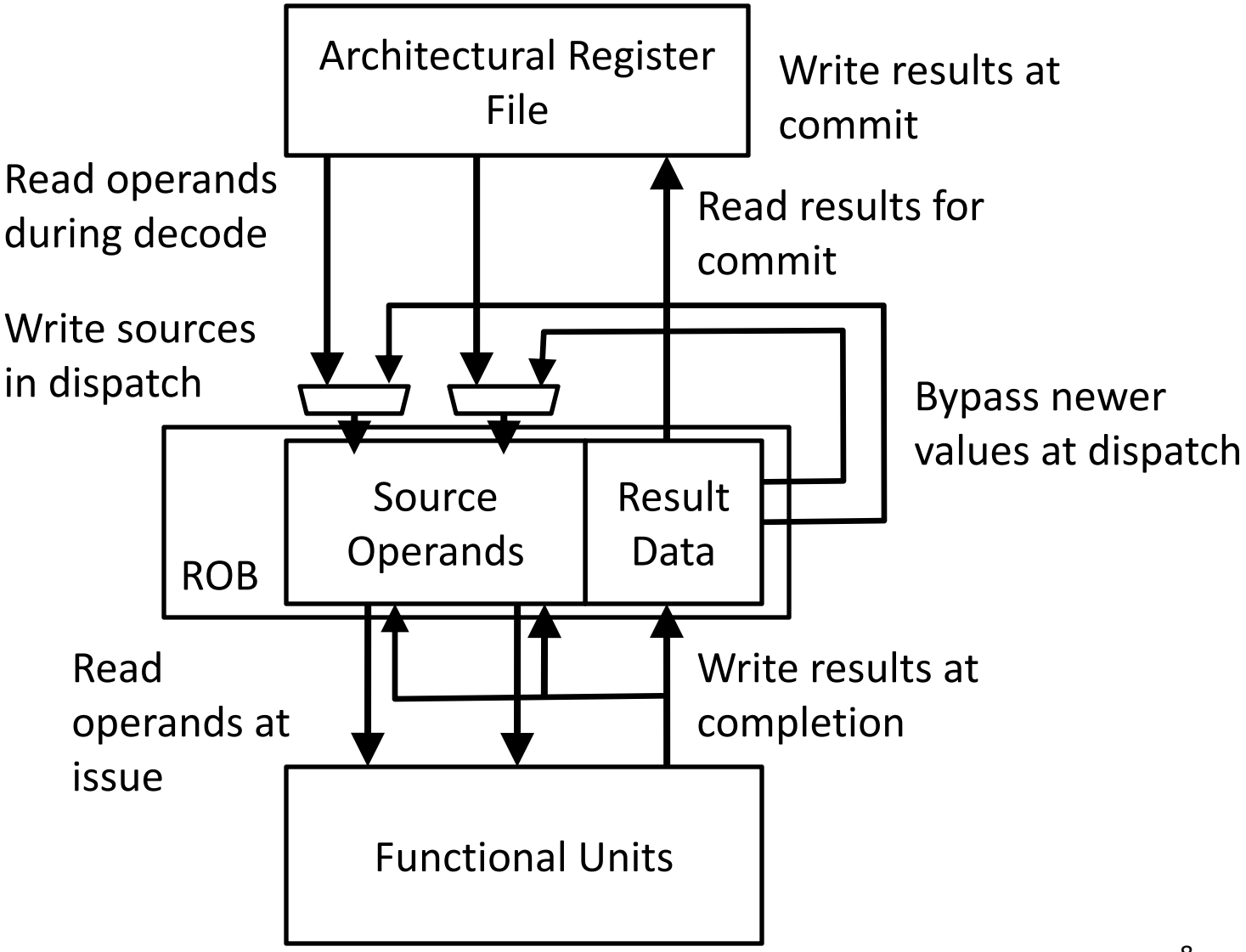
"Data-in-ROB" Design

(HP PA8000, Pentium Pro, Core2Duo, Nehalem)



- Managed as circular buffer in program order, new instructions dispatched to free slots, oldest instruction committed/reclaimed when done ("p" bit set on result)
- Tag is given by index in ROB (Free pointer value)
- In dispatch, non-busy source operands read from architectural register file and copied to Src1 and Src2 with presence bit "p" set. Busy operands copy tag of producer and clear "p" bit.
- Set valid bit "v" on dispatch, set issued bit "i" on issue
- On completion, search source tags, set "p" bit and copy data into src on tag match. Write result and exception flags to ROB.
- On commit, check exception status, and copy result into architectural register file if no trap.

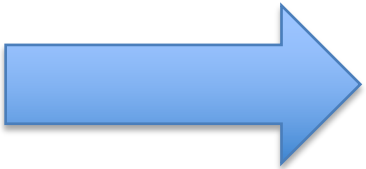
Data Movement in Data-in-ROB Design



Register Renaming

- Programmers/ Compilers (have to) re-use registers for different, unrelated purposes
- Idea: Re-name on the fly to resolve (fake) dependencies (anti-dependency)
- Additional benefit: CPU can have more physical registers than ISA!
  - Alpha 21264 CPU has 80 integer register; ISA only 32

```
1      r1 := m[1024]
2      r1 := r1 + 2
3      m[1032] := r1
4      r1 := m[2048]
5      r1 := r1 + 4
6      m[2056] := r1
```

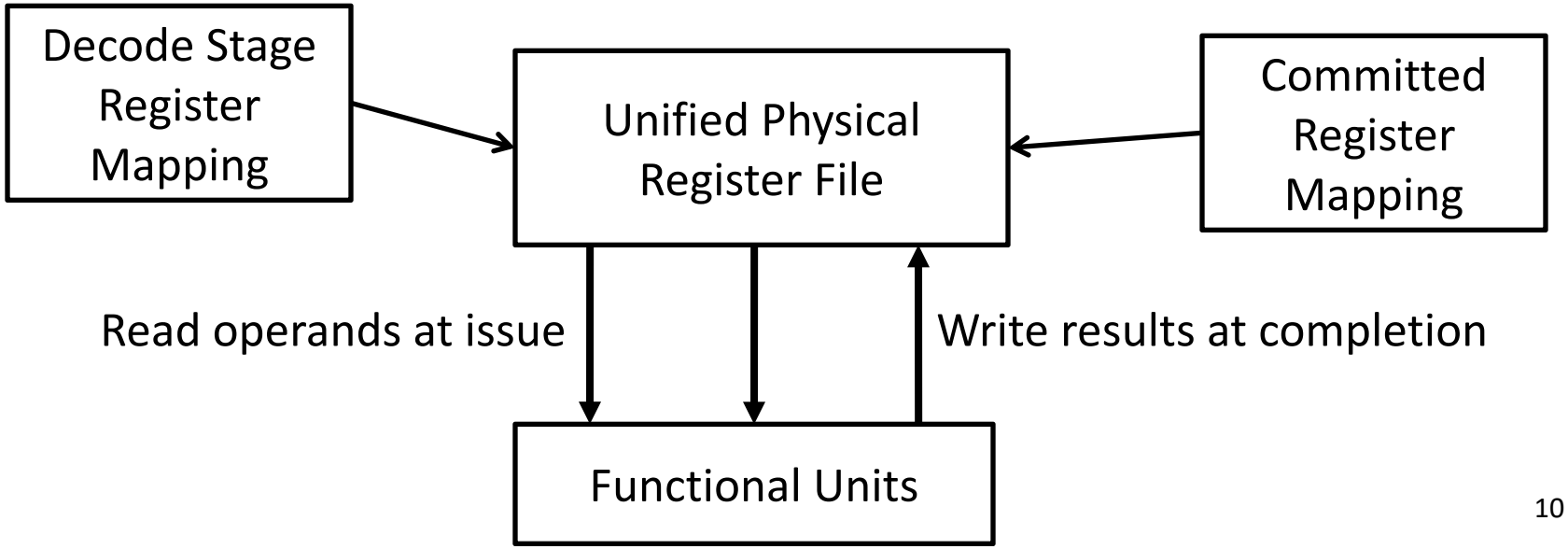


```
1      r1 := m[1024]
2      r1 := r1 + 2
3      m[1032] := r1
4      r2 := m[2048]
5      r2 := r2 + 4
6      m[2056] := r2
```

Alternative to ”Data-in-ROB”: Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement

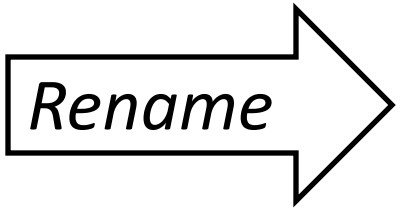


Renamed Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

Architectural Registers (from ISA)

```
ld x1, (x3)
addi x3, x1, 4
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x11)
```



Physical Registers (in CPU)

```
ld P1, (Px)
addi P2, P1, 4
sub P3, Py, Pz
add P4, P2, P3
ld P5, (P1)
add P6, P5, P4
sd P6, (P1)
ld P7, (Pw)
```

Conclusion Superscalar CPUs

- "Iron Law" of Processor Performance to estimate speed
- Complex Pipelines: **more in CA II**
  - Multiple Functional Units => Parallel execution
  - Static Multiple Issues (VLIW)
    - E.g. 2 instructions per cycle
  - Dynamic Multiple Issues (Superscalar)
    - Re-order instructions
    - Issue Buffer; Re-order Buffer; Commit Unit
    - Re-naming of registeres

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

Software | *Hardware*

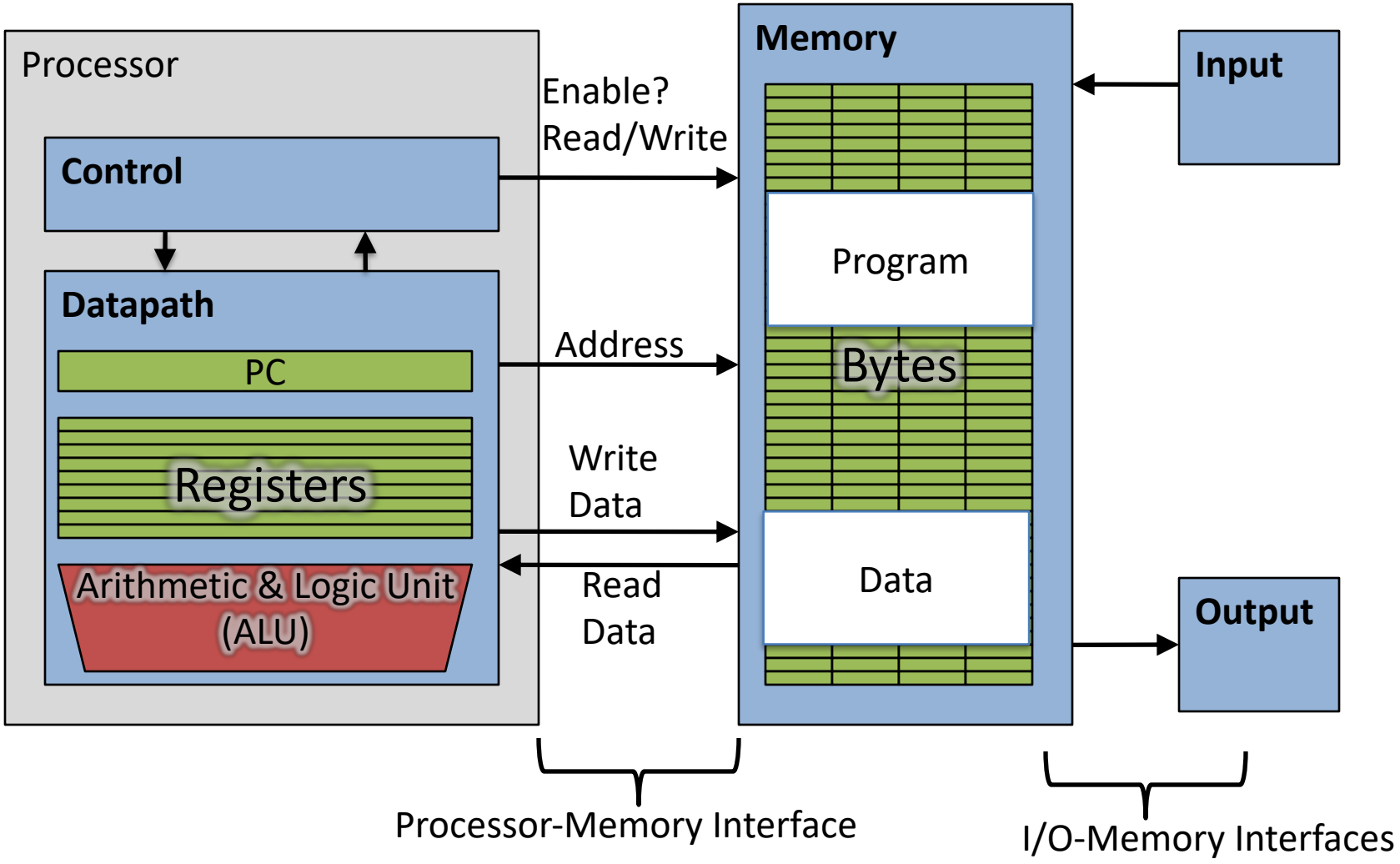
- Parallel Requests
  - Assigned to computer
  - e.g., Search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware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

Harness Parallelism & Achieve High Performance

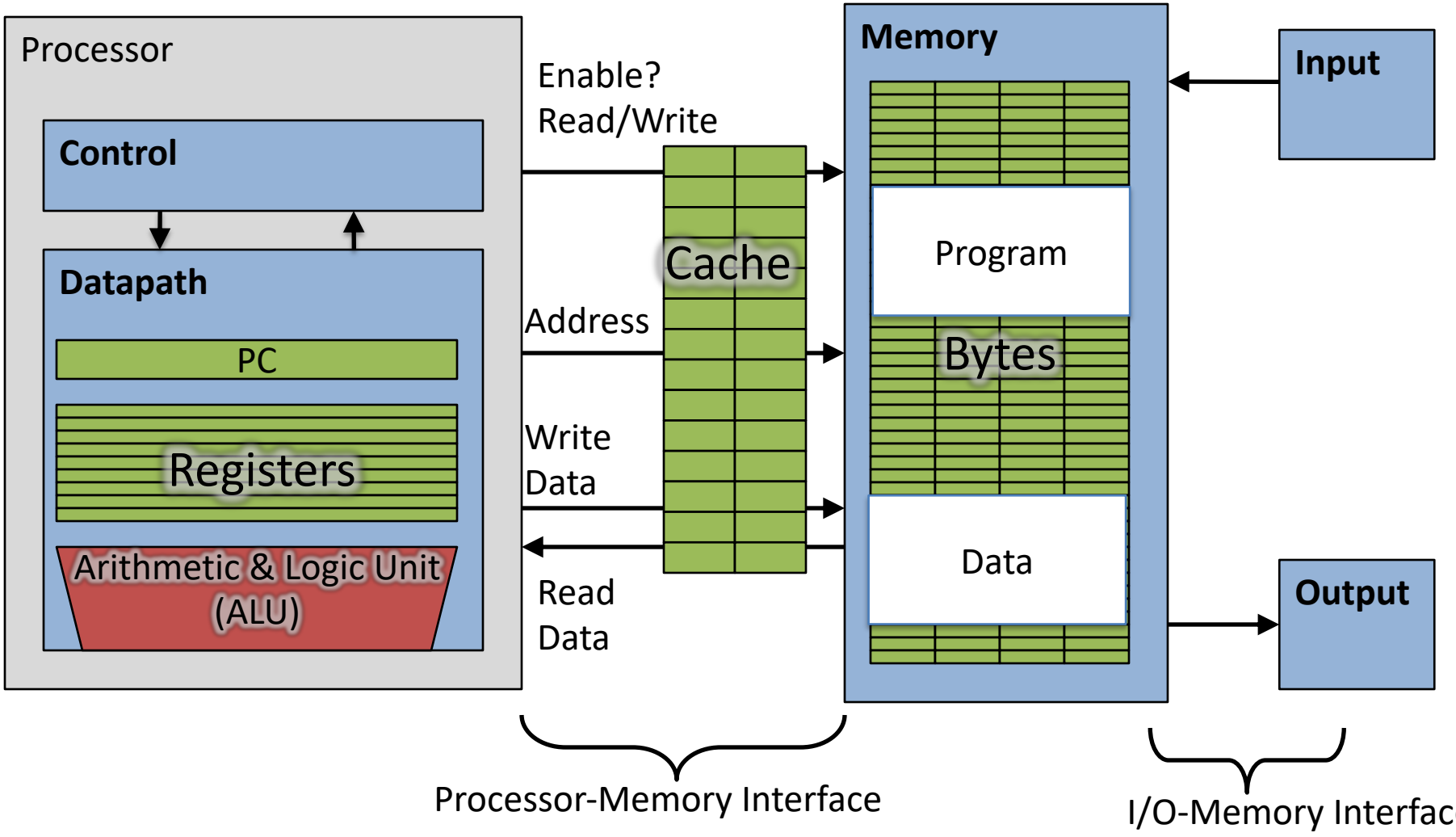
Warehouse Scale Computer

Smart Phone

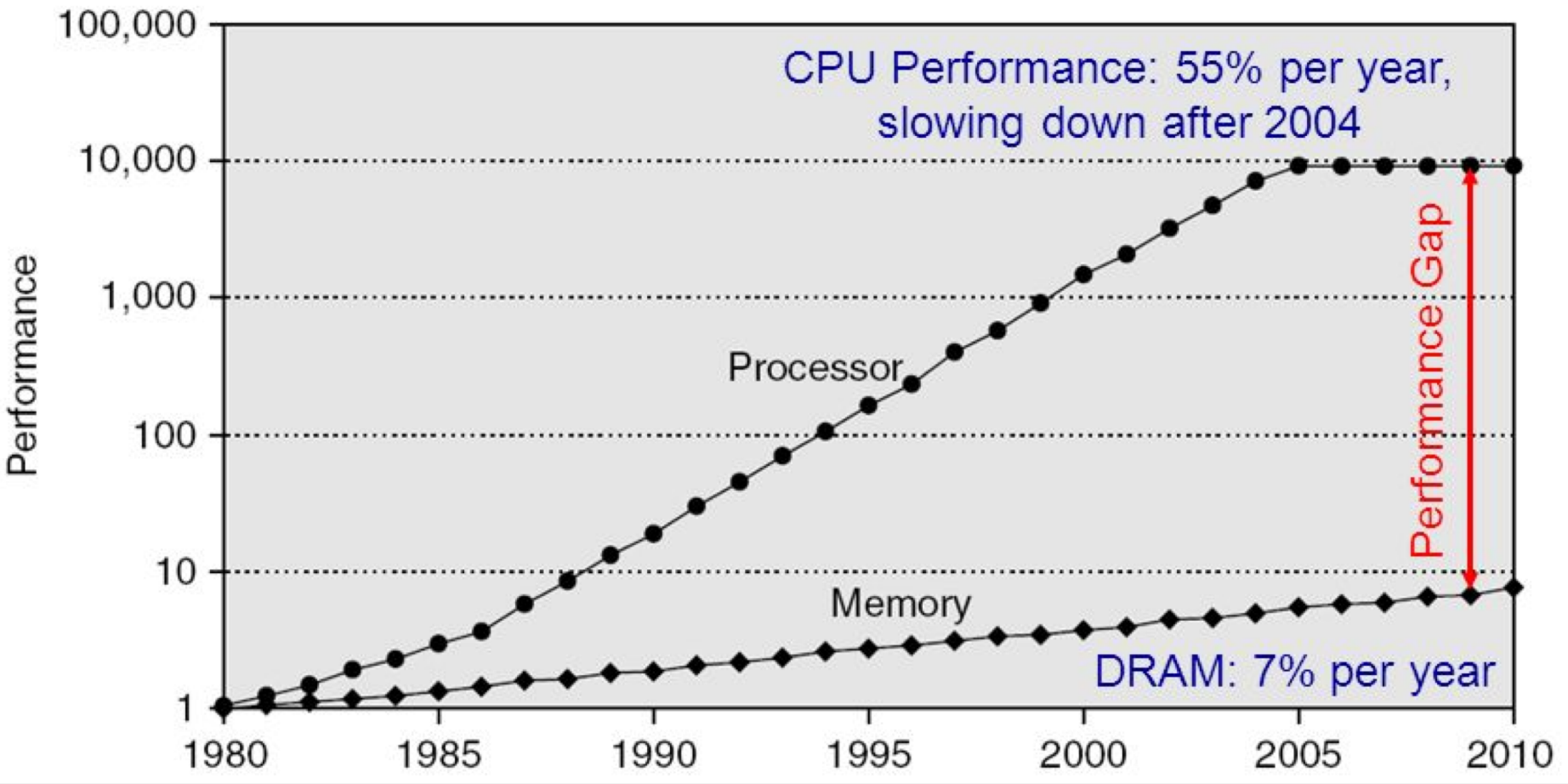
Components of a Computer



Adding Cache to Computer



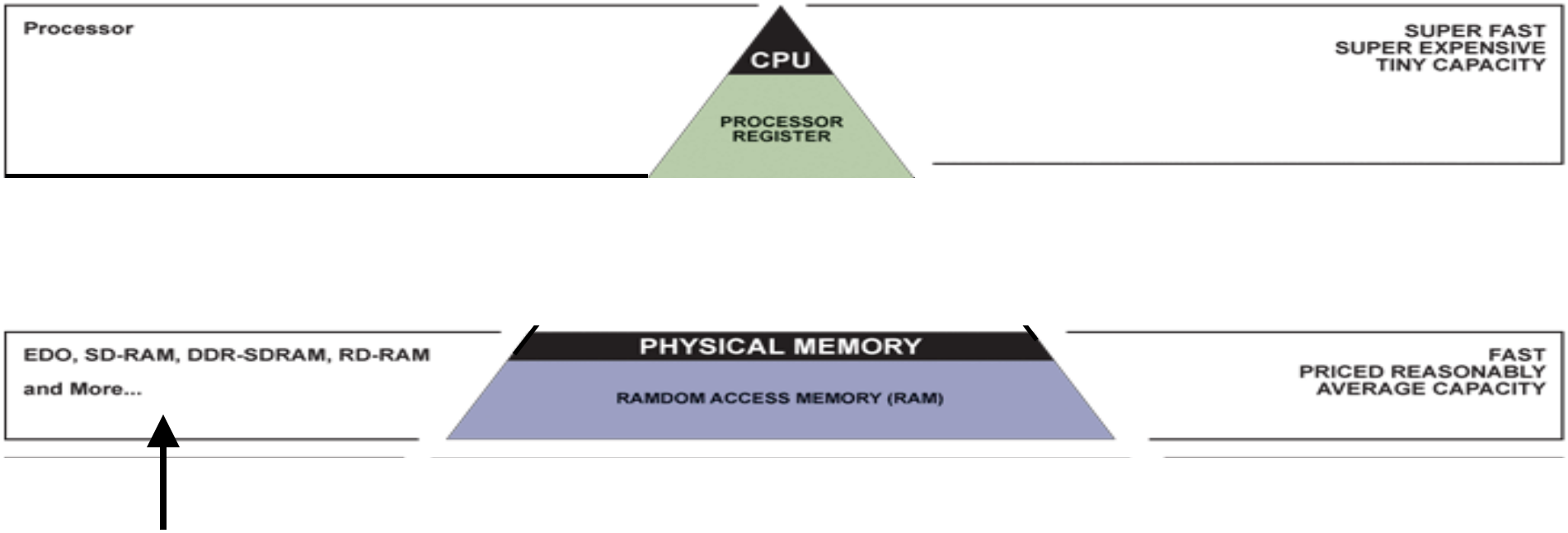
Processor-DRAM Gap (Latency)



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

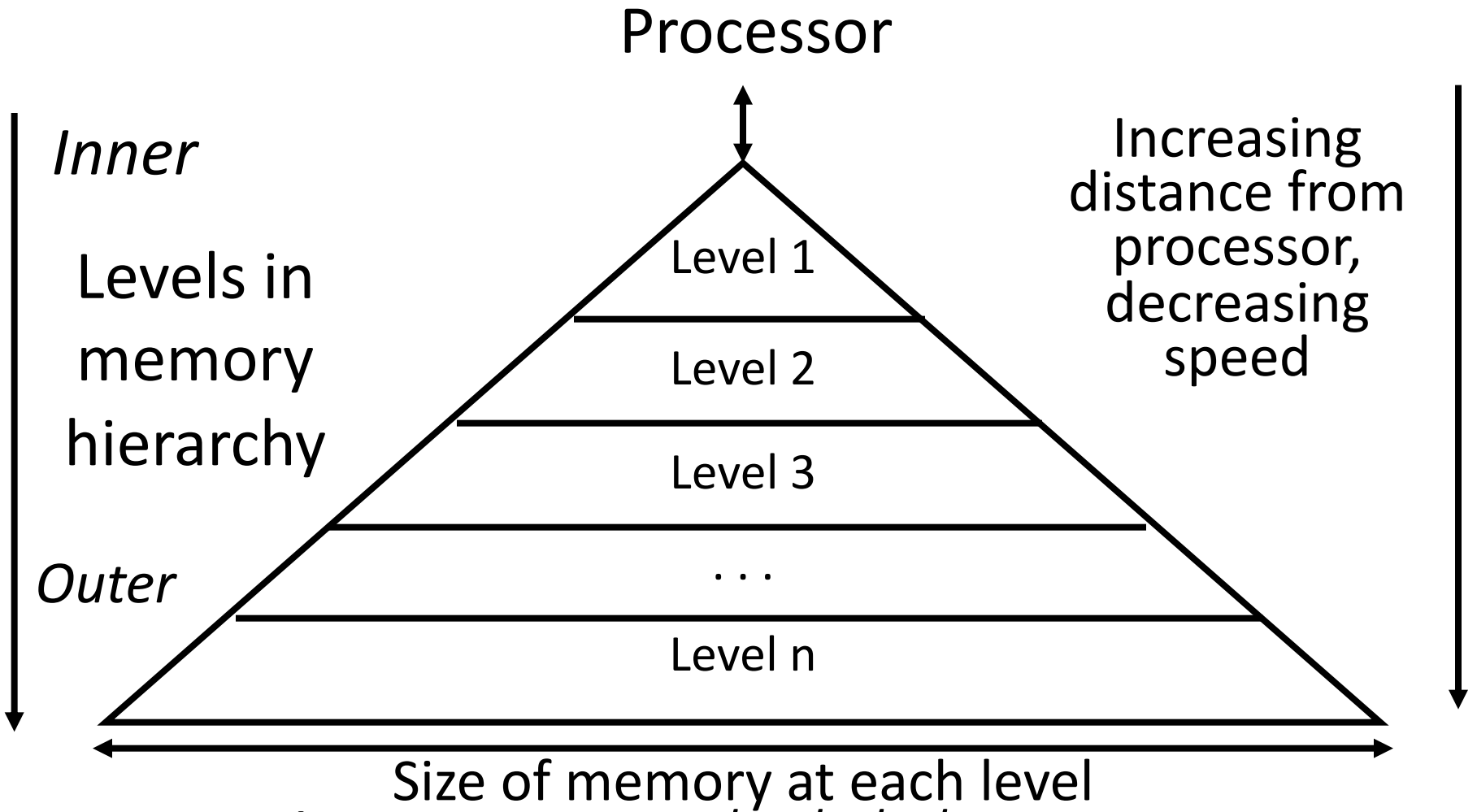
Slow DRAM access has disastrous impact on CPU performance!

Great Idea #3: Principle of Locality / Memory Hierarchy



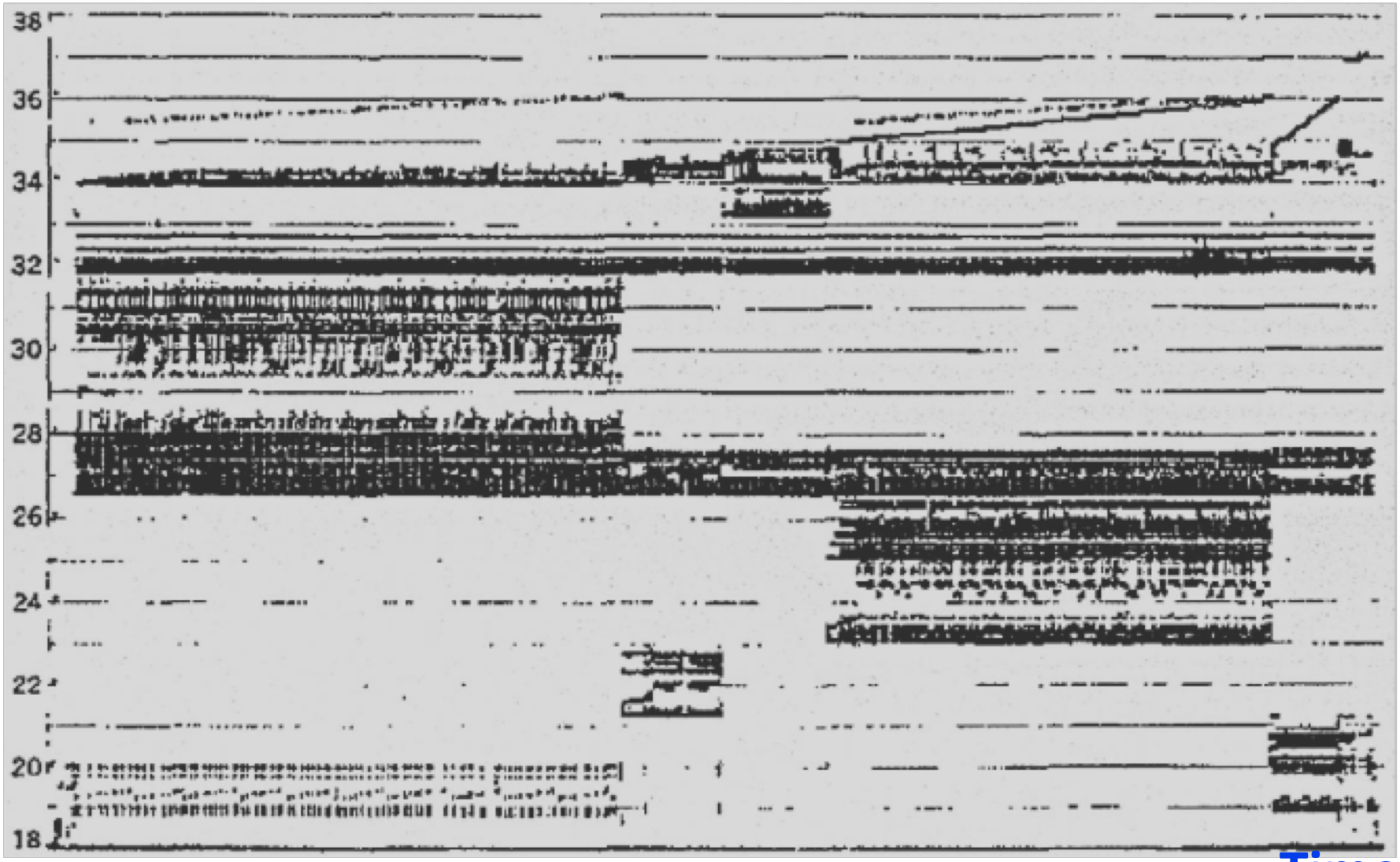
Note: These names are a bit dated

Big Idea: Memory Hierarchy



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

Real Memory Reference Patterns



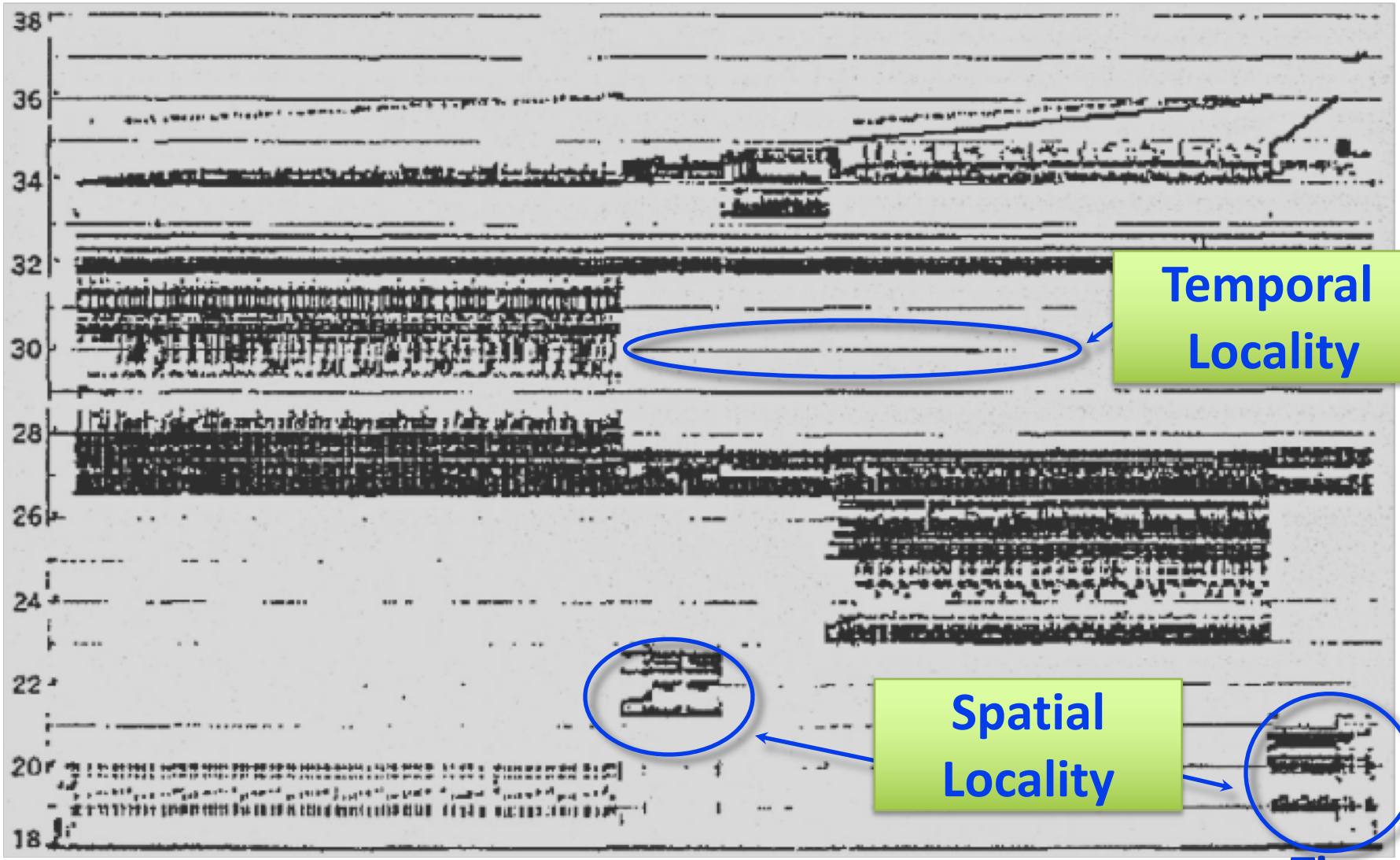
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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

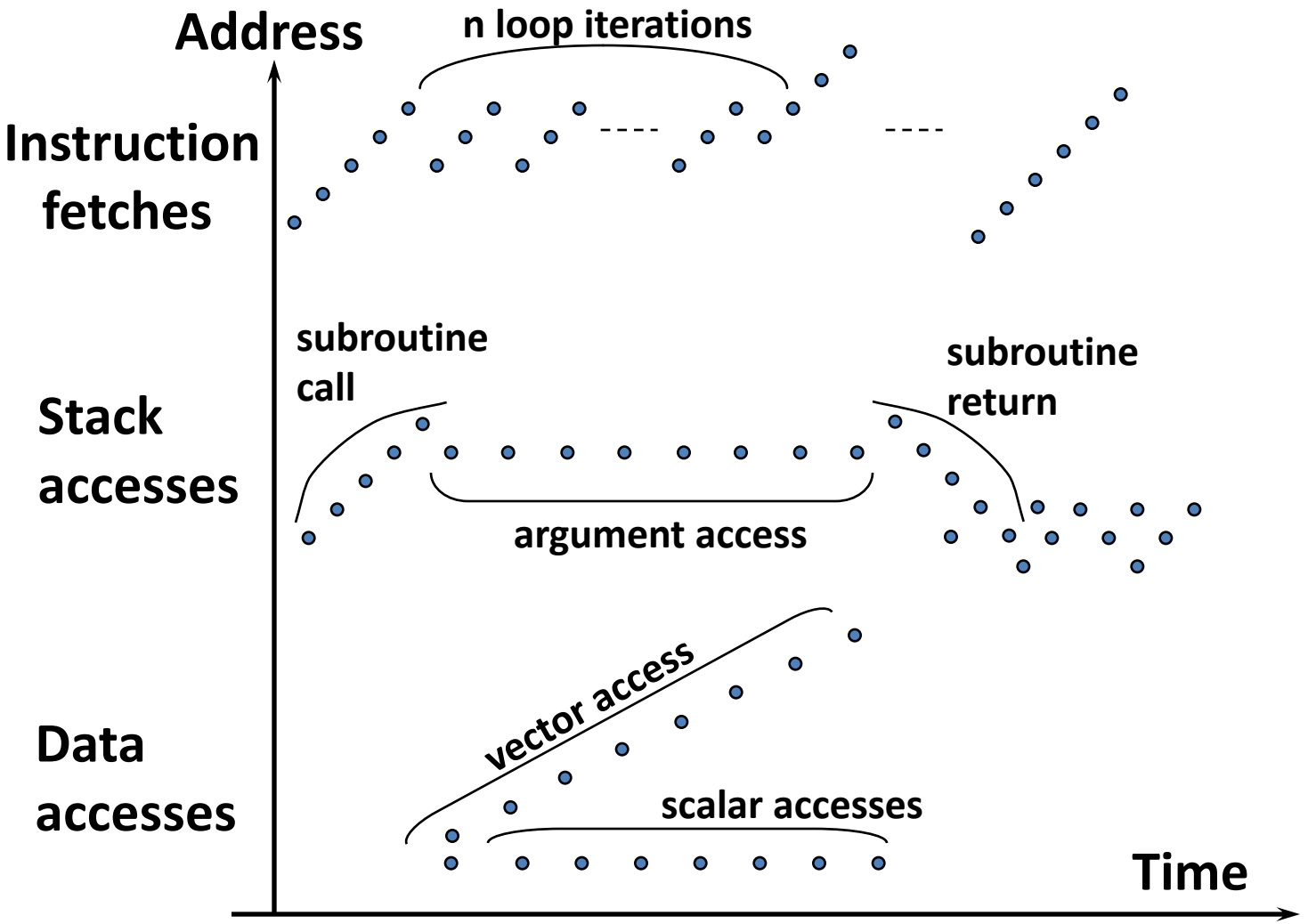


Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

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?

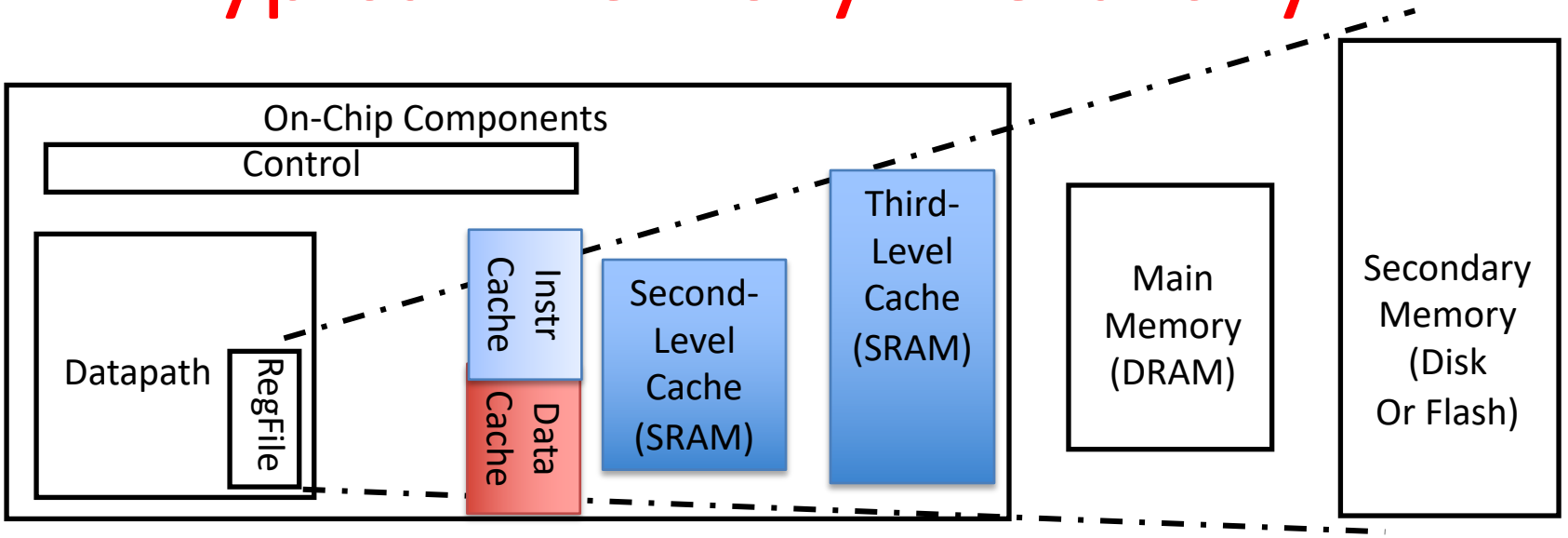
Memory Reference Patterns



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

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)

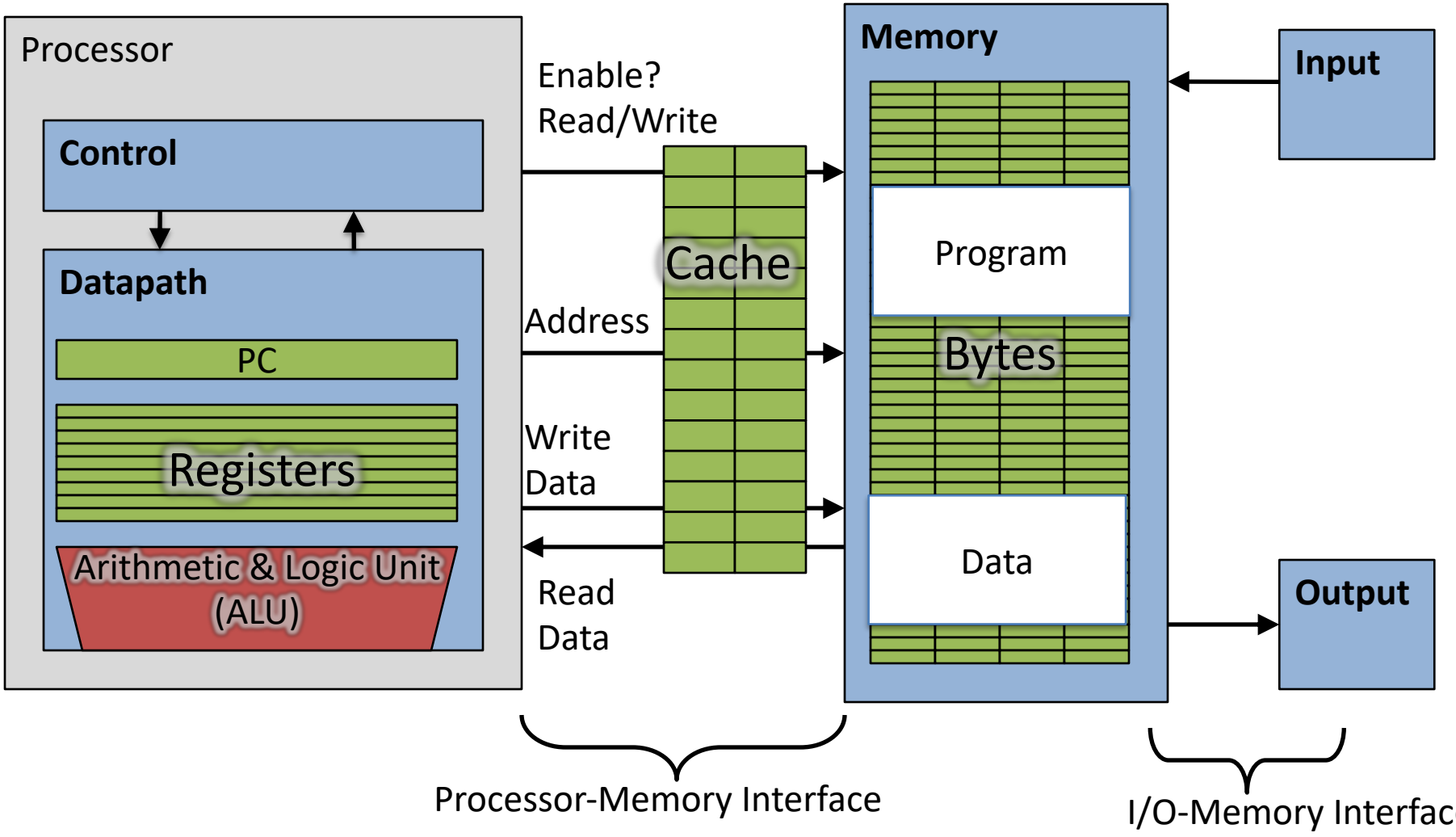
Also a type of cache

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

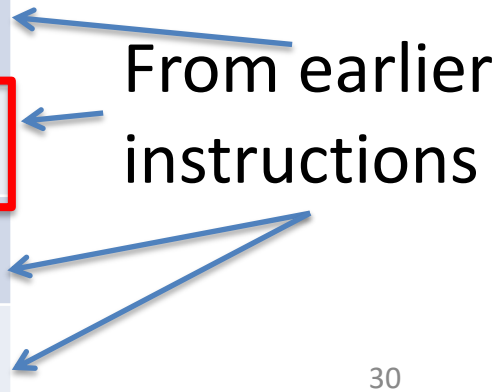
- Load word instruction: `lw t0,0(t1)`
- t1 contains $1022_{ten,}$ Memory[1022] = 99
- With cache: Processor issues address 1022_{ten} to Cache
  1. Cache checks to see if has copy of data at address 1022_{ten}
     - 2a. If finds a match (Hit): cache reads 99, sends to processor
     - 2b. No match (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

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

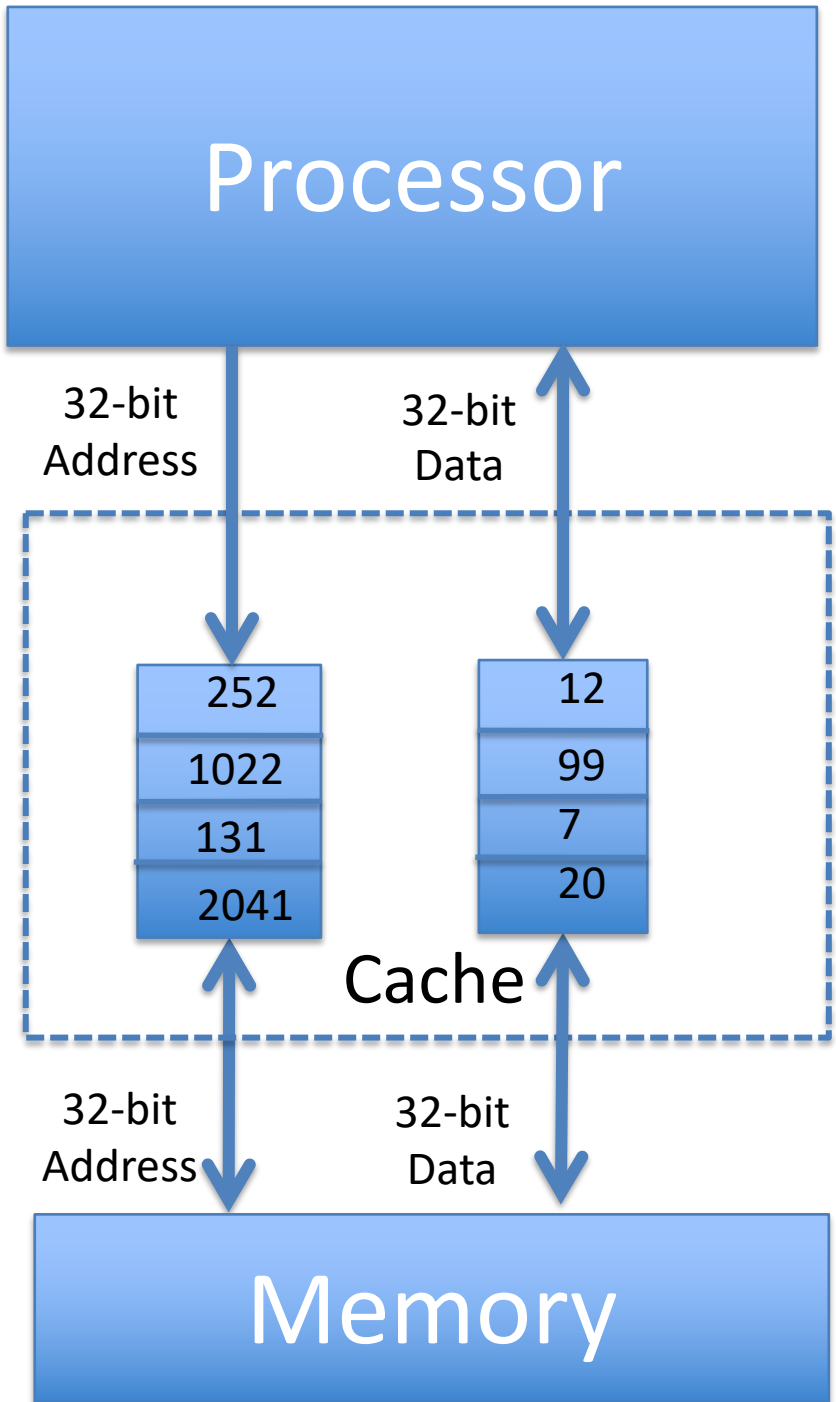
  1022 placed in tag next to data from memory (99)

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



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

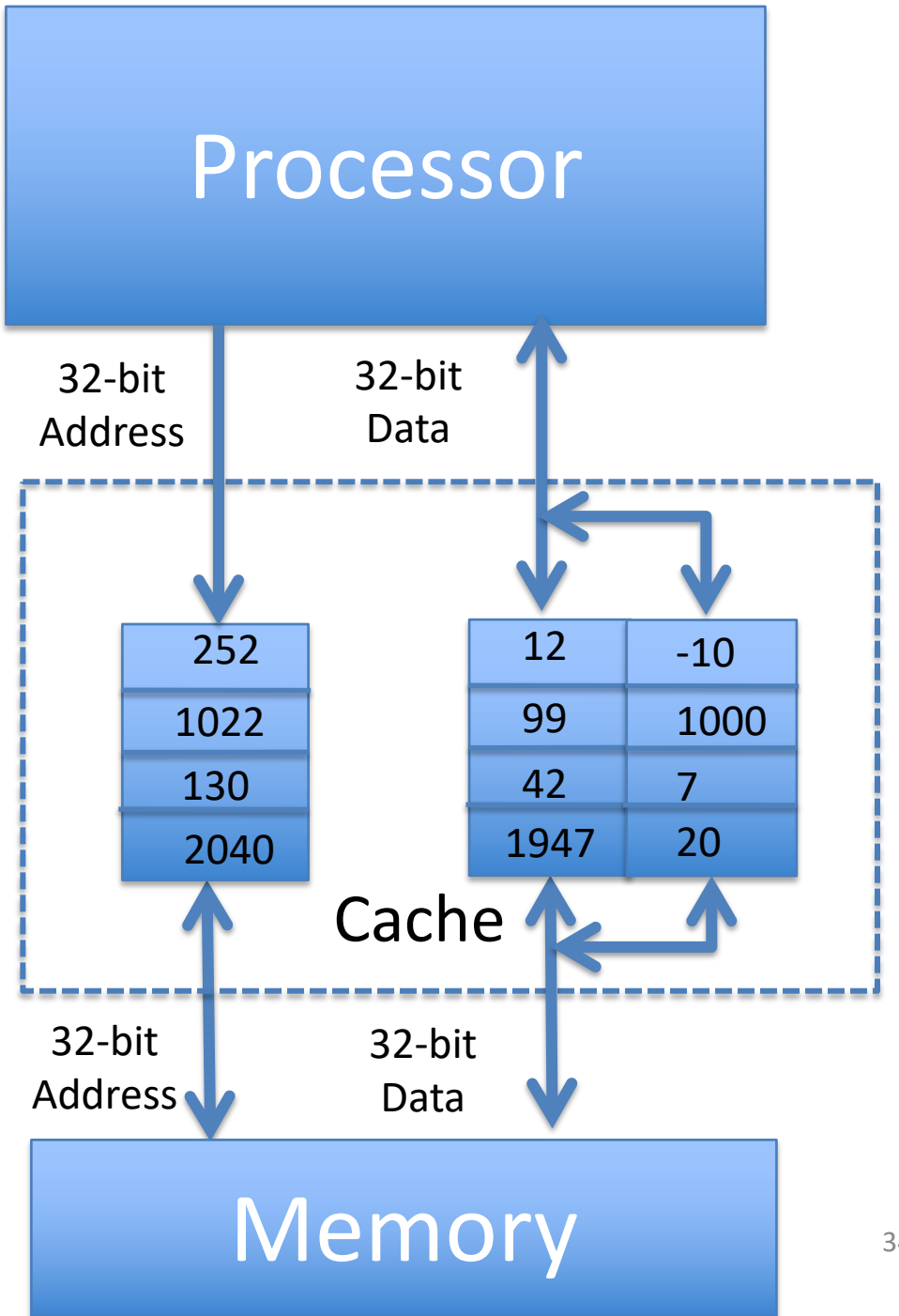
Block Must be Aligned in Memory

- 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)

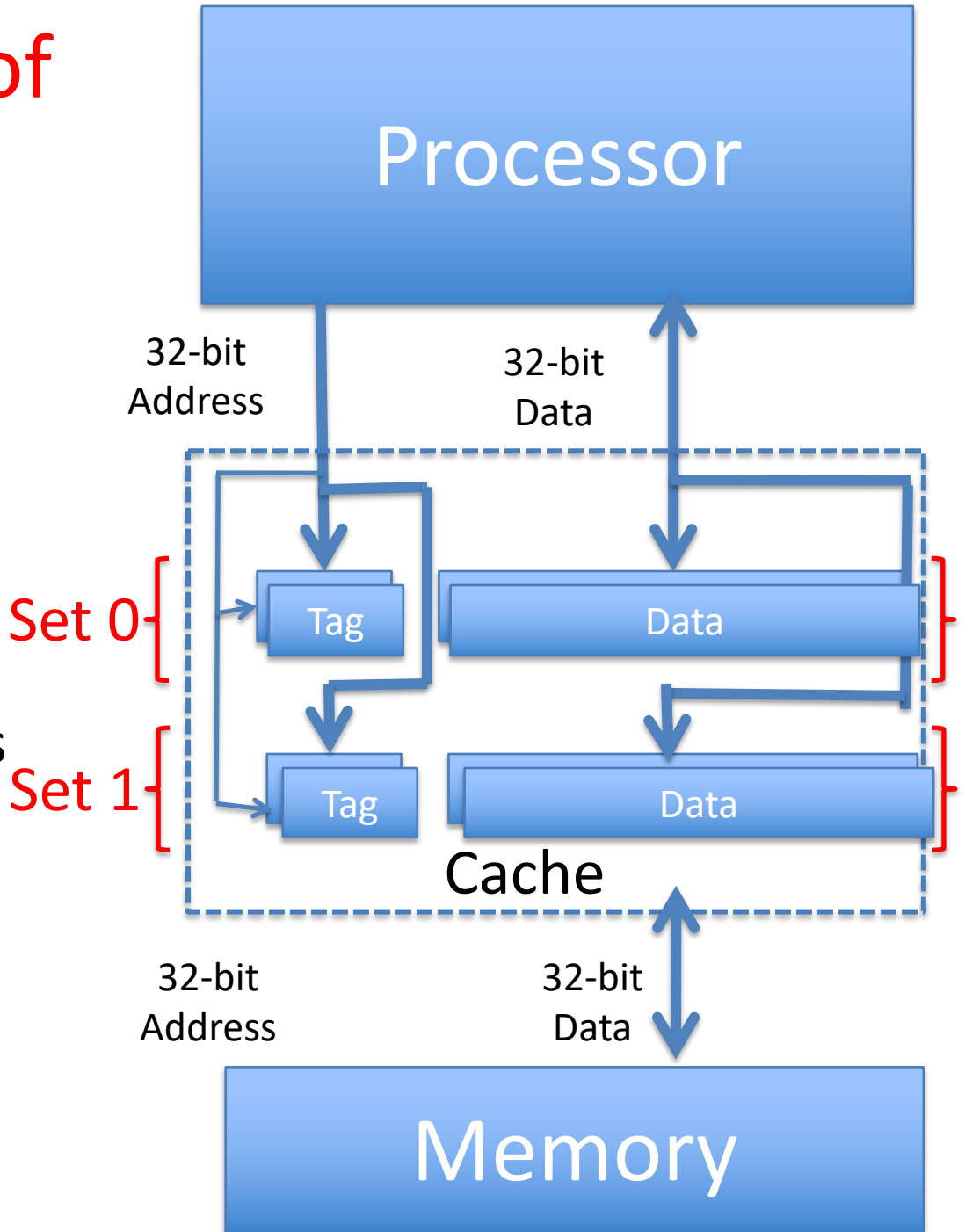
Anatomy of a 32B Cache, 8B Block

- 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



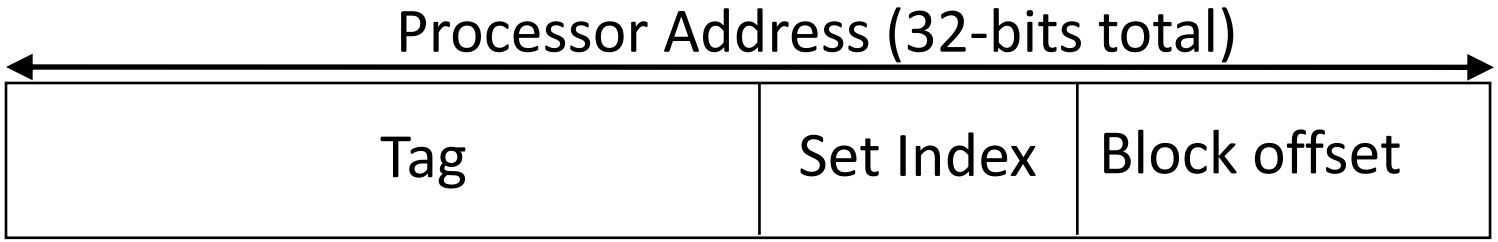
Hardware Cost of Cache

- Need to compare every tag to the Processor address
- Comparators are expensive
- Optimization: use 2 "sets" => ½ comparators
- 1 Address bit selects which set
- Compare only tags from selected set
- Generalize to more sets



Processor Address Fields used by Cache Controller

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

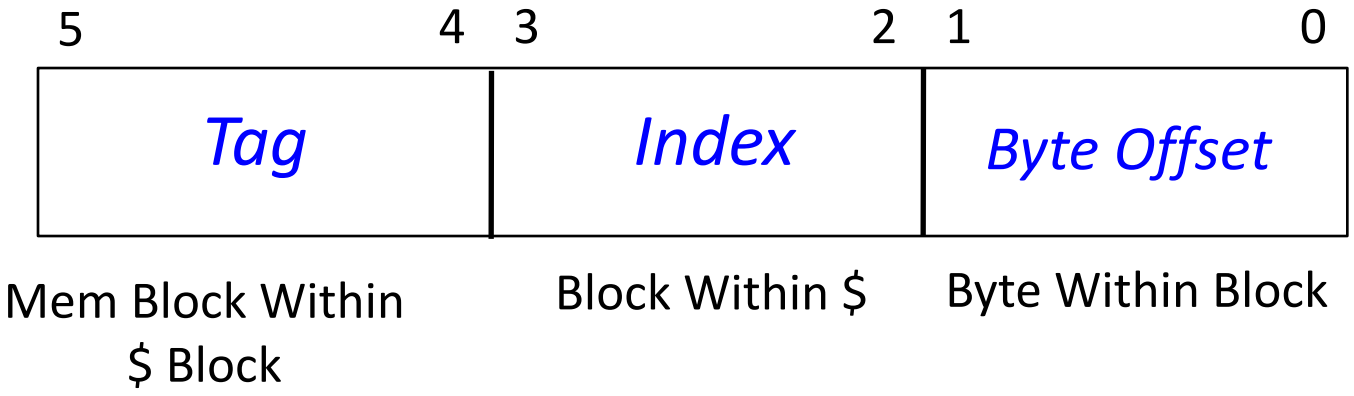


- Size of Index = log2 (number of sets)
- Size of Tag = Address size – Size of Index – log2 (number of bytes/block)

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

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

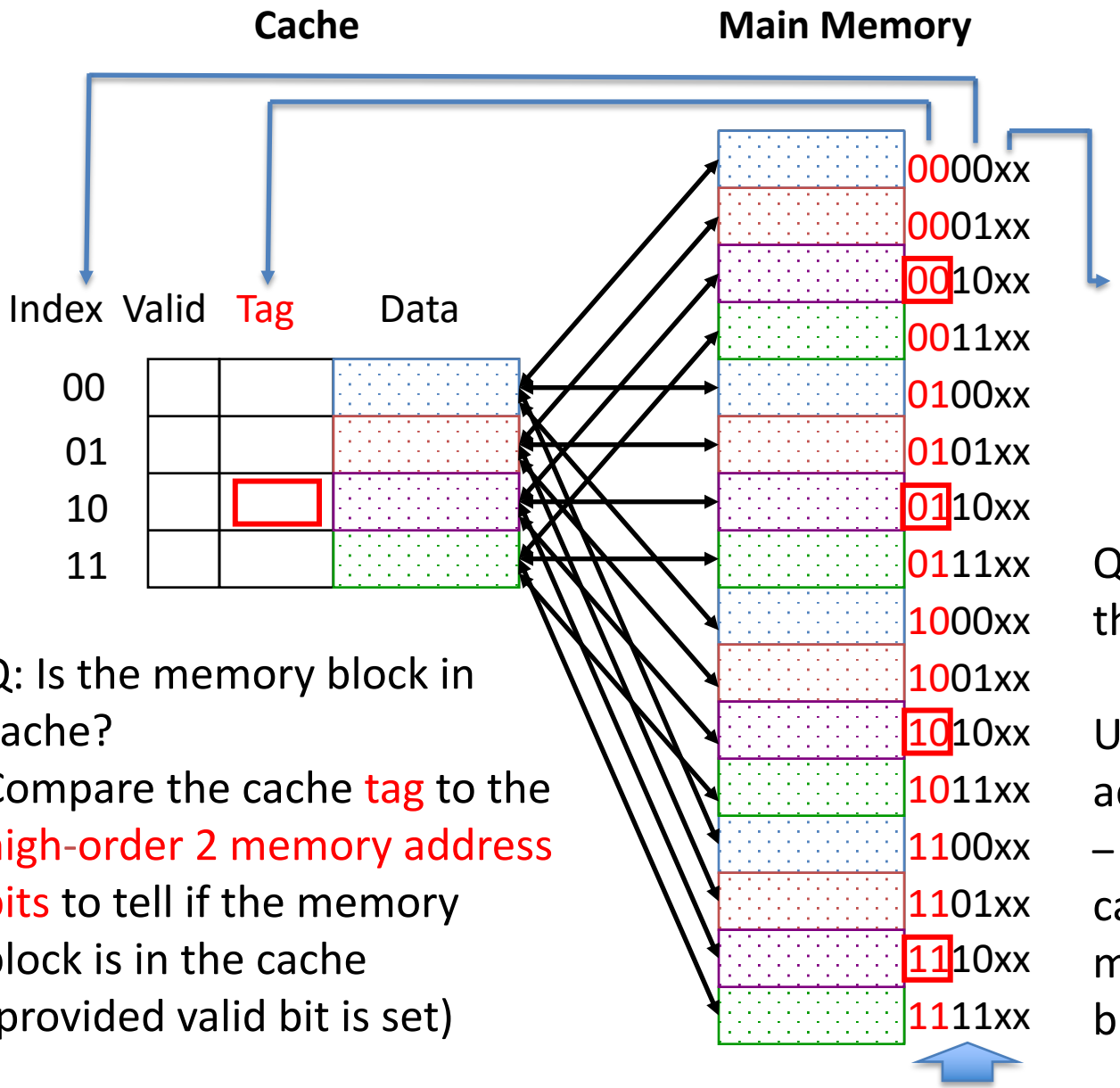
One More Detail: Valid Bit

- 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)

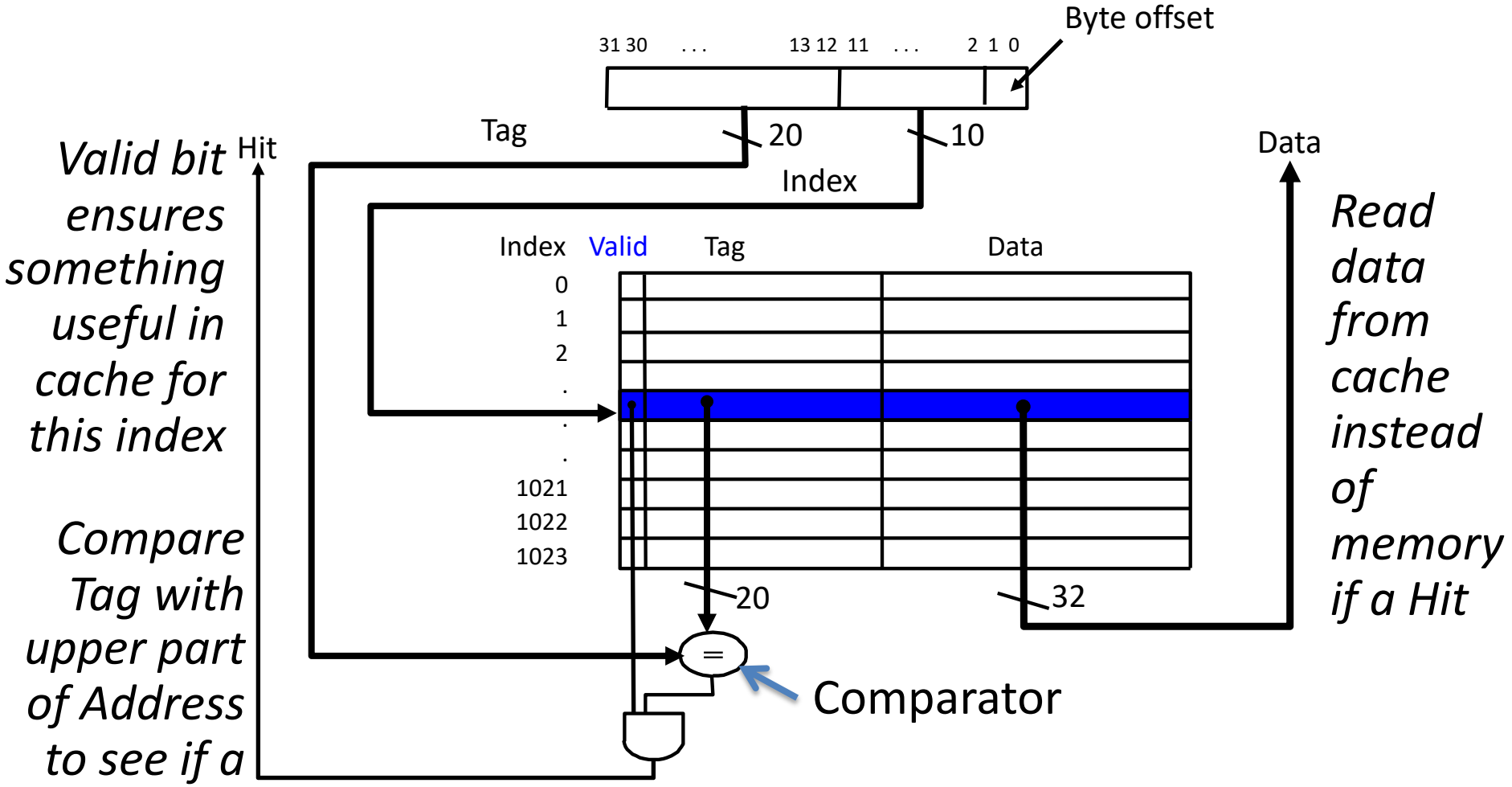
Q: Is the memory block in cache?
Compare the cache tag to the high-order 2 memory address bits to tell if the memory block is in the cache (provided valid bit is set)

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)

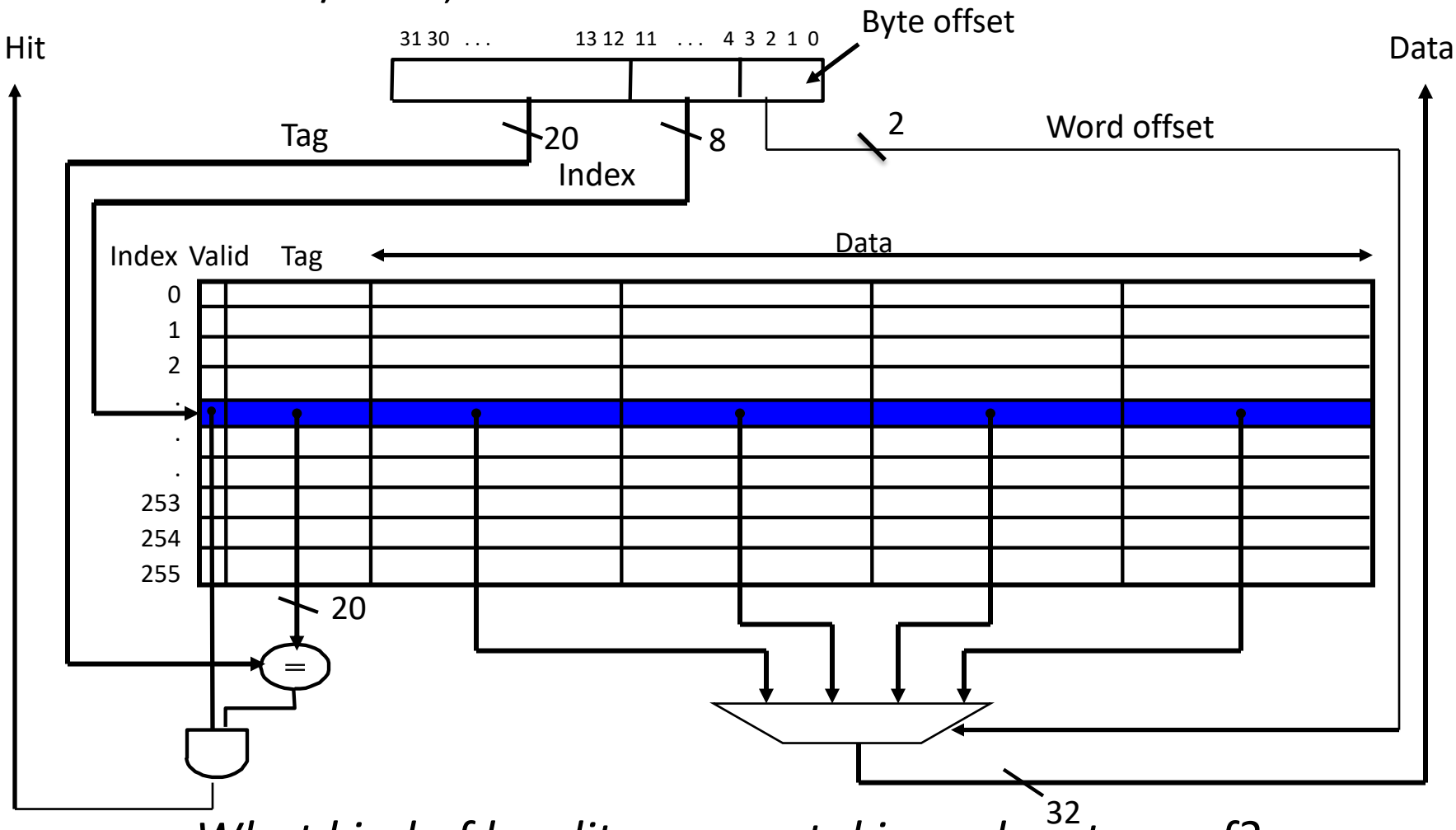
Direct-Mapped Cache Example

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



Multiword-Block Direct-Mapped Cache

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



What kind of locality are we taking advantage of?