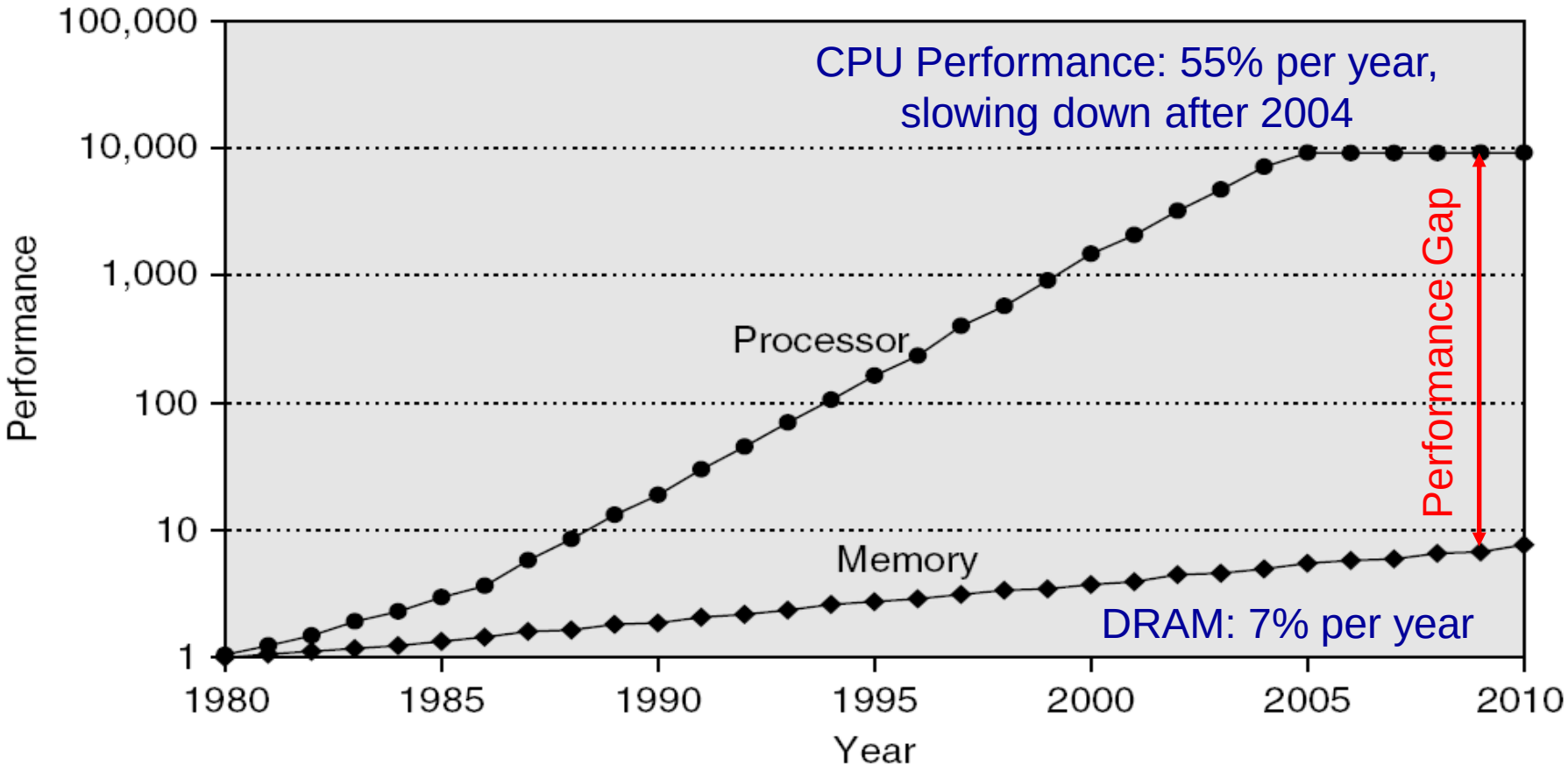
# Memory Hierarchy Design – Cache Memory

# Presentation Outline

- Memory Hierarchy and the need for Cache Memory
- The Basics of Caches
- Multilevel Caches
- Cache Performance and Memory Stall Cycles

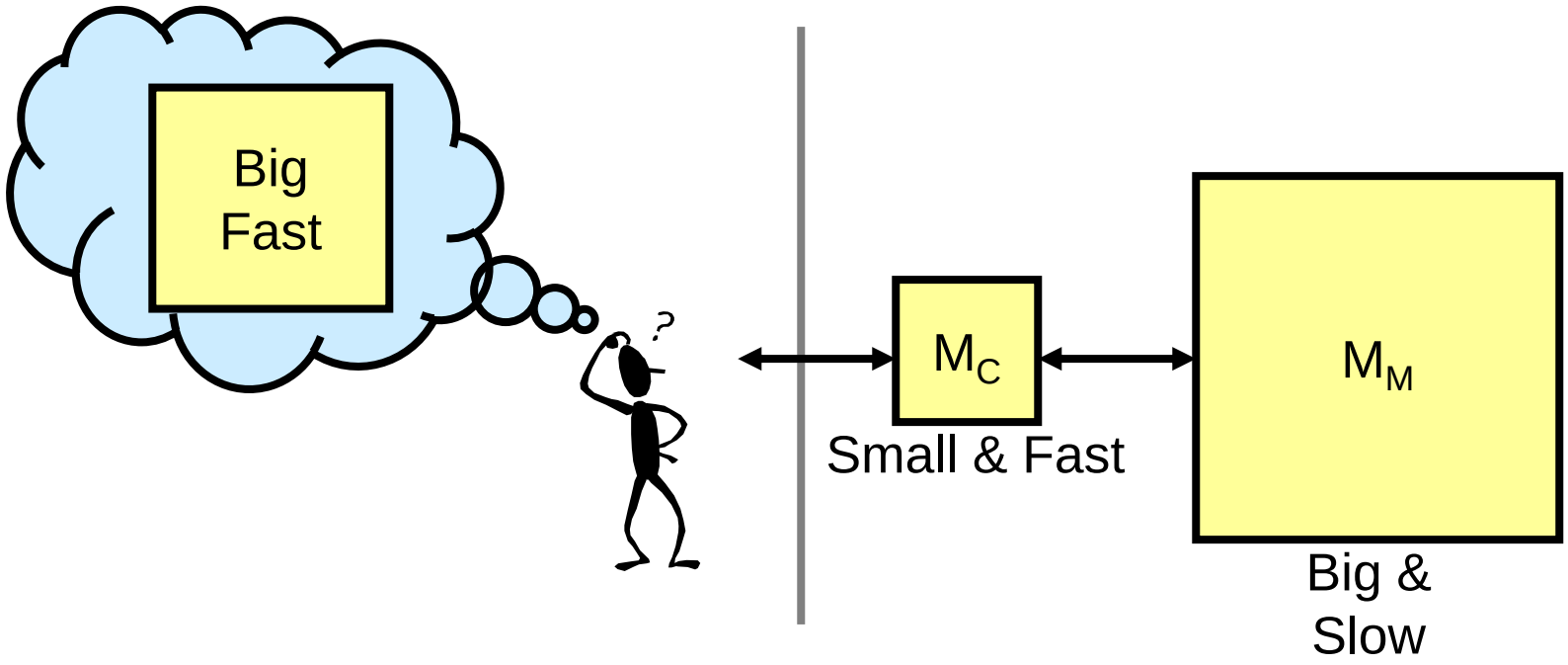
# Processor-Memory Performance Gap




- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

# Motivation for Memory Hierarchy

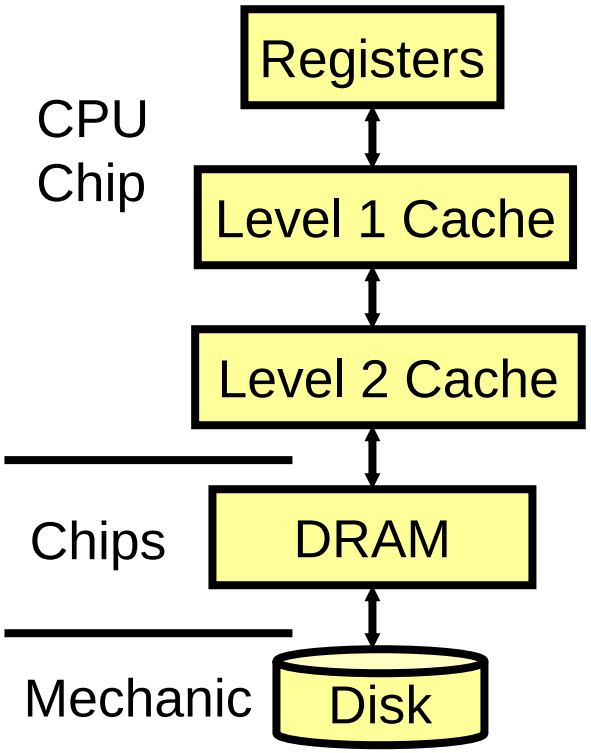
- Programmers want unlimited amounts of memory with low latency
  - But fast memory more expensive than slower memory
- Solution: small fast memory + big slow memory
  = Looks like a big fast memory



# The Need for Cache Memory

- Widening speed gap between CPU and main memory
  - Processor operation takes less than 1 ns
  - Main memory requires more than 50 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - A second memory access for load and store instructions
- Memory bandwidth limits the instruction execution rate
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast

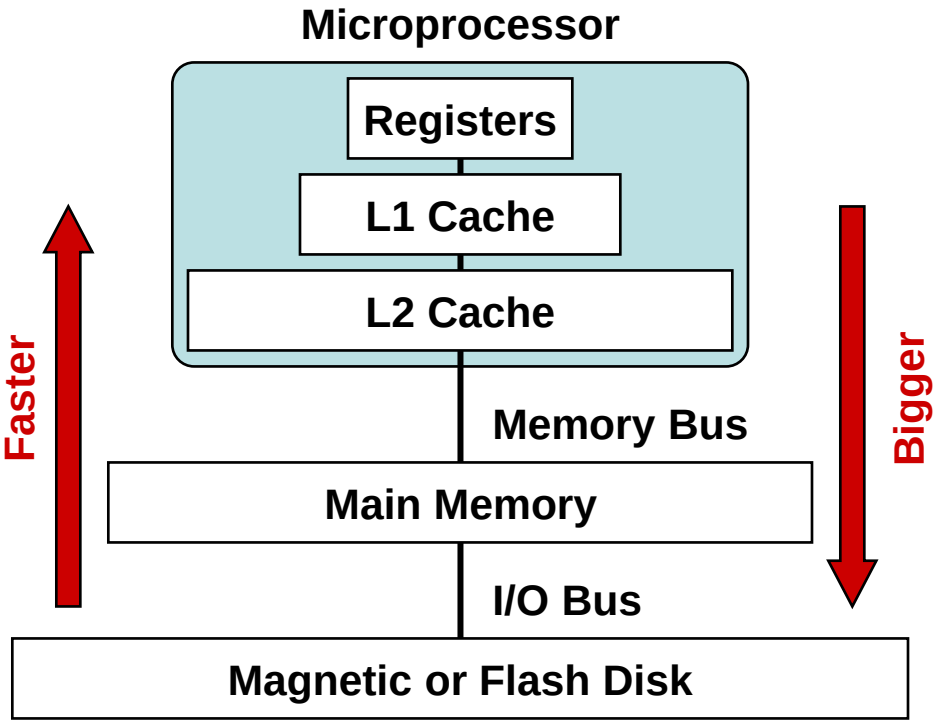
# Model of Memory Hierarchy



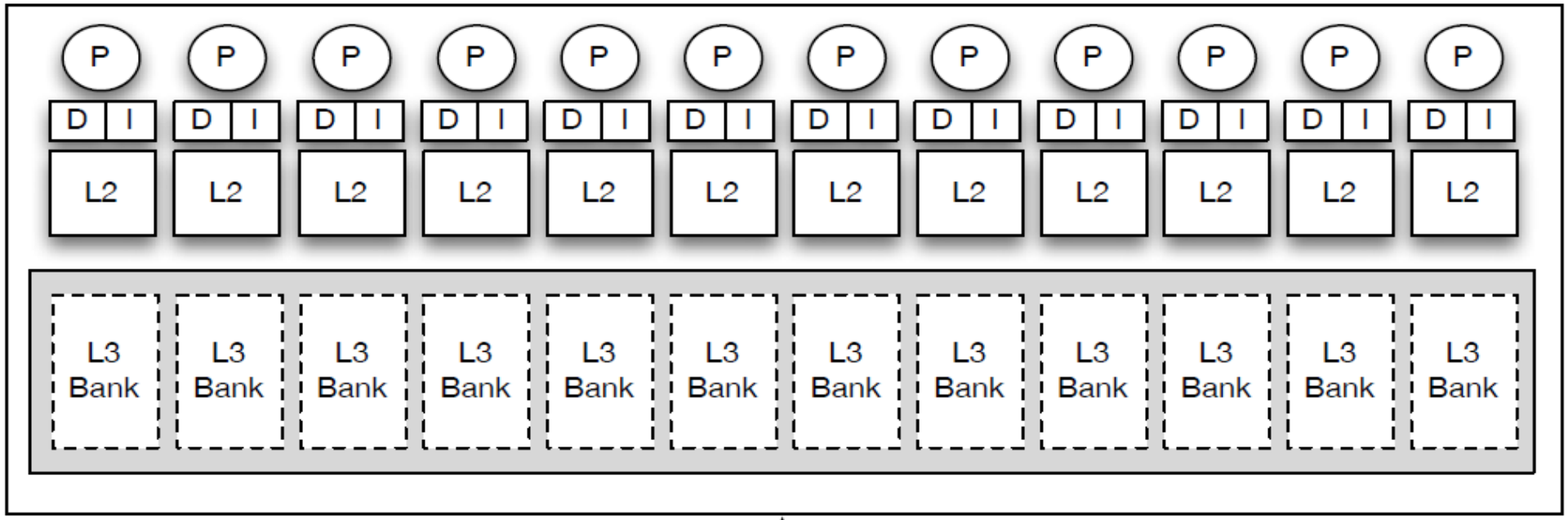

| | Cycle | Words/cycle | Management |
|---|---|---|---|
| Registers | 1 | 3-10 | Compiler |
| Level 1 Cache | 1-3 | 1-2 | Hardware |
| Level 2 Cache | 5-10 | 1 | Hardware |
| DRAM | 30-100 | 0.5 | OS |
| Disk | $10^6$-$10^7$ | 0.01 | OS |
| | | | ? |

# Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KiB)
  - Access time: 1 ns
- L2 Cache (1 MiB – 8 MiB)
  - Access time: 3 – 10 ns
- Main Memory (8 – 32 GiB)
  - Access time: 40 – 50 ns
- Disk Storage (> 200 GB)
  - Access time: 5 – 10 ms

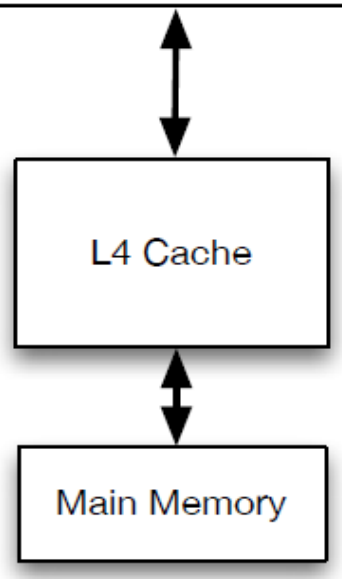


# Typical Memory Hierarchy




L1 Cache: 32 – 64KB (SRAM)
Instruction / Data Cache
1 ns (2-3 cycles) access time

L2 Cache: 256KB – 1MB (SRAM)
3-10 ns (6-20 cycles) access time




L3 Cache: 8MB – 96MB (eDRAM)
10-20 ns access time

Off-chip L4 Cache (larger than L3)
20-40 ns access time

8GB – 256GB Main Memory
DRAM, 50-100 ns access time

# Principle of Locality of Reference

- Programs access small portion of their address space
  - At any time, only a small set of instructions & data is needed
- **Temporal Locality** (in time)
  - If an item is accessed, probably it will be accessed again soon
  - Same loop instructions are fetched each iteration
  - Same procedure may be called and executed many times
- **Spatial Locality** (in space)
  - Tendency to access contiguous instructions/data in memory
  - Sequential execution of Instructions
  - Traversing arrays element by element

# Memory Hierarchy Operation

- If an instruction or operand is required by the CPU, the levels of the memory hierarchy are searched for the item starting with the level closest to the CPU (Level 1 cache): **$L_1$ Cache**
  - If the item is found, it's delivered to the CPU resulting in a cache hit without searching lower levels. **Hit rate for level one cache = $H_1$**
  - If the item is missing from an upper level, resulting in a cache miss, the level just below is searched. **Miss rate for level one cache = 1 – Hit rate = 1 - $H_1$**
  - For systems with several levels of cache, the search continues with cache level 2, 3 etc.
  - If all levels of cache report a miss then main memory is accessed for the item.
    - CPU ↔ cache ↔ memory: Managed by hardware.
  - If the item is not found in main memory resulting in a page fault, then disk (virtual memory), is accessed for the item.
    - Memory ↔ disk: Managed by the operating system with hardware support

# Memory Hierarchy: Terminology

- A Block: The smallest unit of information transferred between two levels.
- Hit: Item is found 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**
    **(S)** RAM access time + Time to determine hit/miss

**Ideally = 1 Cycle**

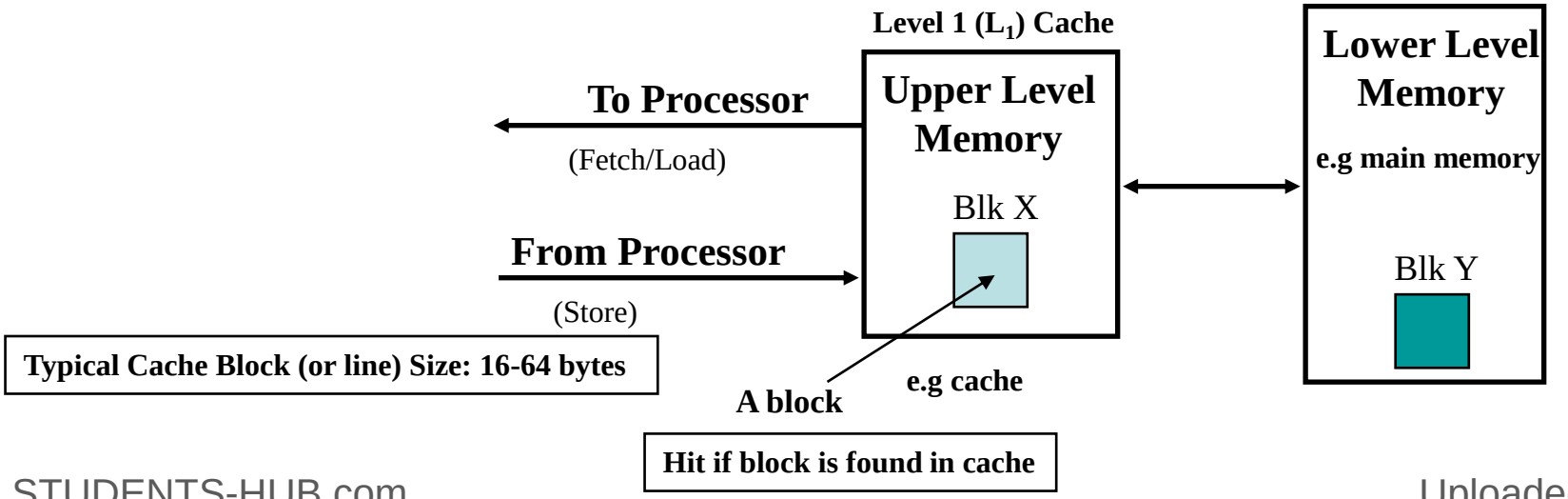
- Miss: Item needs to be retrieved from a block in the lower level (Block Y)
  - **Miss Rate = 1 - (Hit Rate)**

**Miss rate for level one cache = 1 – Hit rate = 1 - $H_1$**

  - **Miss Penalty: Time to replace a block in the upper level + M**

    Time to deliver the missed block to the processor

- Hit Time << Miss Penalty **M**



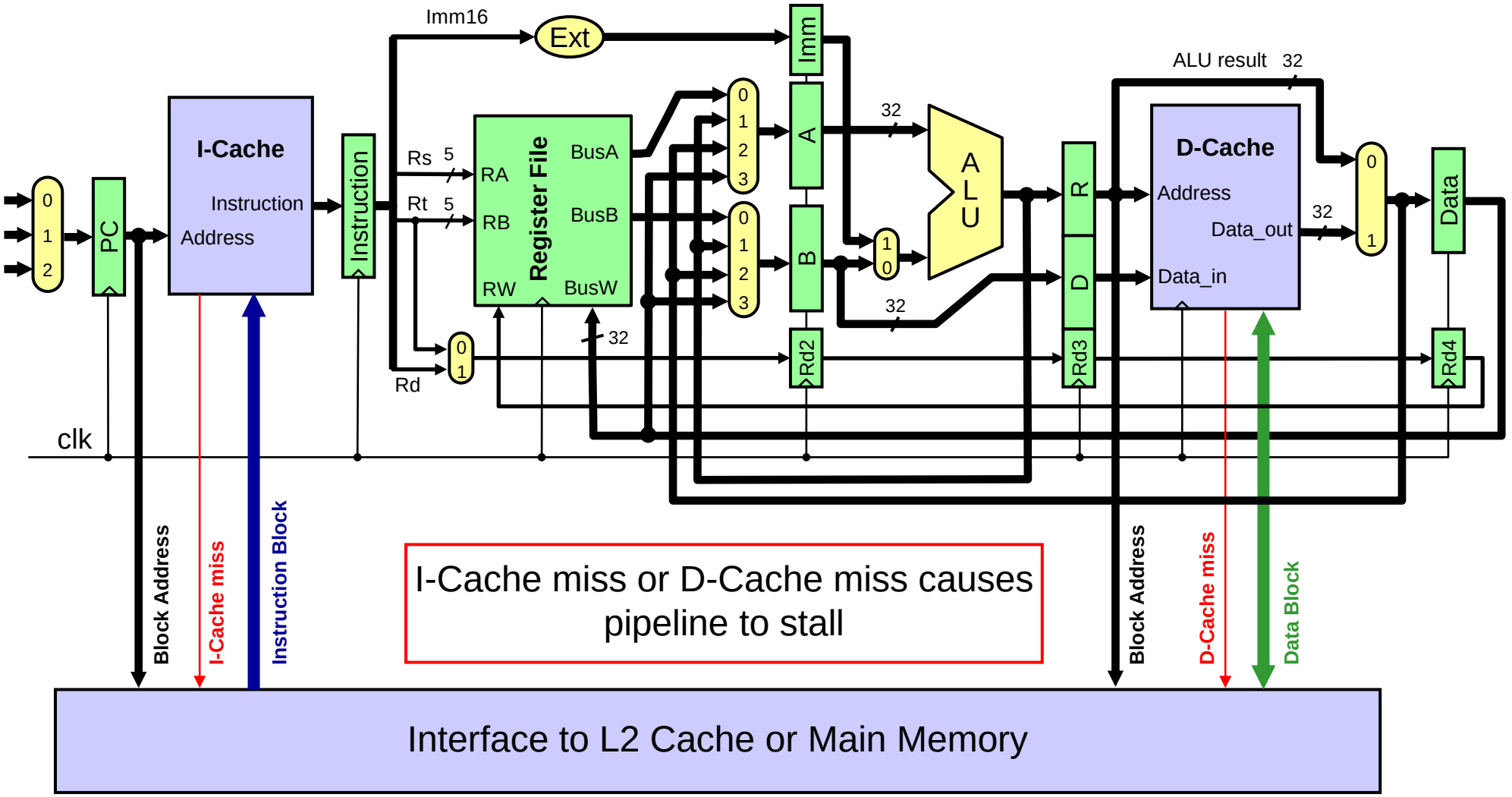
# Basic Cache Concepts

- Cache is the first level of the memory hierarchy once the address leaves the CPU and is searched first for the requested data.
- If the data requested by the CPU is present in the cache, it is retrieved from cache and the data access is a cache hit otherwise a cache miss and data must be read from main memory.
- On a cache miss a block of data must be brought in from main memory to cache to possibly *replace* an existing cache block.
- The allowed block addresses where blocks can be mapped (placed) into cache from main memory is determined by *cache placement strategy*.
- Locating a block of data in cache is handled by cache block identification mechanism: Tag matching.
- On a cache miss choosing the cache block being removed (replaced) is handled by the *block replacement strategy* in place.
- When a write to cache is requested, a number of main memory update strategies exist as part of *the cache write policy*.

# What is a Cache Memory ?

- Small and fast (SRAM) memory technology
  - Stores the subset of instructions & data currently being accessed
- Used to reduce average access time to memory
- Caches exploit **temporal locality** by …
  - Keeping recently accessed data closer to the processor
- Caches exploit **spatial locality** by …
  - Moving blocks consisting of multiple contiguous words
- Goal is to achieve
  - **Fast speed** of cache memory access
  - Balance the **cost** of the memory system

# Cache Memories in the Datapath



# Almost Everything is a Cache !

- In computer architecture, almost everything is a cache!
- Registers: a **cache on variables** – software managed
- First-level cache: a **cache on second-level cache**
- Second-level cache: a **cache on memory** (or L3 cache)
- Memory: a **cache on hard disk**
  - Stores recent programs and their data
  - Hard disk can be viewed as an extension to main memory
- Branch target and prediction buffer
  - **Cache on branch target and prediction** information

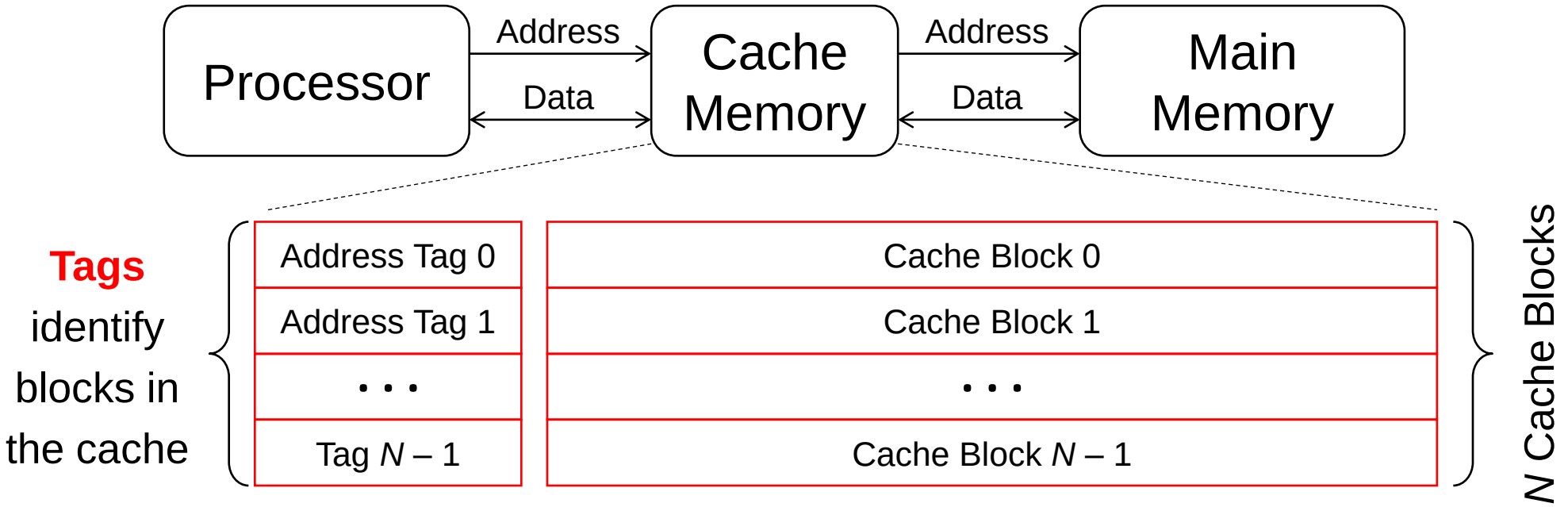
# Presentation Outline

- Memory Hierarchy and the need for Cache Memory
- **The Basics of Caches**
- Multilevel Caches
- Cache Performance and Memory Stall Cycles

# Four Basic Questions on Caches

- Q1: Where can a block be placed in a cache?
  - **Block placement**
  - Direct Mapped, Set Associative, Fully Associative
- Q2: How is a block found in a cache?
  - **Block identification**
  - Block address, tag, index
- Q3: Which block should be replaced on a cache miss?
  - **Block replacement**
  - FIFO, Random, LRU
- Q4: What happens on a write?
  - **Write strategy**
  - Write Back or Write Through cache (with Write Buffer)

# Inside a Cache Memory

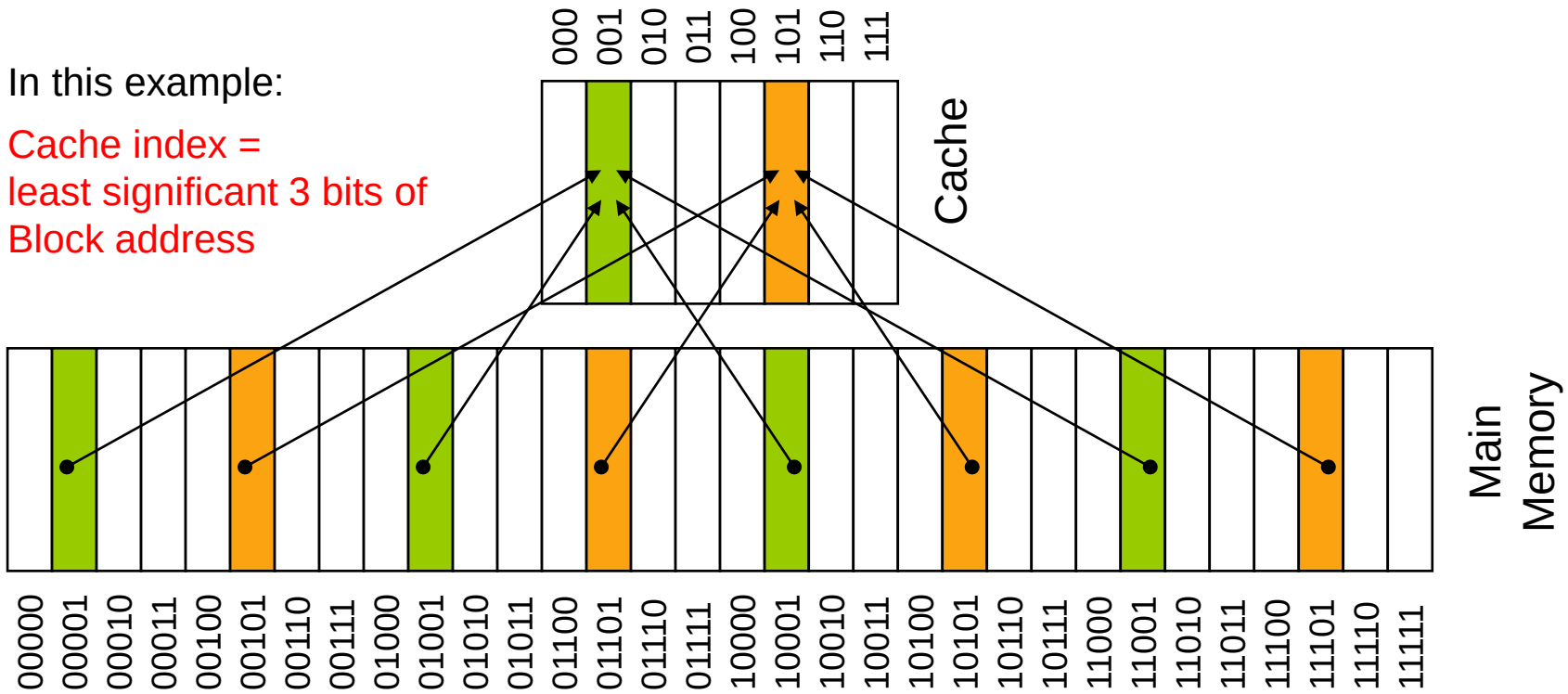



❖ **Cache Block** (or Cache Line)

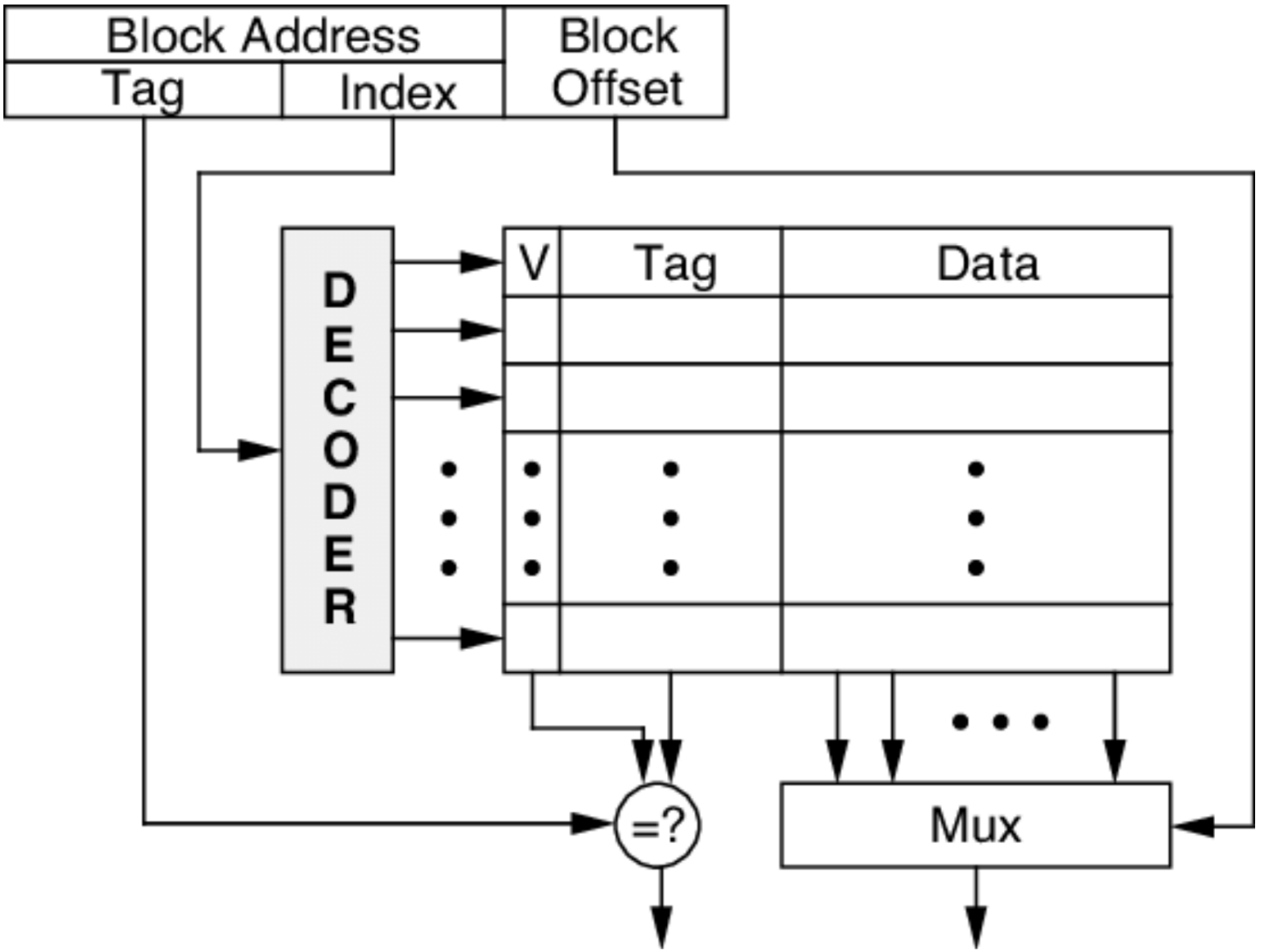
- ✧ Unit of data transfer between main memory and a cache
- ✧ Large block size ➔ Less tag overhead + Burst transfer from DRAM
- ✧ Typically, cache block size = 64 bytes in recent caches

# Block Placement: Direct Mapped

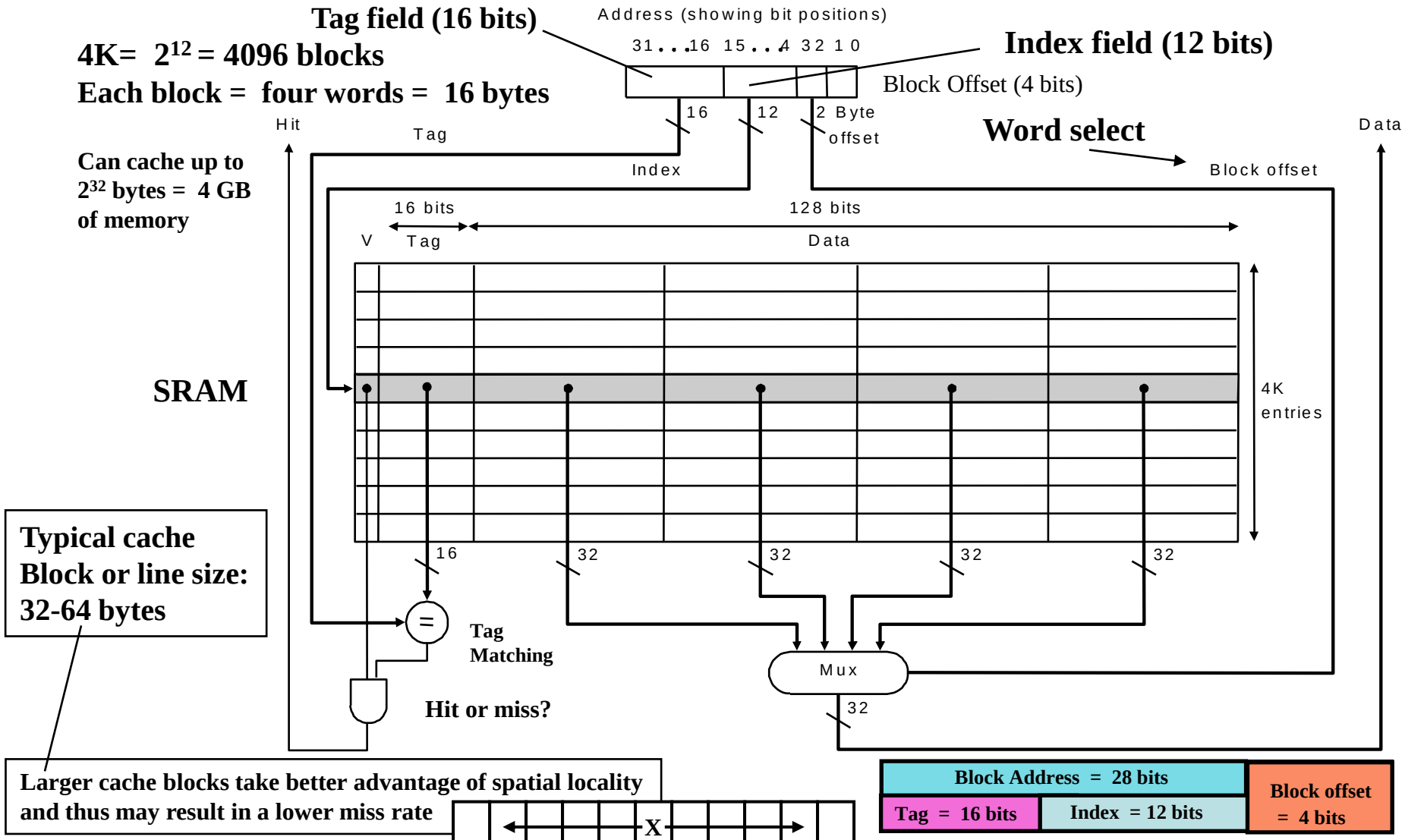
- **Block**: unit of data transfer between cache and memory
- **Direct Mapped Cache**:
  - A block can be placed in exactly one location in the cache



# Direct Mapped Cache



# 64KB Direct Mapped Cache Example




**Mapping Function:** Cache Block frame number = (Block address) MOD (4096)

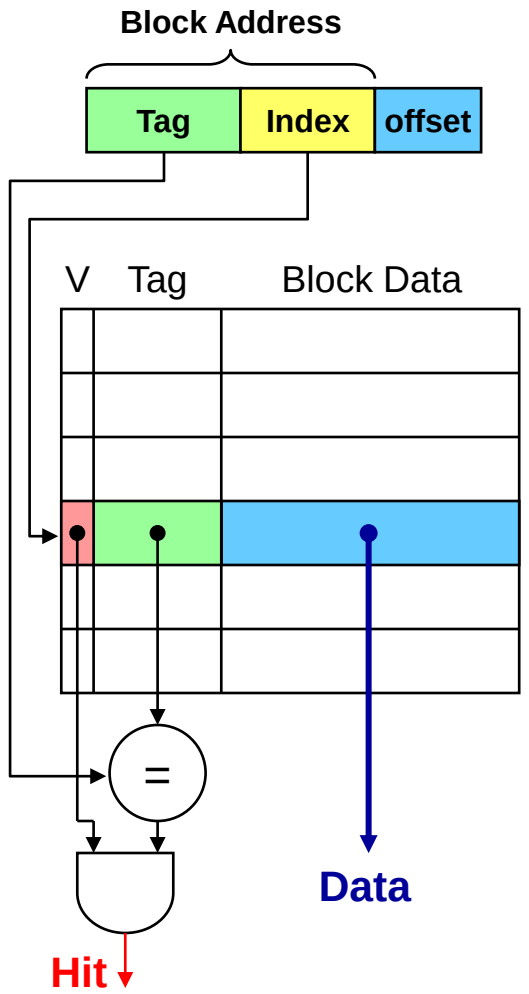
i.e. index field or 12 low bit of block address

**Hit Access Time = SRAM Delay + Hit/Miss Logic Delay**

# Direct-Mapped Cache

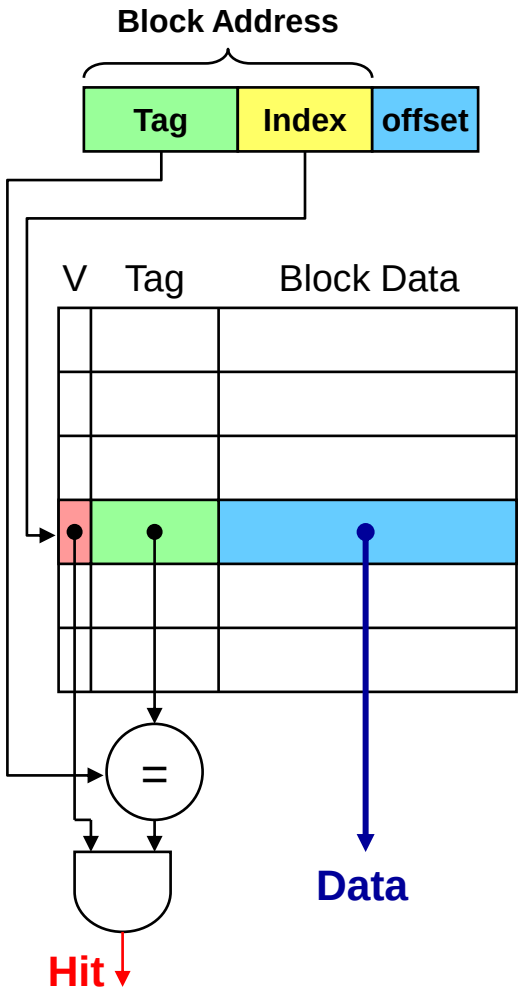
- A memory address is divided into
  - **Block address**: identifies block in memory
  - **Block offset**: to access bytes within a block
- A block address is further divided into
  - **Index**: used for direct cache access
  - **Tag**: most-significant bits of block address

  *Index = Block Address* **mod** *Cache Blocks*
- Tag must be stored also inside cache
  - For block identification
- A **valid bit** is also required to indicate
  - Whether a cache block is valid or not



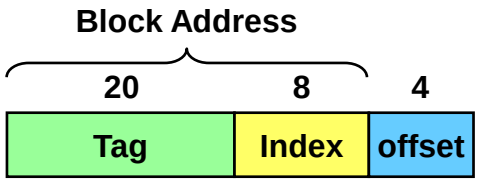
# Direct Mapped Cache – cont'd

- **Cache hit**: block is stored inside cache
  - Index is used to access cache block
  - Address tag is compared against stored tag
  - If equal and cache block is valid then **hit**
  - Otherwise: **cache miss**
- If number of cache blocks is $2^n$
  - $n$ bits are used for the cache index
- If number of bytes in a block is $2^b$
  - $b$ bits are used for the block offset
- If 32 bits are used for an address
  - $32 - n - b$ bits are used for the tag
- Cache data size = $2^{n+b}$ bytes



# Mapping an Address to a Cache Block

- Example
  - Consider a direct-mapped cache with 256 blocks
  - Block size = 16 bytes
  - Compute tag, index, and byte offset of address: 0x01FFF8AC
- **Solution**

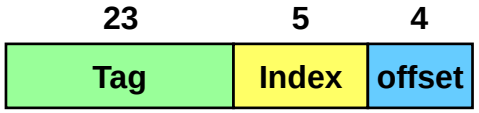



  - 32-bit address is divided into:
    - 4-bit byte offset field, because block size = $2^4$ = 16 bytes
    - 8-bit cache index, because there are $2^8$ = 256 blocks in cache
    - 20-bit tag field
  - Byte offset = 0xC = 12 (least significant 4 bits of address)
  - Cache index = 0x8A = 138 (next lower 8 bits of address)
  - Tag = 0x01FFF (upper 20 bits of address)

# Example on Cache Placement & Misses

- Consider a small direct-mapped cache with 32 blocks
  - Cache is initially empty, Block size = 16 bytes
  - The following memory addresses (in decimal) are referenced:

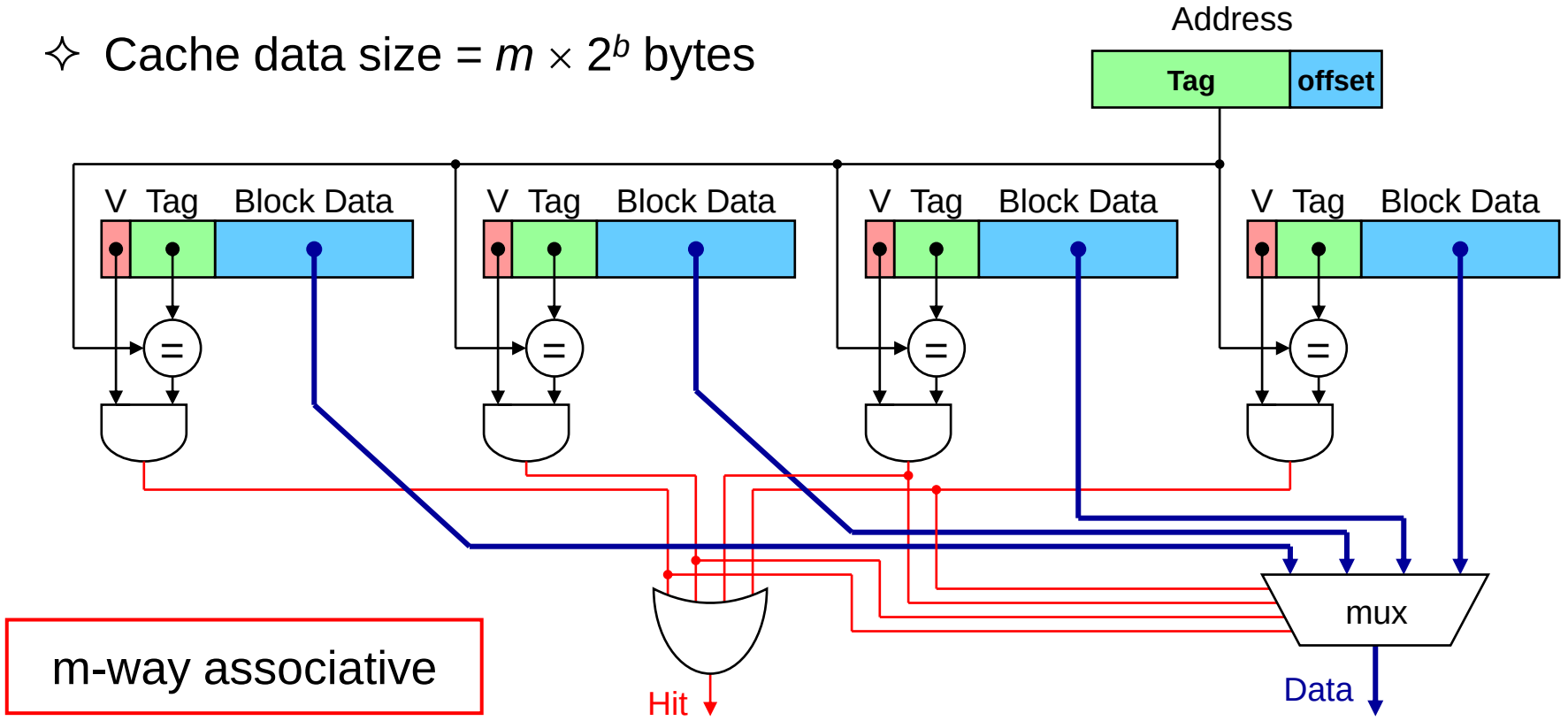
    1000, 1004, 1008, 2548, 2552, 2556.
  - Map addresses to cache blocks and indicate whether hit or miss




- Solution:
  - 1000 = 0x3E8 cache index = 0x1E Miss (first access)
  - 1004 = 0x3EC cache index = 0x1E Hit
  - 1008 = 0x3F0 cache index = 0x1F Miss (first access)
  - 2548 = 0x9F4 cache index = 0x1F Miss (different tag)
  - 2552 = 0x9F8 cache index = 0x1F Hit
  - 2556 = 0x9FC cache index = 0x1F Hit

# Fully Associative Cache

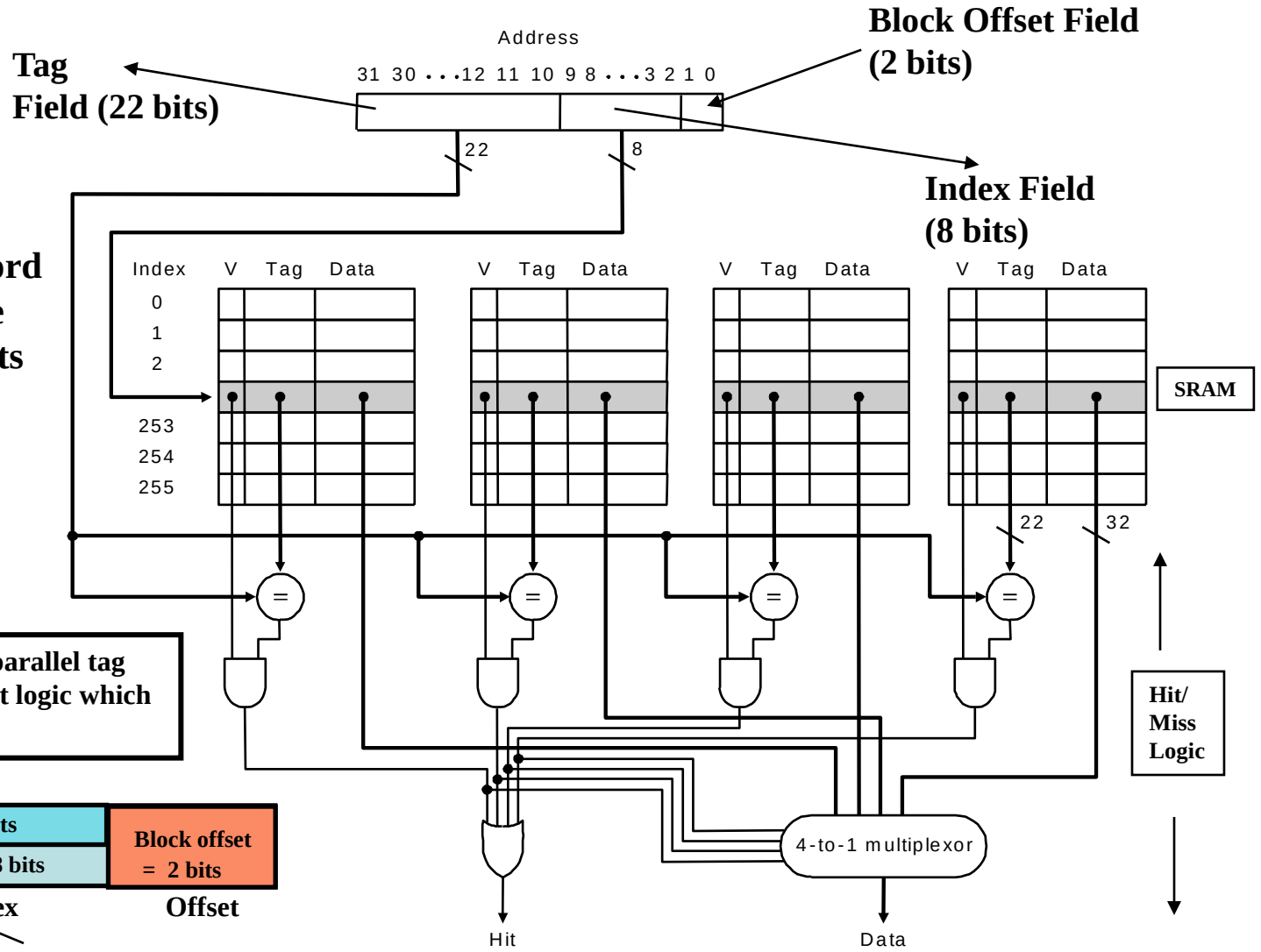
- A block can be placed anywhere in cache ⇒ no indexing
- If *m* blocks exist then
  - *m* comparators are needed to match *tag*
  - Cache data size = $m \times 2^b$ bytes



# Set-Associative Cache

- A **set** is a group of blocks that can be indexed
- One set = ***m*** blocks ➔ ***m*-way** set associative
- Set index field is $k$-bit long ➔ Number of Sets = $2^k$
- Set index is decoded and only one set is examined
  - ***m*** tags are checked in parallel using ***m*** comparators
  - If address tag matches a stored tag within set then **Cache Hit**
  - Otherwise: **Cache Miss**
- Cache data size = $m \times 2^{k+b}$ bytes ($2^b$ bytes per block)
- A direct-mapped cache has one block per set ($m = 1$)
- A fully-associative cache has only one set ($k = 0$)

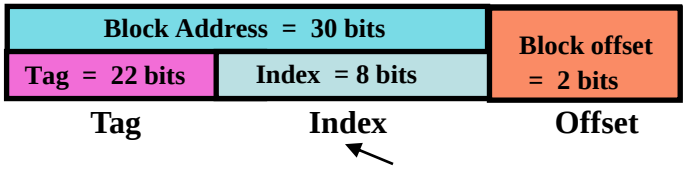
# 4K Four-Way Set Associative Cache: MIPS Implementation Example




**1024 block frames**
**Each block = one word**
**4-way set associative**
**1024 / 4= $2^8$= 256 sets**

**Can cache up to**
**$2^{32}$ bytes = 4 GB**
**of memory**

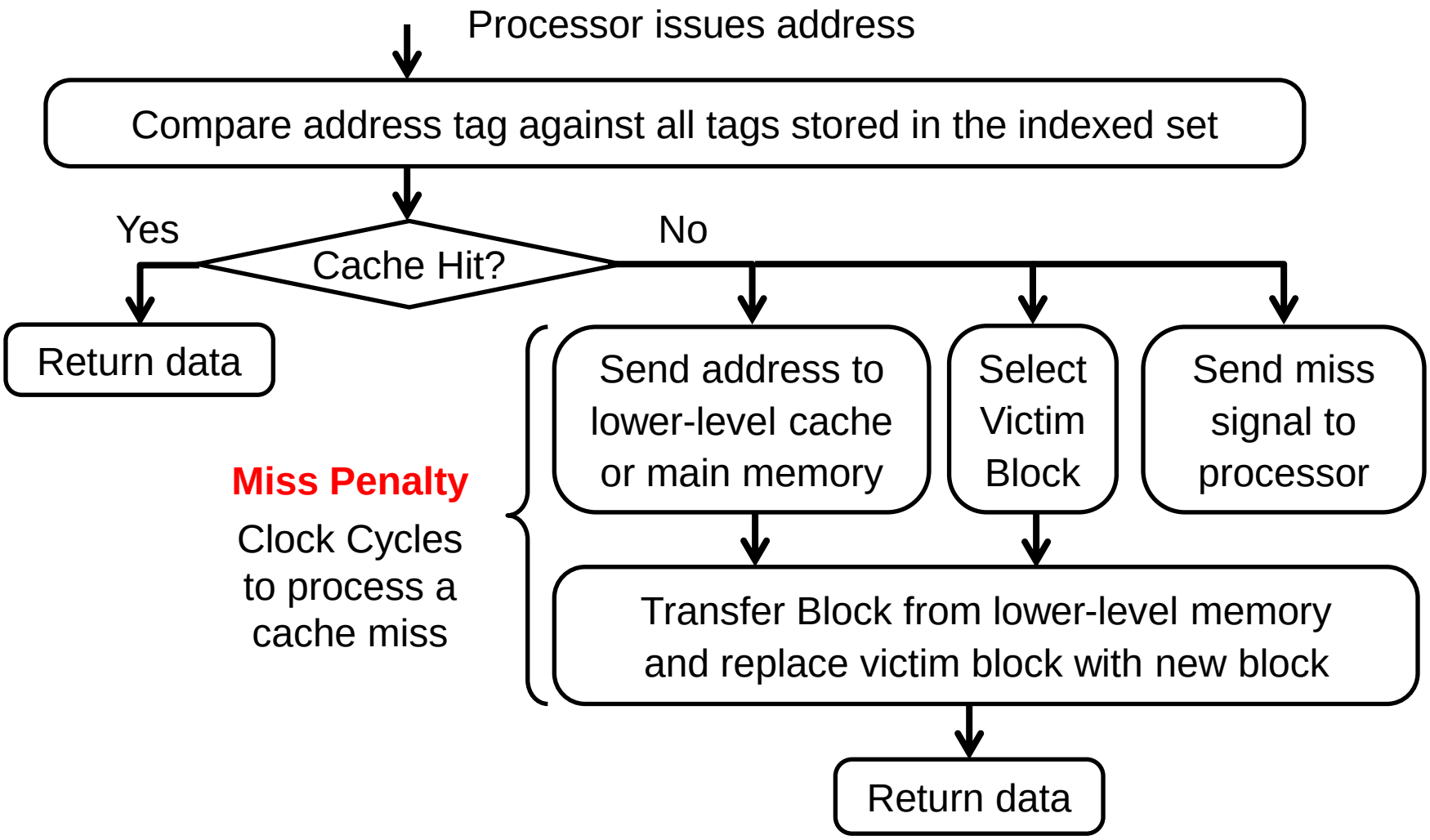
**Set associative cache requires parallel tag matching and more complex hit logic which may increase hit time**




**Mapping Function:** **Cache Set Number = index= (Block address) MOD (256)**

**Hit Access Time = SRAM Delay + Hit/Miss Logic Delay**

# Handling a Cache Read



# Replacement Policy

- Which block to replace on a cache miss?
- No choice for direct-mapped cache
- ***m*** choices for *m*-way set associative cache
- **Random replacement**
  - Candidate block is selected randomly
  - One counter for all sets (0 to *m* – 1): incremented on every cycle
  - On a cache miss, replace block specified by counter
- **First In First Out (FIFO) replacement**
  - Replace oldest block in set (Round-Robin)
  - One counter per set (0 to *m* – 1): specifies oldest block to replace
  - Counter is incremented on a cache miss

# LRU Replacement Policy

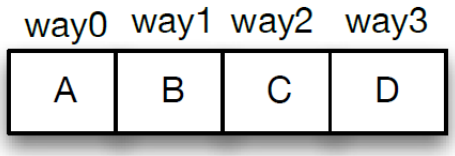
## ❖ LRU: Replace Least Recently Used Block in a Set

- ✧ LRU state must be updated on every cache hit
- ✧ If *m* = 2, there are only 2 permutations ➔ a single LRU bit is needed
- ✧ If *m* = 4 then 4! = 24 permutations. If *m* = 8 then 8! = 40320 permutations
- ✧ Pure LRU is difficult to implement for large *m*

## ❖ Pseudo LRU Tree

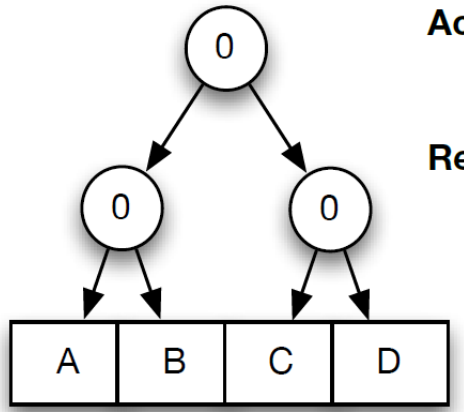
- ✧ LRU approximation that requires (*m* – 1) LRU bits per set

**Cache Content:**




On cache hit, track the MRU block
On cache miss, replace non-MRU block

**Pseudo LRU Tree**

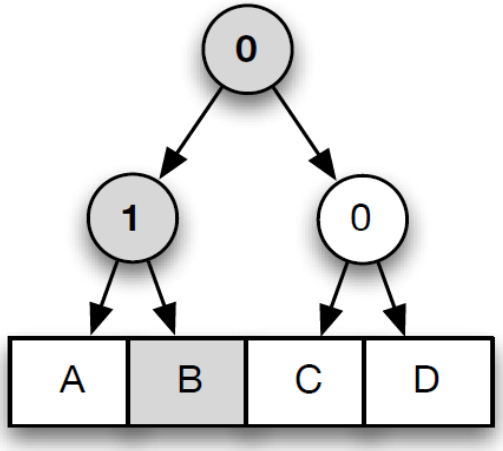



**Access:**
0 = Left, 1 = Right

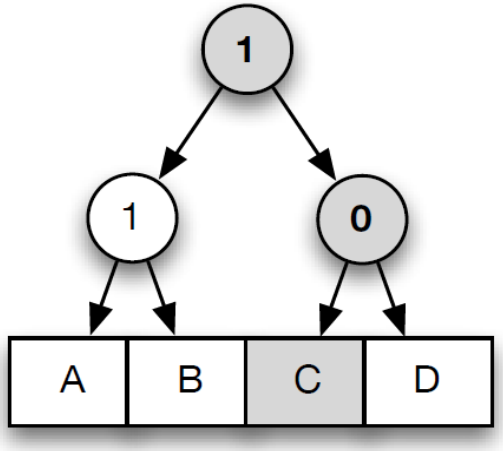
**Replacement:**
0 = Right, 1 = Left

# Access Stream: B, C, A, D, E

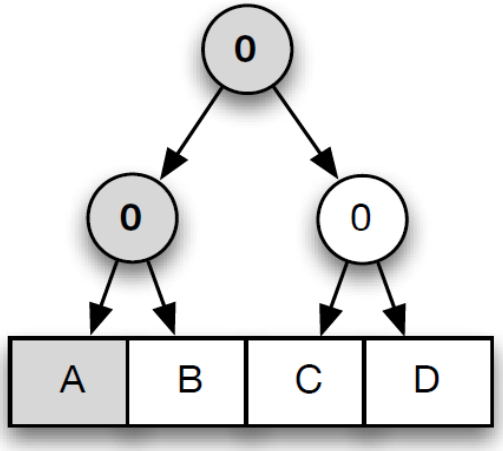
*After access to B*



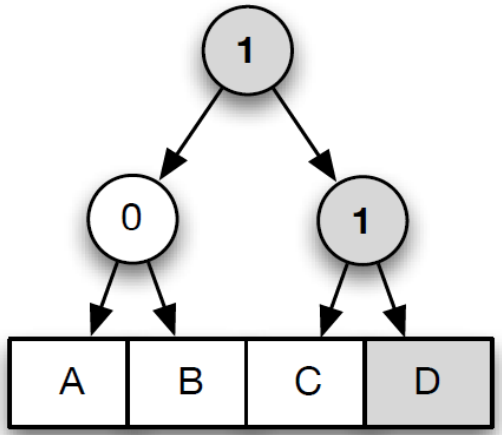

*After access to C*



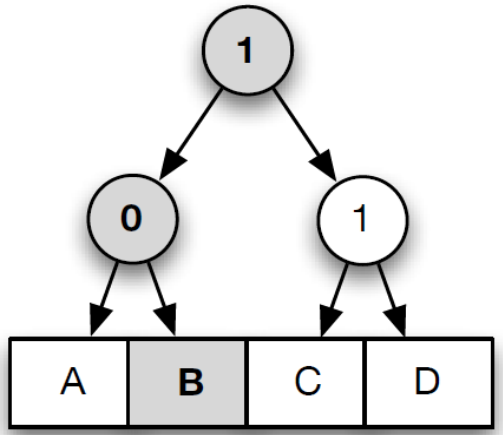

*After access to A*



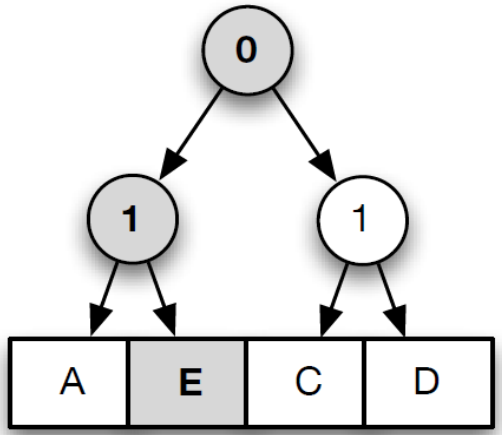

*After access to D*




Finding a block to replace
to make room for E
Block B is selected




*After access to E,*
E replaces B,
Bits flipped along the path



# Comparing Random, FIFO, and LRU

- Data cache misses per 1000 instructions
  - 10 SPEC2000 benchmarks on Alpha processor
  - Block size = 64 bytes
  - LRU outperforms FIFO and Random for a small cache
  - Little difference between LRU and Random for a large cache
- Random is the simplest to implement: one counter for all sets
- Pseudo LRU Tree requires ($m$ – 1) replacement bits per set

| | 2-way | | | 4-way | | | 8-way | | |
|---|---|---|---|---|---|---|---|---|---|
| **Size** | **LRU** | **Rand** | **FIFO** | **LRU** | **Rand** | **FIFO** | **LRU** | **Rand** | **FIFO** |
| 16 KB | 114.1 | 117.3 | 115.5 | 111.7 | 115.1 | 113.3 | 109.0 | 111.8 | 110.4 |
| 64 KB | 103.4 | 104.3 | 103.9 | 102.4 | 102.3 | 103.1 | 99.7 | 100.5 | 100.3 |
| 256 KB | 92.2 | 92.1 | 92.5 | 92.1 | 92.1 | 92.5 | 92.1 | 92.1 | 92.5 |

# Write Policy

## ❖ Write Through

- ✧ Writes update data cache and lower-level (L2) cache
- ✧ Cache control bit: only a Valid bit is needed
- ✧ Lower-level cache always has latest data, which simplifies data coherency
- ✧ Can always discard cached data when a block is replaced

## ❖ Write Back

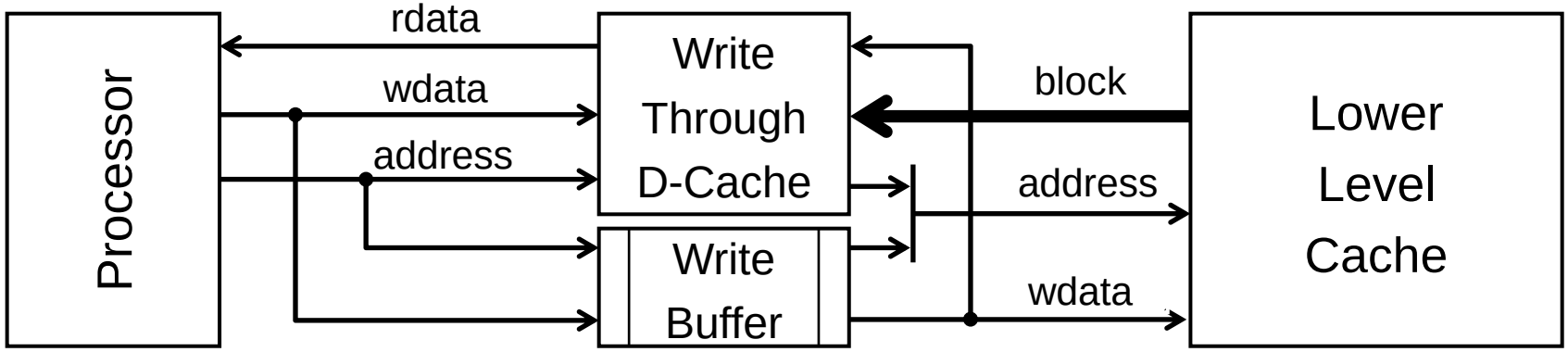
- ✧ Writes update data cache only
- ✧ Two cache control bits: Valid and Modified bits are required
- ✧ Modified bit indicates that cache block ≠ memory block
- ✧ Modified cached block is written back when replaced
- ✧ Multiple writes to same cache block require only one write-back
- ✧ Uses less memory bandwidth than write-through and less energy

# Write Miss Policy

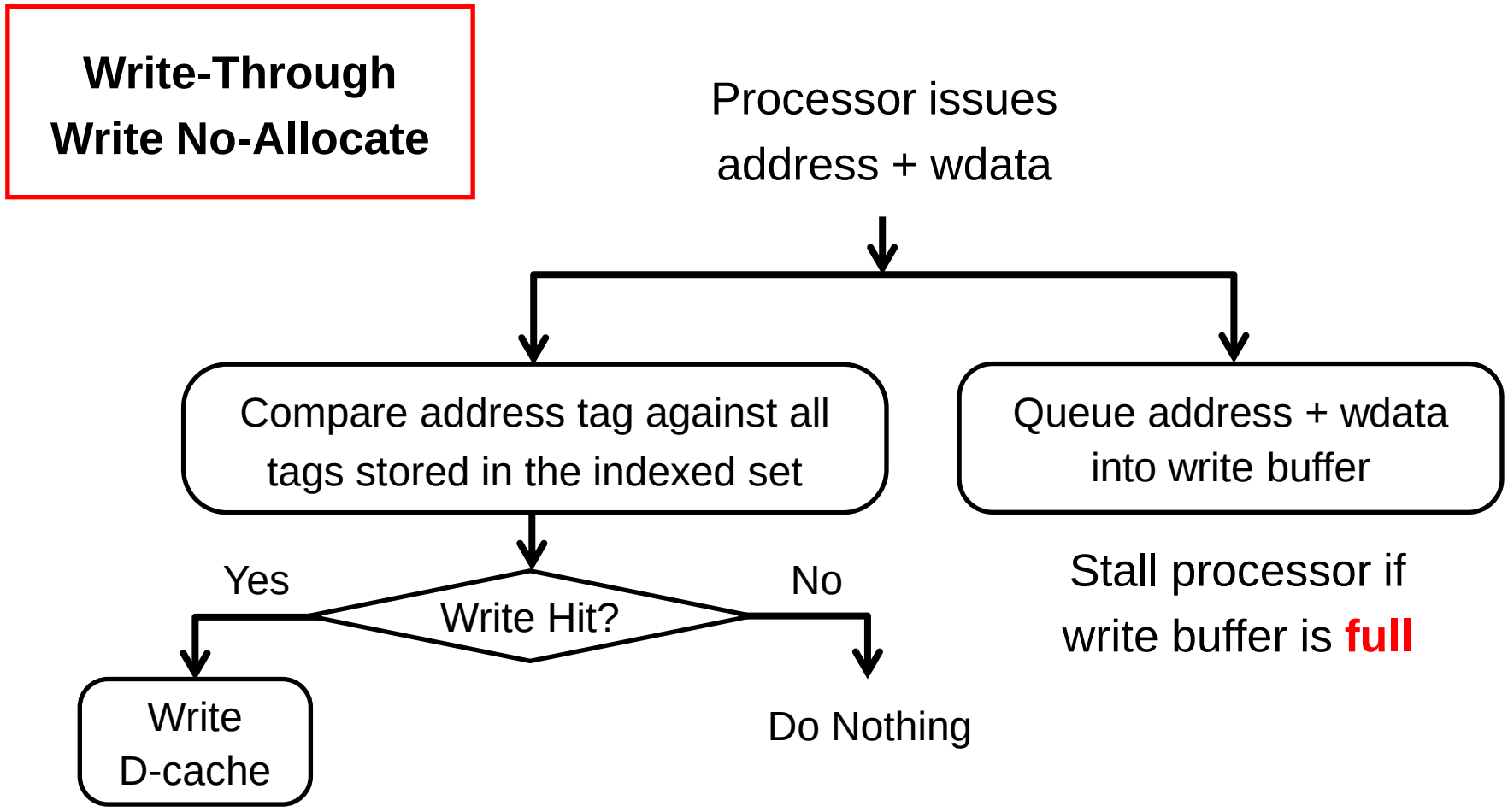
- What happens on a write miss?
- **Write Allocate**
  - Allocate new block in data cache
  - Write miss acts like a read miss, block is fetched then updated
- **Write No-Allocate**
  - Send data to lower-level (L2) cache
  - Data cache is not modified
- Either write-miss policy can be combined with either write policy
- **Write back** caches typically use **write allocate** on a write miss
  - Reasoning: subsequent writes will be captured in the cache
- **Write-through** caches might choose **write no-allocate**
  - Reasoning: writes must still go to lower level memory

# Write Buffer

- Write buffer is a queue that holds: address + write-data (wdata)
  - Write-through: all writes are sent to lower-level cache
- Buffer decouples the write from the memory bus writing
  - Write occurs **without stalling processor**, until buffer is full
- **Problem:** write buffer may hold data on a read miss
  - If address is found, return data value in write buffer
  - Transfer block from lower level cache and update D-cache

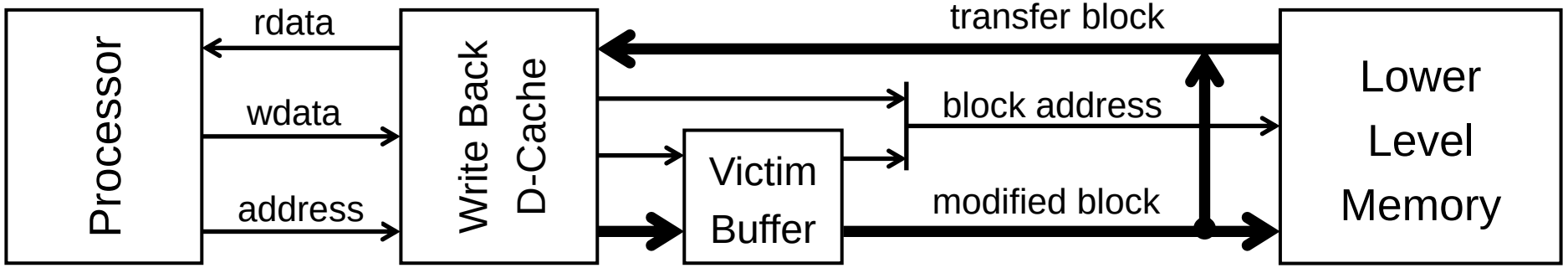


# Write Through with Write No-Allocate

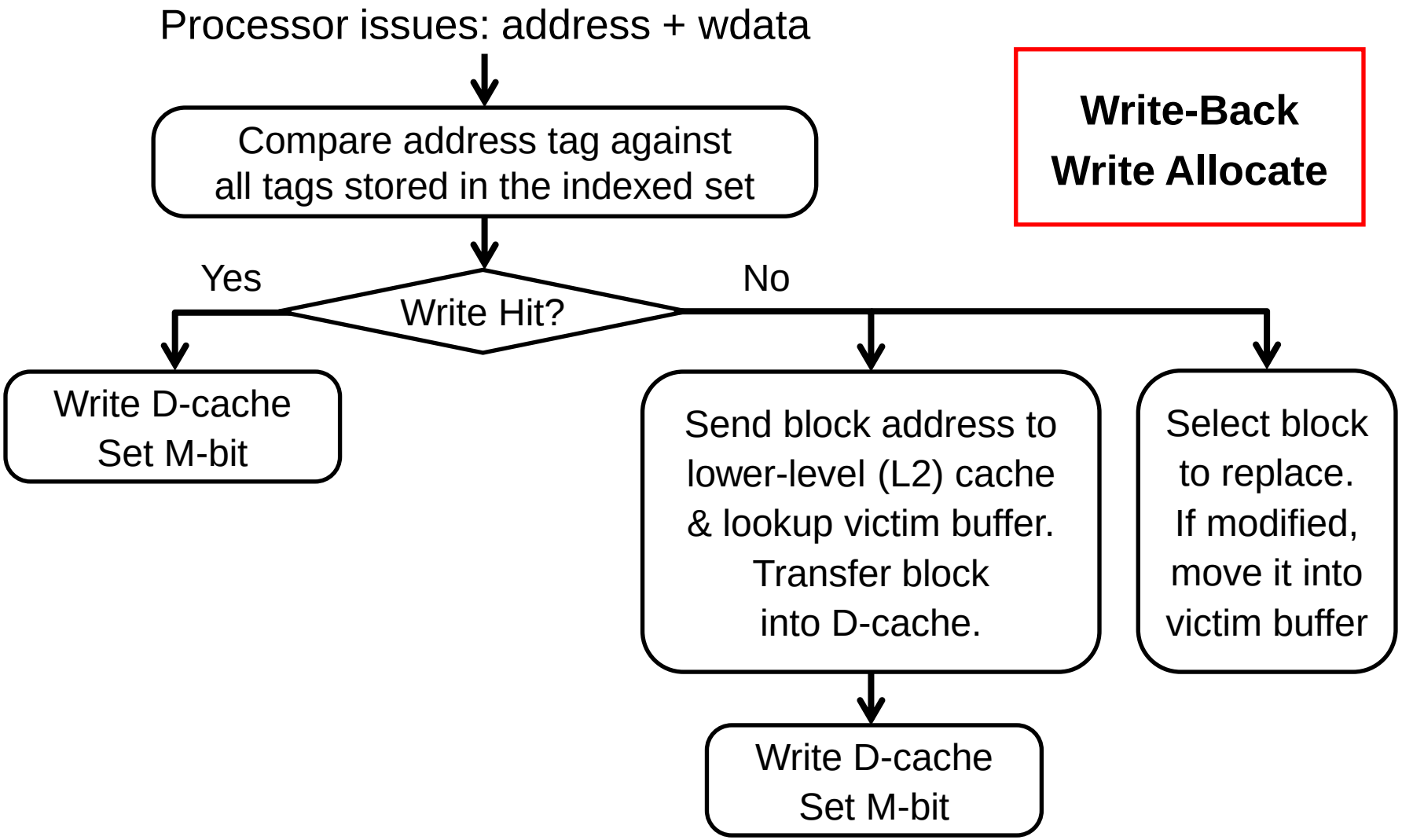


# Victim Buffer

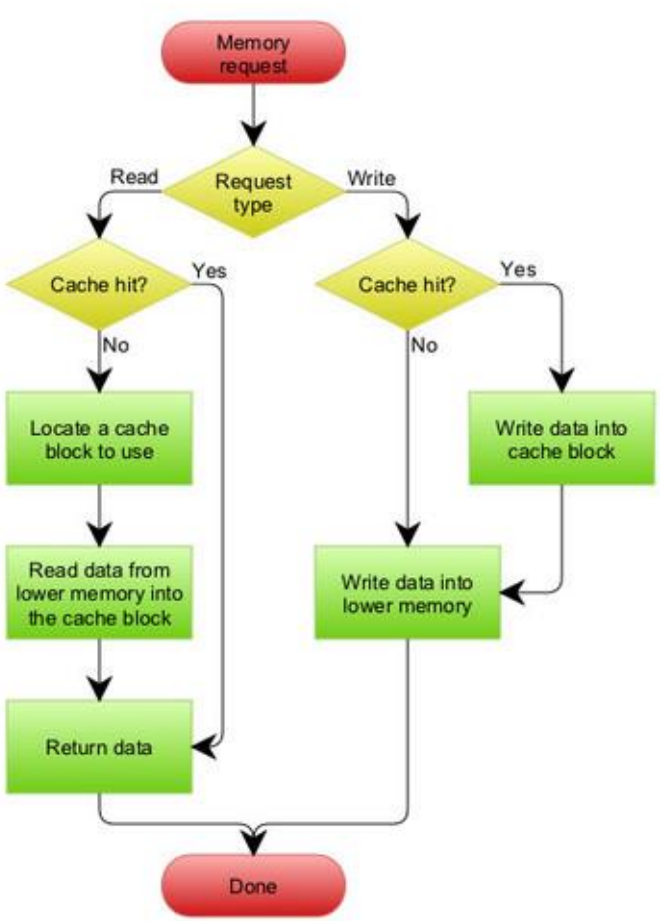
- Used by modified evicted blocks and their addresses
  - When a modified block is replaced (evicted) in the D-cache
  - Prepares a modified block to be written back to lower memory
- Victim buffer decouples the write back to lower memory
  - Giving priority to read miss over write back to reduce miss penalty
- Problem: Victim buffer may hold block on a cache miss
  - Solution: transfer modified block in victim buffer into data cache

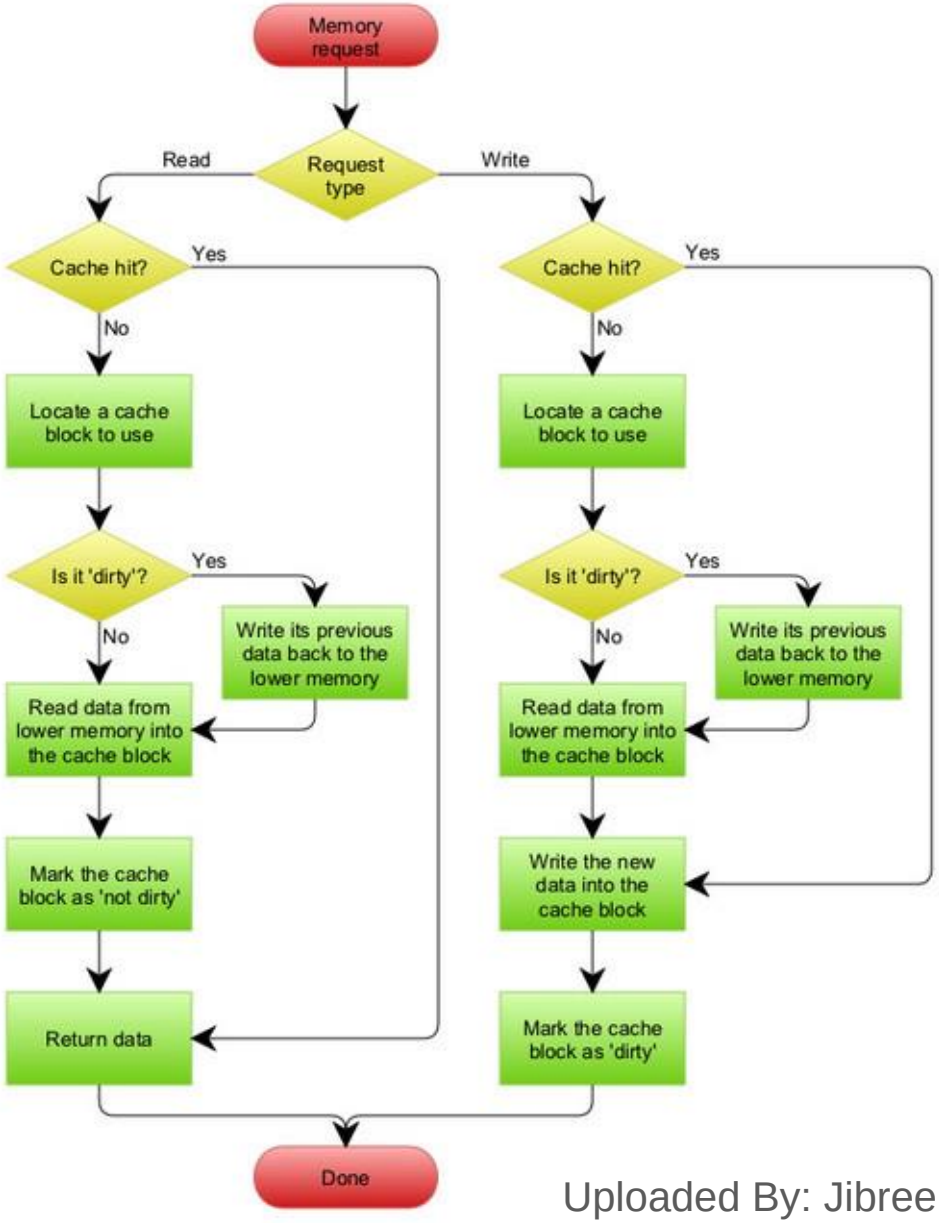


# Write Back with Write Allocate



# Write Through vs. Write Back



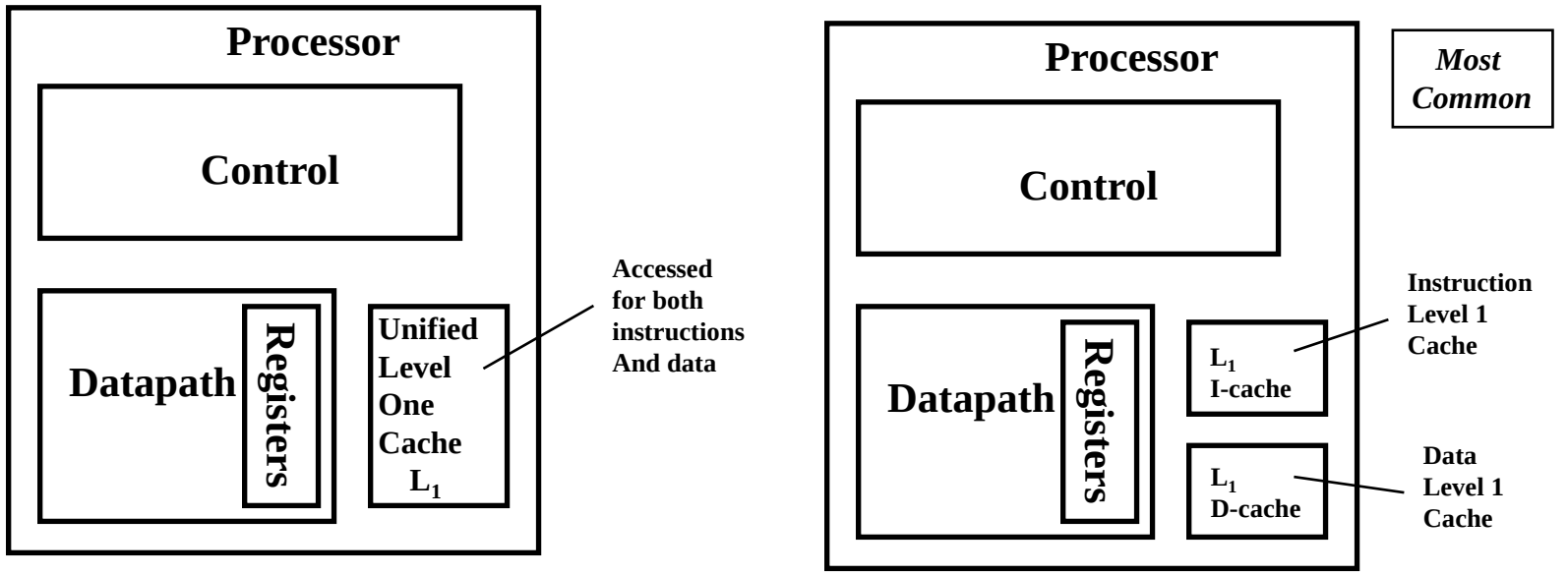
# Unified vs. Separate Level 1 Cache

- Unified Level 1 Cache (Princeton Memory Architecture). **AKA Shared Cache**

  A single level 1 ($L_1$) cache is used for both instructions and data.

- Separate instruction/data Level 1 caches (Harvard Memory Architecture):

  The level 1 ($L_1$) cache is split into two caches, one for instructions (instruction cache, $L_1$ I-cache) and the other for data (data cache, $L_1$ D-cache).



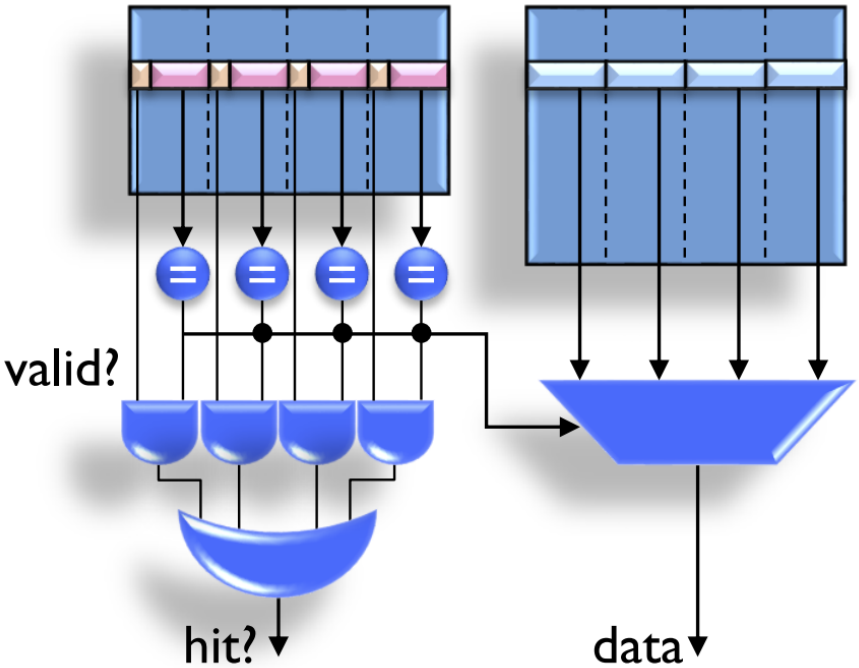

Why?

**Split Level 1 Cache is more preferred in pipelined CPUs to avoid instruction fetch/Data access structural hazards**

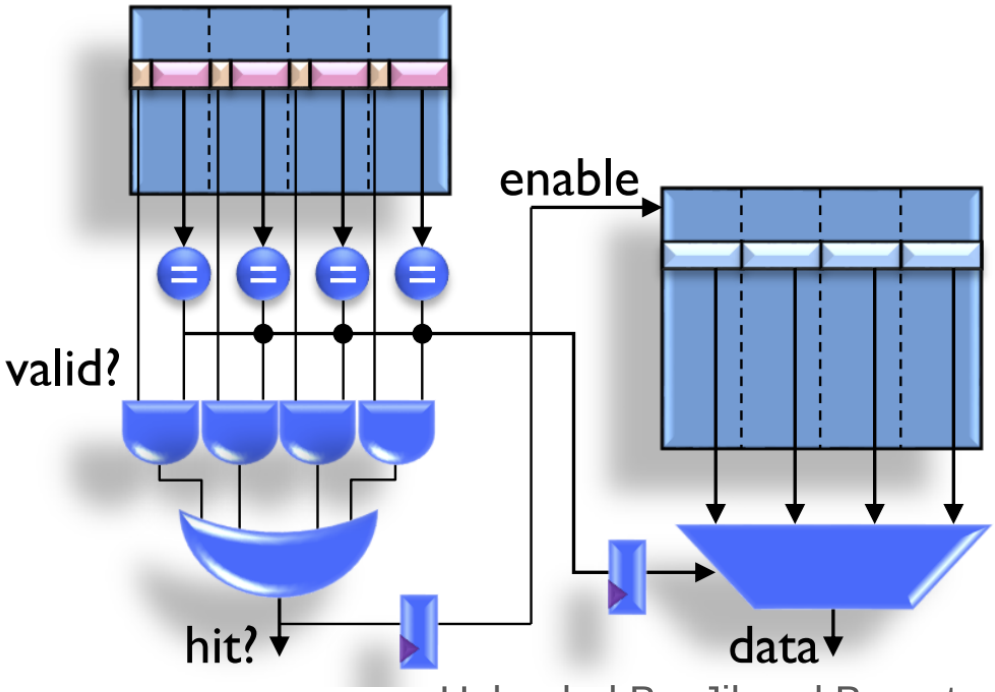
# Parallel vs. Serial Caches

- Tag and Data usually separate (tag is smaller & faster)
  - – State bits stored along with tags
    - Valid bit, “LRU” bit(s), …

Parallel access to Tag and Data reduces latency (good for L1)




Serial access to Tag and Data reduces power (good for L2+)

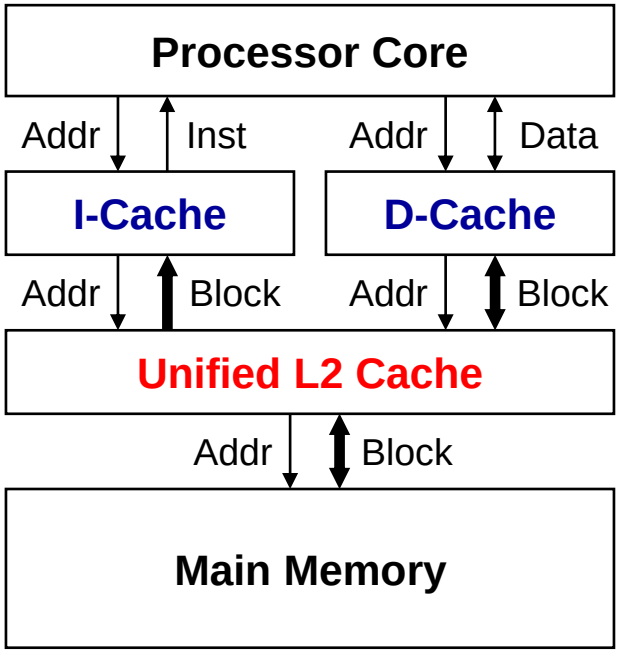


# Presentation Outline

- Memory Hierarchy and the need for Cache Memory
- The Basics of Caches
- **Multilevel Caches**
- Cache Performance and Memory Stall Cycles

# Multilevel Caches

- Top level cache is kept small to
  - Reduce hit time
  - Reduce energy per access
- Add another cache level to
  - Reduce the memory gap
  - Reduce memory bus loading
- Multilevel caches can help
  - Reduce miss penalty
  - Reduce average memory access time
- Large L2 cache can capture many misses in L1 caches
  - Reduce the global miss rate




**For simplicity,**
**L3 cache is not included**

# Inclusion Policy

- Multi-Level Cache Inclusion
  - The content of the L1 cache is fully contained in the larger L2 cache
  - Wastes L2 cache space, but L2 has space for additional blocks
- Multi-Level Cache Exclusion
  - The L2 cache can only have blocks that are not in the L1 cache
  - Prevents wasting space
- Non-Inclusive Non-Exclusion (NINE)
  - Multi-level inclusion and exclusion must be enforced by a protocol
  - If neither is enforced then L2 cache is neither inclusive nor exclusive

# Multilevel Inclusion

- L1 cache blocks are always present in L2 cache
  - Total number of unique cache blocks = L2 cache size
- A miss in L1, but a hit in L2 copies block from L2 to L1
- A miss in L1 and L2 brings a block into L1 and L2
- A write in L1 causes data to be written in L1 and L2
- Write-through policy is used from L1 to L2
- Write-back policy is used from L2 to lower-level memory
  - To reduce traffic on the memory bus
- A replacement (or invalidation) in L2 must be seen in L1
  - A block which is evicted from L2 must also be evicted from L1

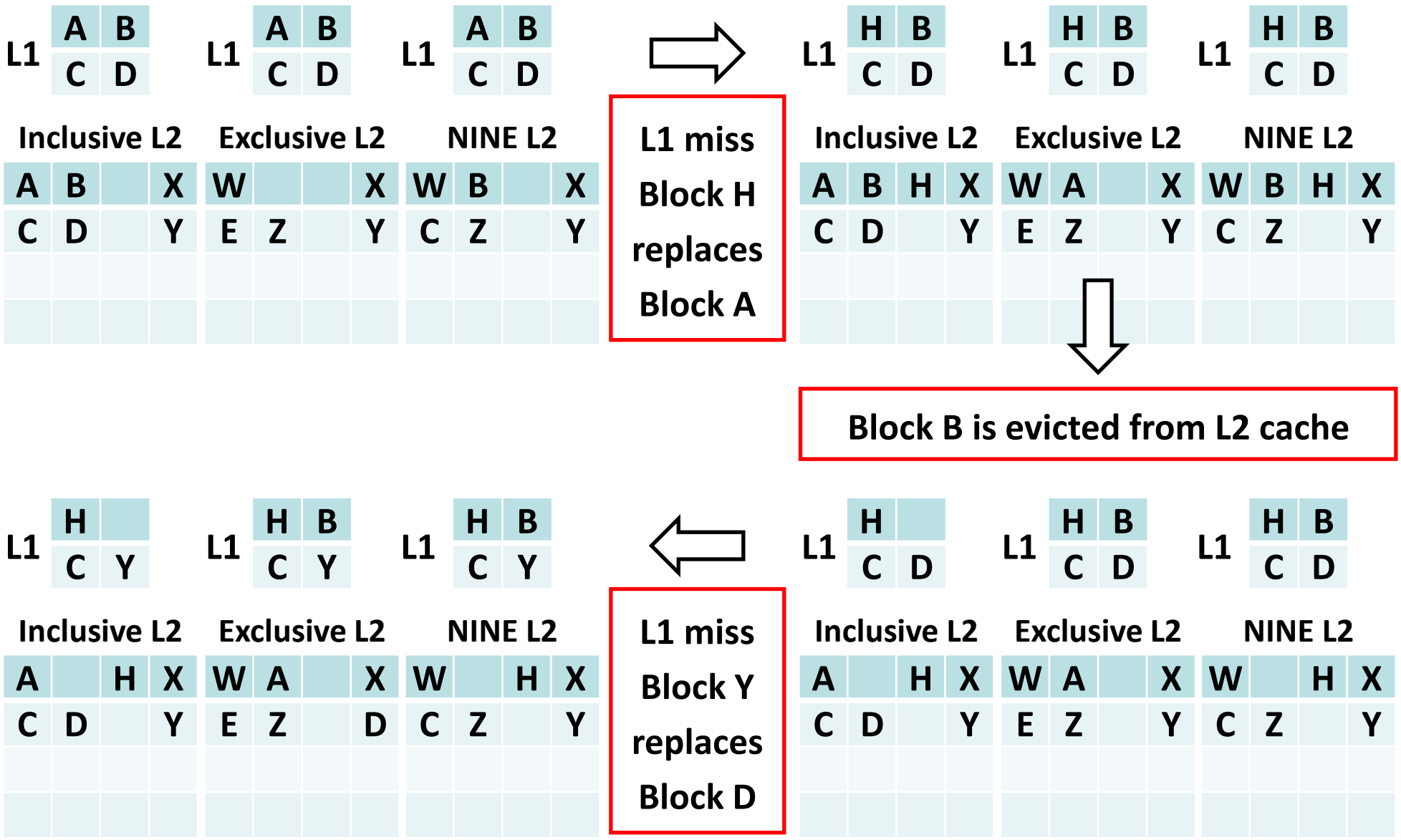
# Multilevel Exclusion

- L1 cache blocks are never found in L2 cache
  - Prevents wasting space
  - Total number of unique cache blocks = L1 + L2 cache size
- Cache miss in L1 and L2 brings the block into L1 only
- Cache miss in L1, but hit in L2 results in a swap of blocks
  - More complex to implement
- Block replaced in L1 is moved into L2
  - L2 cache stores L1 evicted blocks, in case needed later in L1
  - L2 cache acts as a **victim cache**
- Write-Back policy from L1 to L2
- Write-Back policy from L2 to lower-level memory

# Non-Inclusive, Non Exclusive (NINE)

- L1 cache blocks may or may not be present in L2 cache
  - Total number of unique cache blocks > L2 cache size, but < (L1 + L2) size
- Cache miss in L1 and L2 brings a block into L1 and L2
- A miss in L1, but a hit in L2 copies block from L2 to L1
  - A block replaced in L1 can be discarded, if not modified
  - If a block is replaced and modified in L1, then it must be written back to L2
- A block can be replaced in L2 without removing it from L1
  - If L2 is inclusive, then block evicted from L2 must also be evicted from L1
- Write-back policy between L1 and L2
  - Write-through is not possible if a block exists in L1, but replaced in L2

# Inclusive, Exclusive, and NINE L2 Cache



# Local and Global Miss Rates

## ❖ Local Miss Rate

Number of cache misses / Memory accesses to this cache

**Miss Rate$_{L1}$** for L1 cache

**Miss Rate$_{L2}$** for L2 cache

## ❖ Global Miss Rate

Number of cache misses / Memory accesses generated by processor

**MissRate$_{L1}$** for L1 cache (same as local)

**MissRate$_{L1}$ × MissRate$_{L2}$** for L2 cache (different from local)

Global miss rate is a better measure for L2 cache

Fraction of the total memory accesses that miss in L1 and L2

# Presentation Outline

- Memory Hierarchy and the need for Cache Memory
- The Basics of Caches
- Multilevel Caches
- **Cache Performance and Memory Stall Cycles**

# Hit Rate and Miss Rate

- Hit Rate = Hits / (Hits + Misses)
- Miss Rate = Misses / (Hits + Misses)
- I-Cache Miss Rate = Miss rate in the Instruction Cache
- D-Cache Miss Rate = Miss rate in the Data Cache
- Example:
  - Out of 1000 instructions fetched, 60 missed in the I-Cache
  - 25% are load-store instructions, 50 missed in the D-Cache
  - What are the I-cache and D-cache miss rates?
- I-Cache Miss Rate = 60 / 1000 = 6%
- D-Cache Miss Rate = 50 / (25% × 1000) = 50 / 250 = 20%

# Memory Stall Cycles

- The processor **stalls** on a Cache miss
  - When fetching instruction from I-Cache and Block is not present
  - When loading/storing data in a D-cache and Block is not present
  - When writing data in a write-though D-cache and write buffer is full
- Miss Penalty: clock cycles to process a cache miss

  Miss Penalty is assumed equal for I-cache & D-cache

  Miss Penalty is assumed equal for Load and Store
- Memory Stall Cycles =

  I-Cache Misses × Miss Penalty +

  D-Cache Read Misses × Miss Penalty +

  D-Cache Write Misses × Miss Penalty

# Combined Misses

- Combined Misses =

  I-Cache Misses + D-Cache Read Misses + Write Misses

  I-Cache Misses = I-Count × I-Cache Miss Rate

  Read Misses = Load Count × D-Cache Read Miss Rate

  Write Misses = Store Count × D-Cache Write Miss Rate

- Combined misses are often reported per 1000 instructions

- Memory Stall Cycles = Combined Misses × Miss Penalty

# Memory Stall Cycles Per Instruction

- Memory Stall Cycles Per Instruction =

  Combined Misses Per Instruction × Miss Penalty

  Miss Penalty is assumed equal for I-cache and D-cache

  Miss Penalty is assumed equal for Load and Store

- Combined Misses Per Instruction =

  I-Cache Miss Rate +

  Load Frequency × D-Cache Read Miss Rate +

  Store Frequency × D-Cache Write Miss Rate

# Example on Memory Stall Cycles

- Consider a program with the given characteristics
  - 20% of instructions are load and 10% are store
  - I-cache miss rate is 2%
  - D-cache miss rate is 5% for load, and 1% for store
  - Miss penalty is 20 clock cycles for all cases (I-Cache & D-Cache)
  - Compute combined misses and stall cycles per instruction
- Combined misses per instruction in I-Cache and D-Cache
  - 2% + 20%×5% + 10%×1% = 0.031 combined misses per instruction
  - Equal to an average of 31 misses per 1000 instructions
- Memory stall cycles per instruction
  - 0.031 × 20 (miss penalty) = 0.62 memory stall cycles per instruction

# CPU Time with Memory Stall Cycles

CPU Time =

(CPU execution cycles + Memory Stall Cycles) × Clock Cycle

$$\text{CPU Time} = \text{I-Count} \times \text{CPI}_{\text{overall}} \times \text{Clock Cycle}$$

$$\text{CPI}_{\text{overall}} = \text{CPI}_{\text{execution}} + \text{Memory Stall Cycles per Instruction}$$

- $\text{CPI}_{\text{overall}}$ = Overall CPI in the presence of cache misses
- $\text{CPI}_{\text{execution}}$ = Execution CPI (not counting cache misses)
- Memory stall cycles per instruction increases the overall CPI

# Example on CPI with Memory Stalls

- A processor has $CPI_{execution}$ = 1.5 (not counting cache misses)
  - I-Cache miss rate is 2%, D-cache miss rate is 5% for load & store
  - 20% of instructions are loads and stores
  - Cache miss penalty is 100 clock cycles for I-cache and D-cache
- What is the impact of cache misses on the overall CPI?
- Answer: Memory Stall Cycles per Instruction =

  0.02×100 (I-Cache) + 0.2×0.05×100 (D-Cache) = 3

  $CPI_{overall}$ = 1.5 + 3 = 4.5 cycles per instruction

  $CPI_{overall}$ / $CPI_{execution}$ = 4.5 / 1.5 = 3

  Processor is 3 times slower due to memory stall cycles

# Average Memory Access Time

- Average Memory Access Time (AMAT)

  AMAT = Hit time + Combined Miss rate × Miss penalty

  Hit Time = time to access the I-cache or D-cache (for a hit)

  Hit time is assumed to be the same for I-cache and D-cache

- Combined Miss Rate for Instruction Access and Data Access

$$\text{Combined Miss Rate} = \frac{\text{Combined Misses per Instruction}}{\text{Memory Accesses per Instruction}}$$

Memory Accesses per Instruction = 1 + %Load + %Store

Miss penalty is assumed to be the same for I-cache and D-cache

# Memory Hierarchy Performance:
## Average Memory Access Time (AMAT), Memory Stall cycles

- The Average Memory Access Time (AMAT): The number of cycles required to complete an average memory access request by the CPU.
- Memory stall cycles per memory access: The number of stall cycles added to CPU execution cycles for one memory access.
- Memory stall cycles per average memory access = (AMAT -1)
- For ideal memory: AMAT = 1 cycle, this results in zero memory stall cycles.
- Memory stall cycles per average instruction =

Number of memory accesses per instruction

Instruction Fetch ↓ x Memory stall cycles per average memory access

= ( 1 + fraction of loads/stores) x (AMAT -1 )

Base CPI = $CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

# AMAT Example

- Compute the overall average memory access time
  - Hit time = 1 clock cycle in both I-Cache and D-Cache
  - Miss penalty = 50 clock cycles for I-Cache and D-Cache
  - I-Cache misses = 3.8 misses per 1000 instructions
  - D-Cache misses = 41 misses per 1000 instructions
  - Load + Store frequency = 25%
- Solution:

Combined Misses per Instruction = (3.8 + 41) / 1000 = 0.0448

Combined Miss rate (per access) = 0.0448 / (1 + 0.25) = 0.03584

Overall AMAT = 1 + 0.03584 × 50 = 2.792 cycles

# Cache Performance: Single Level L1 Princeton (Unified) Memory Architecture

*(Ignoring Write Policy)*

CPUtime = Instruction count x CPI x Clock cycle time

$CPI_{execution}$ = CPI with ideal memory

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

Mem Stall cycles per instruction =

Memory accesses per instruction x Memory stall cycles per access

Assuming no stall cycles on a cache hit (cache access time = 1 cycle, stall = 0)

Cache Hit Rate = H1 Miss Rate = 1- H1

**i.e No hit penalty**

Memory stall cycles per memory access = Miss rate x Miss penalty = **(1- H1 ) x M**

AMAT = 1 + Miss rate x Miss penalty = **1 + (1- H1) x M**

Memory accesses per instruction = ( 1 + fraction of loads/stores)

Miss Penalty = M = the number of stall cycles resulting from missing in cache

= Main memory access time - 1

Thus for a unified L1 cache with no stalls on a cache hit:

CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x (1 - H1) x M

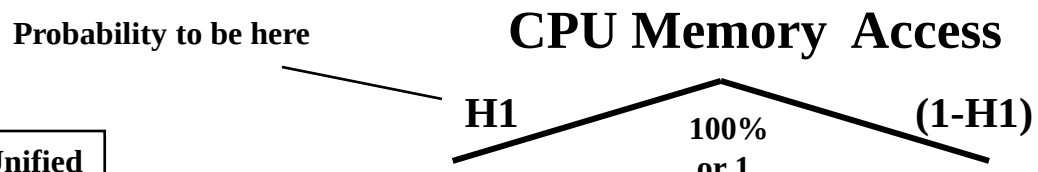
AMAT = 1 + (1 - H1) x M

**CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access**

**= $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT - 1)**

# Memory Access Tree:For Unified Level 1 Cache

*(Ignoring Write Policy)*




Unified $L_1$

**L1 Hit:**
% = Hit Rate = H1
Hit Access Time = 1
Stall cycles per access = 0
Stall= H1 x 0 = 0
( No Stall)

**L1 Miss:**
% = (1- Hit rate) = (1-H1)
Access time = M + 1
Stall cycles per access = M
Stall = M x (1-H1)

**Assuming:**
Ideal access on a hit

Hit Rate | Hit Time | Miss Rate | Miss Time

$$AMAT = H1 \times 1 + (1 - H1) \times (M + 1) = 1 + M \times (1 - H1)$$

Stall Cycles Per Access = AMAT - 1 = M x (1 -H1)

$$CPI = CPI_{execution} + (1 + \text{fraction of loads/stores}) \times M \times (1 - H1)$$

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

AMAT = 1 + Stalls per average memory access

# Cache Performance Example

- Suppose a CPU executes at Clock Rate = 200 MHz (5 ns per cycle) with a single level of cache.
- $CPI_{execution}$ = 1.1
- Instruction mix: 50% arith/logic, 30% load/store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of M= 50 cycles.

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction =

Mem accesses per instruction x Miss rate x Miss penalty

Mem accesses per instruction = 1 + 0.3 = 1.3

Mem Stalls per memory access = (1- H1) x M = 0.015 x 50 = .75 cycles

AMAT = 1 +.75 = 1.75 cycles

Mem Stalls per instruction = 1.3 x 0.015 x 50 = 0.975

CPI = 1.1 + .975 = 2.075

The ideal memory CPU with no misses is 2.075/1.1 = 1.88 times faster

# Cache Performance Example

- Suppose for the previous example we double the clock rate to 400 MHz, how much faster is this machine, assuming similar miss rate, instruction mix?
- Since memory speed is not changed, the miss penalty takes more CPU cycles:

Miss penalty = M = 50 x 2 = 100 cycles.

CPI = 1.1 + 1.3 x .015 x 100 = 1.1 + 1.95 = 3.05

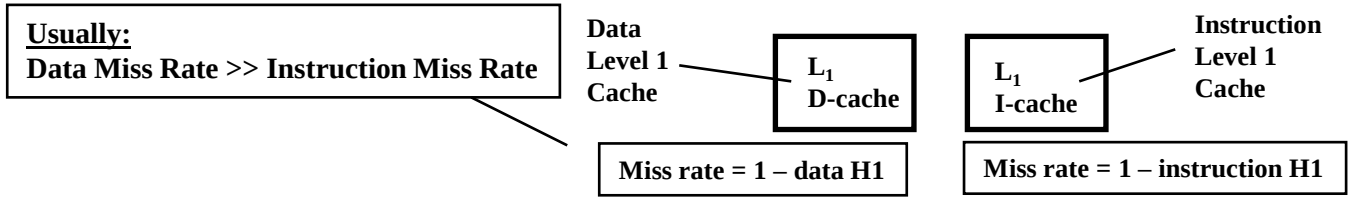
Speedup = $(CPI_{old} \times C_{old}) / (CPI_{new} \times C_{new})$

= 2.075 x 2 / 3.05 = 1.36

The new machine is only 1.36 times faster rather than 2 times faster due to the increased effect of cache misses.

→ *CPUs with higher clock rate, have more cycles per cache miss and more memory impact on CPI.*

# Cache Performance: Single Level L1 Harvard (Split) Memory Architecture




- For a CPU with separate or split level one (L1) caches for instructions and data (Harvard memory architecture) and no stalls for cache hits:

CPUtime = Instruction count x CPI x Clock cycle time

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

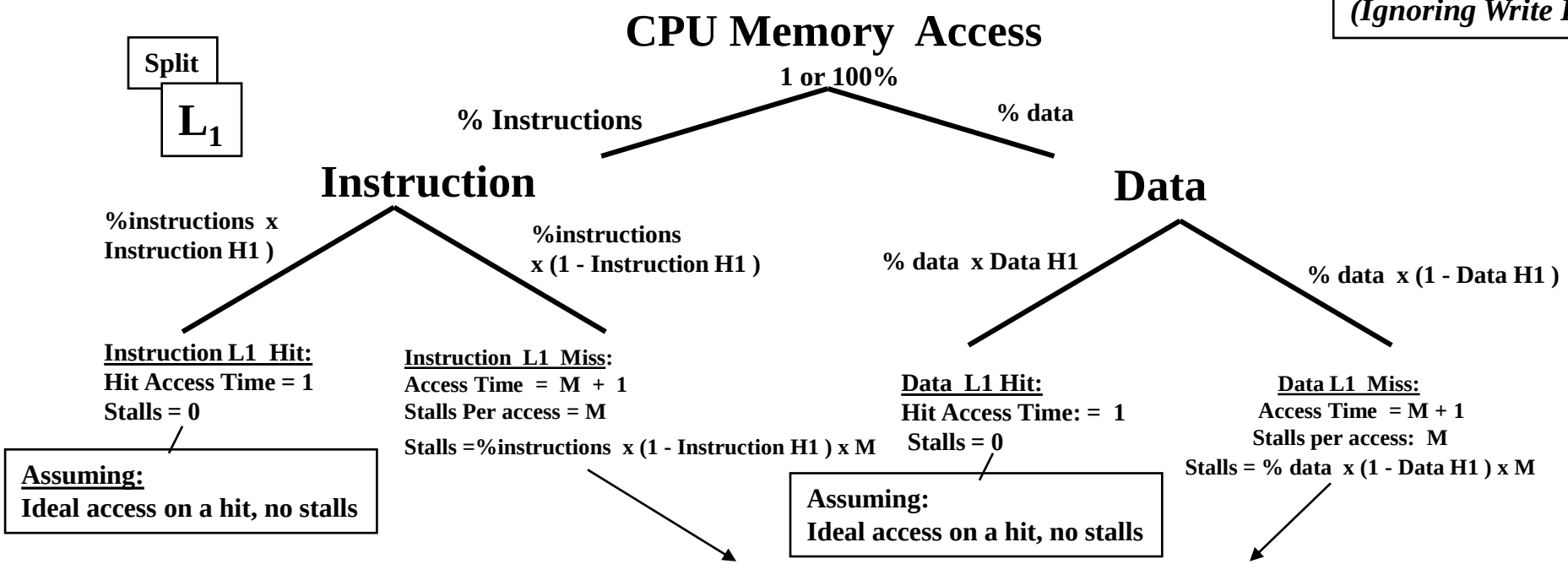
Mem Stall cycles per instruction =

Instruction Fetch Miss rate x M +

Data Memory Accesses Per Instruction x Data Miss Rate x M

# Memory Access Tree For Separate Level 1 Caches

*(Ignoring Write Policy)*




**Stall Cycles Per Access = % Instructions x ( 1 - Instruction H1 ) x M + % data x (1 - Data H1 ) x M**

**AMAT = 1 + Stall Cycles per access**

**Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access**

**CPI = $CPI_{execution}$ + Stall cycles per instruction**
**= $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access**

**M = Miss Penalty = stall cycles per access resulting from missing in cache**
**M + 1 = Miss Time = Main memory access time**
**Data H1 = Level 1 Data Hit Rate** **1- Data H1 = Level 1 Data Miss Rate**
**Instruction H1 = Level 1 Instruction Hit Rate** **1- Instruction H1 = Level 1 Instruction Miss Rate**
**% Instructions = Percentage or fraction of instruction fetches out of all memory accesses**
**% Data = Percentage or fraction of data accesses out of all memory accesses**

# Split L1 Cache Performance Example

- Suppose a CPU uses separate level one (L1) caches for instructions and data (Harvard memory architecture) with different miss rates for instruction and data access:
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
  - **$CPI_{execution}$ = 1.1**
  - **Instruction mix: 50% arith/logic, 30% load/store, 20% control**
  - **Assume a cache miss rate of 0.5% for instruction fetch and a cache data miss rate of 6%.**
  - **A cache hit incurs no stall cycles while a cache miss incurs 200 stall cycles for both memory reads and writes.**
- Find the resulting stalls per access, AMAT and CPI using this cache?

**CPI = $CPI_{execution}$ + mem stalls per instruction**

Memory Stall cycles per instruction = Instruction Fetch Miss rate x Miss Penalty +
Data Memory Accesses Per Instruction x Data Miss Rate x Miss Penalty

Memory Stall cycles per instruction = 0.5/100 x 200 + 0.3 x 6/100 x 200 = 1 + 3.6 = 4.6 cycles
Stall cycles per average memory access = 4.6/1.3 = 3.54 cycles
AMAT = 1 + 3.54 = 4.54 cycles

CPI = $CPI_{execution}$ + mem stalls per instruction = 1.1 + 4.6 = 5.7 cycles

- What is the miss rate of a single level unified cache that has the same performance?

4.6 = 1.3 x Miss rate x 200 which gives a miss rate of 1.8 % for an equivalent unified cache

- How much faster is the CPU with ideal memory?

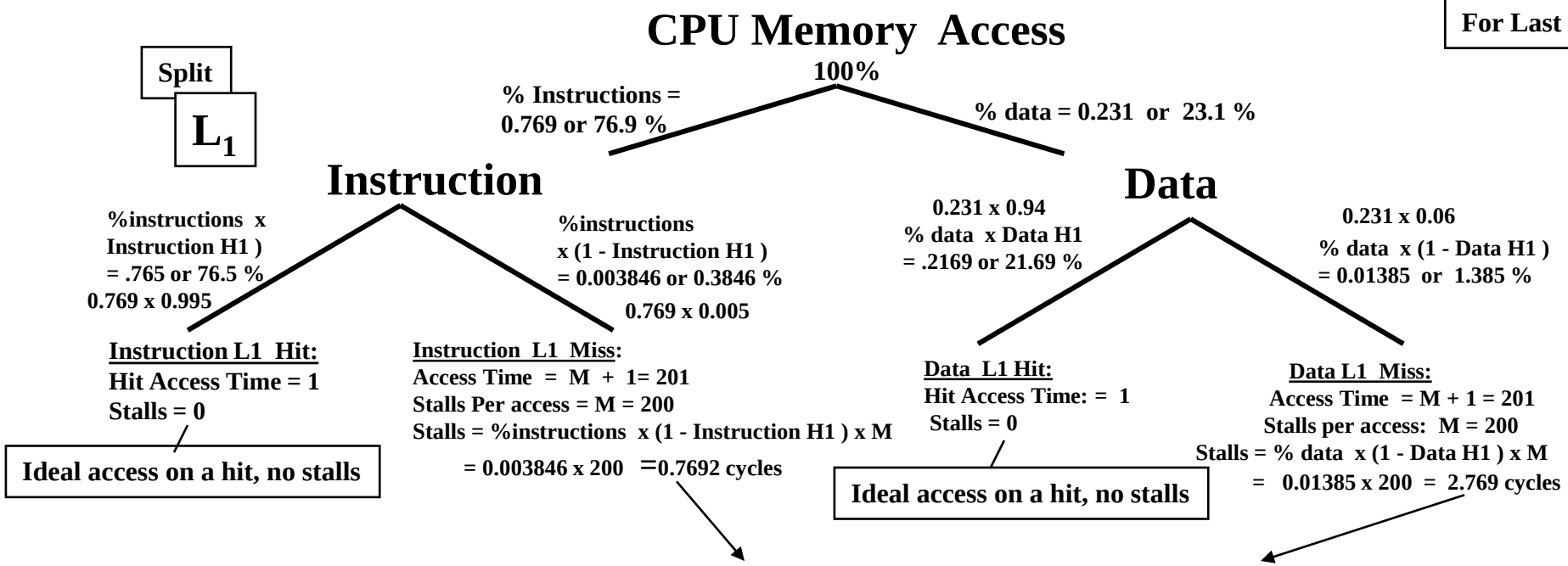
The CPU with ideal cache (no misses) is 5.7/1.1 = 5.18 times faster
With no cache at all the CPI would have been = 1.1 + 1.3 X 200 = 261.1 cycles !!

# Memory Access Tree For Separate Level 1 Caches Example

**30% of all instructions executed are loads/stores, thus:**
**Fraction of instruction fetches out of all memory accesses = 1/ (1+0.3) = 1/1.3 = 0.769 or 76.9 %**
**Fraction of data accesses out of all memory accesses = 0.3/ (1+0.3) = 0.3/1.3 = 0.231 or 23.1 %**

***(Ignoring Write Policy)***

**For Last Example**




**Stall Cycles Per Access = % Instructions x ( 1 - Instruction H1 ) x M + % data x (1 - Data H1 ) x M**
**= 0.7692 + 2.769 = 3.54 cycles**

**AMAT = 1 + Stall Cycles per access = 1 + 3.5 = 4.54 cycles**

**Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 3.54 = 4.6 cycles**

**CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 4.6 = 5.7**

**M = Miss Penalty = stall cycles per access resulting from missing in cache = 200 cycles**
**M + 1 = Miss Time = Main memory access time = 200+1 =201 cycles L1 access Time = 1 cycle**
**Data H1 = 0.94 or 94% 1- Data H1 = 0.06 or 6%**
**Instruction H1 = 0.995 or 99.5% 1- Instruction H1 = 0.005 or 0.5 %**
**% Instructions = Percentage or fraction of instruction fetches out of all memory accesses = 76.9 %**
**% Data = Percentage or fraction of data accesses out of all memory accesses = 23.1 %**

# Cache Write Miss Policy

- Since data is usually not needed immediately on a write miss two options exist on a cache write miss:

**Write Allocate:** ***(Bring old block to cache then update it)***

The missed cache block is loaded into cache on a write miss followed by write hit actions.

**i.e A cache block frame is allocated for the block to be modified (written-to)**

**No-Write Allocate:**

**i.e A cache block frame is not allocated for the block to be modified (written-to)**

The block is modified in the lower level (lower cache level, or main memory) and not loaded (written or updated) into cache.

***While any of the above two write miss policies can be used with either write back or write through:***

- **Write back caches always use write allocate to capture subsequent writes to the block in cache.**
- **Write through caches usually use no-write allocate since subsequent writes still have to go to memory.**

**Cache Write Miss = Block to be modified is not in cache**
**Allocate = Allocate or assign a cache block frame for written data**

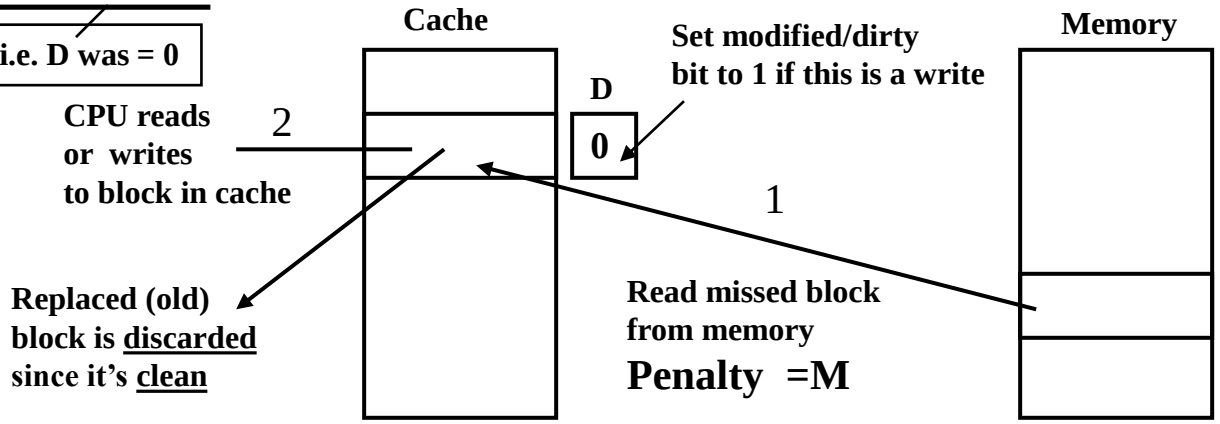
# Write Back Cache With Write Allocate: Cache Miss Operation

## Block to be replaced is clean

(read or write miss)

i.e. D was = 0

Miss Penalty = M

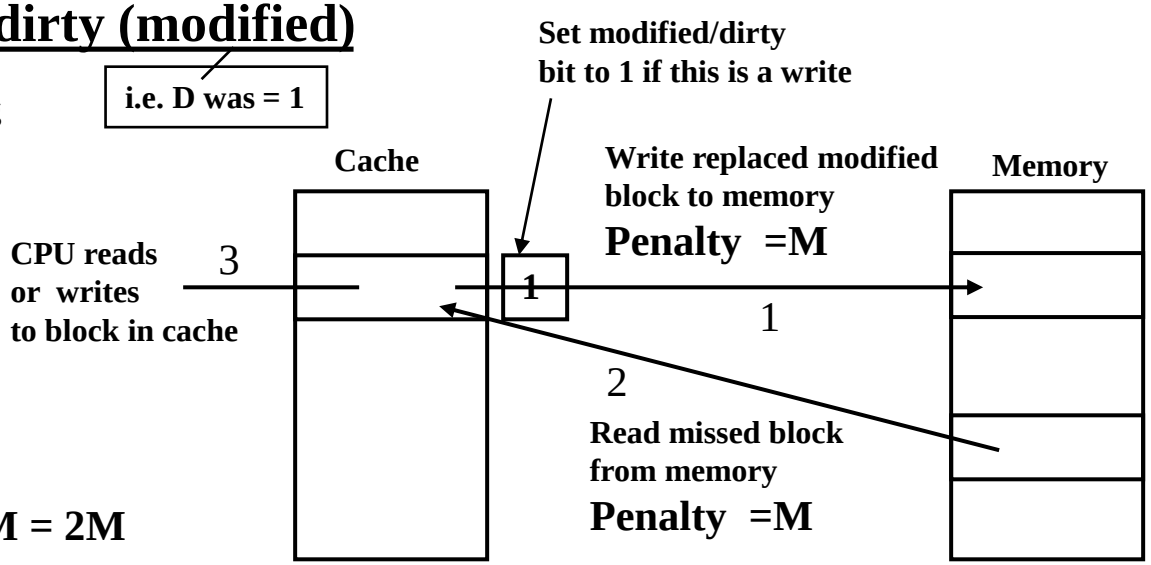



## Block to be replaced is dirty (modified)

i.e. D was = 1

1 Write back modified block being replaced to memory
Penalty =M

2 Read missed block from memory
Penalty =M



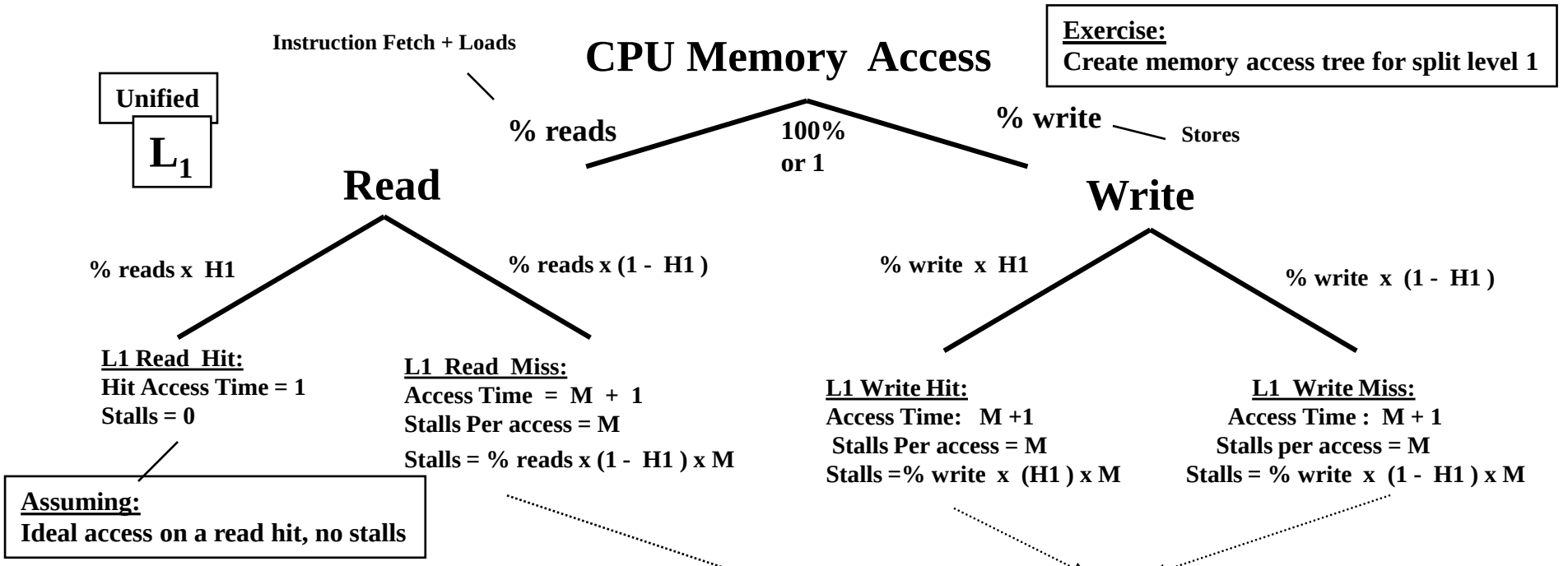

**Thus:**

**Total Miss Penalty = M + M = 2M**

**M = Miss Penalty = stall cycles per access resulting from missing in cache**

# Memory Access Tree, Unified $L_1$

## Write Through, No Write Allocate, No Write Buffer




**Stall Cycles Per Memory Access = % reads x (1 - H1 ) x M + % write x M**

**AMAT = 1 + % reads x (1 - H1 ) x M + % write x M**

**CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access**

**Stall Cycles per access = AMAT - 1**

**M = Miss Penalty**
**H1 = Level 1 Hit Rate**
**1- H1 = Level 1 Miss Rate**

**M = Miss Penalty = stall cycles per access resulting from missing in cache**
**M + 1 = Miss Time = Main memory access time**
**H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate**

# Reducing Write Stalls For Write Though Cache Using Write Buffers

- To reduce write stalls when write though is used, a write buffer is used to eliminate or reduce write stalls:
  - **Perfect write buffer: All writes are handled by write buffer, no stalling for writes**
  - **In this case (for unified L1 cache):**

    Stall Cycles Per Memory Access = % reads x (1 - H1 ) x M

    (i.e No stalls at all for writes)
  - **Realistic Write buffer: A percentage of write stalls are not eliminated when the write buffer is full.**
  - **In this case (for unified L1 cache):**

    Stall Cycles/Memory Access = ( % reads x (1 - H1 ) + % write stalls not eliminated ) x M

# Write Through Cache Performance Example

- A CPU with $CPI_{execution}$ = 1.1 Mem accesses per instruction = 1.3
- Uses a unified L1 Write Through, No Write Allocate, with:
  1. **No write buffer.**
  2. **Perfect Write buffer**
  3. **A realistic write buffer that eliminates 85% of write stalls**
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

CPI = $CPI_{execution}$ + mem stalls per instruction

% reads = 1.15/1.3 = 88.5% % writes = .15/1.3 = 11.5%

1. **With No Write Buffer :** **Stall on all writes**

**Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 11.5%) = 8.33 cycles**

**CPI = 1.1 + 8.33 = 9.43**

2. **With Perfect Write Buffer (all write stalls eliminated):**

**Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5%) = 0.86 cycles**

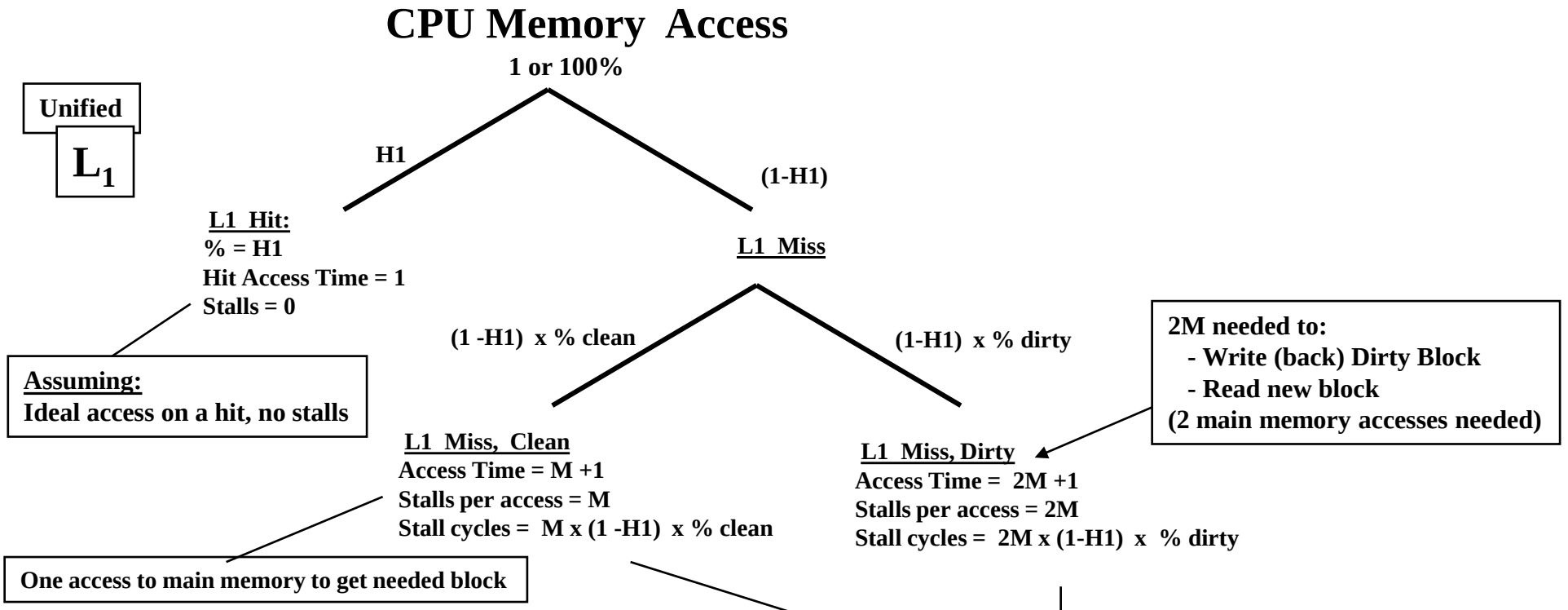
**CPI = 1.1 + 0.86 = 1.96**

3. **With Realistic Write Buffer (eliminates 85% of write stalls)**

**Mem Stalls/ instruction = 1.3 x 50 x (88.5% x 1.5% + 15% x 11.5%) = 1.98 cycles**

**CPI = 1.1 + 1.98 = 3.08**

# Memory Access Tree Unified $L_1$ Write Back, With Write Allocate




**Stall Cycles Per Memory Access = (1-H1) x ( M x % clean + 2M x % dirty )**

**AMAT = 1 + Stall Cycles Per Memory Access**

**CPI = $CPI_{execution}$ + (1 + fraction of loads/stores) x Stall Cycles per access**

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
H1 = Level 1 Hit Rate 1- H1 = Level 1 Miss Rate

# Write Back Cache Performance Example

- A CPU with $CPI_{execution}$ = 1.1 uses a unified L1 with with write back, with write allocate, and the probability a cache block is dirty = 10%
- Instruction mix: 50% arith/logic, 15% load, 15% store, 20% control
- Assume a cache miss rate of 1.5% and a miss penalty of 50 cycles.

CPI = $CPI_{execution}$ + mem stalls per instruction

Mem Stalls per instruction =

Mem accesses per instruction x Stalls per access

Mem accesses per instruction = 1 + 0.3 = 1.3

Stalls per access = (1-H1) x ( M x % clean + 2M x % dirty )

Stalls per access = 1.5% x (50 x 90% + 100 x 10%) = 0.825 cycles

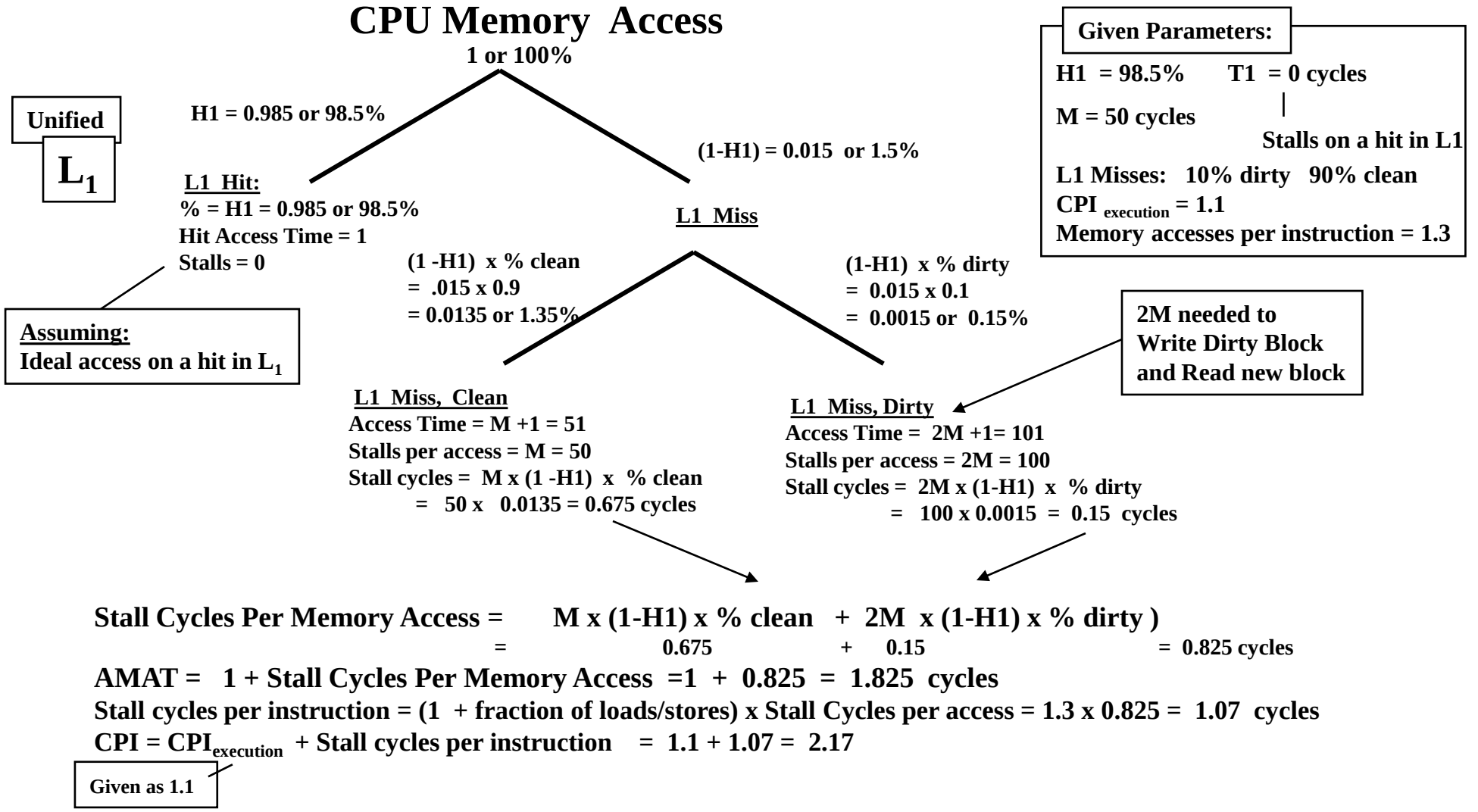
AMAT = 1 + stalls per access = 1 + 0.825 = 1.825 cycles

Mem Stalls per instruction = 1.3 x 0.825 = 1.07 cycles

CPI = 1.1 + 1.07 = 2.17

The ideal CPU with no misses is 2.17/1.1 = 1.97 times faster

# Memory Access Tree For Unified $L_1$
# Write Back, With Write Allocate Example




**Stall Cycles Per Memory Access = ( M x (1-H1) x % clean + 2M x (1-H1) x % dirty )**

**= 0.675 + 0.15 = 0.825 cycles**

**AMAT = 1 + Stall Cycles Per Memory Access =1 + 0.825 = 1.825 cycles**

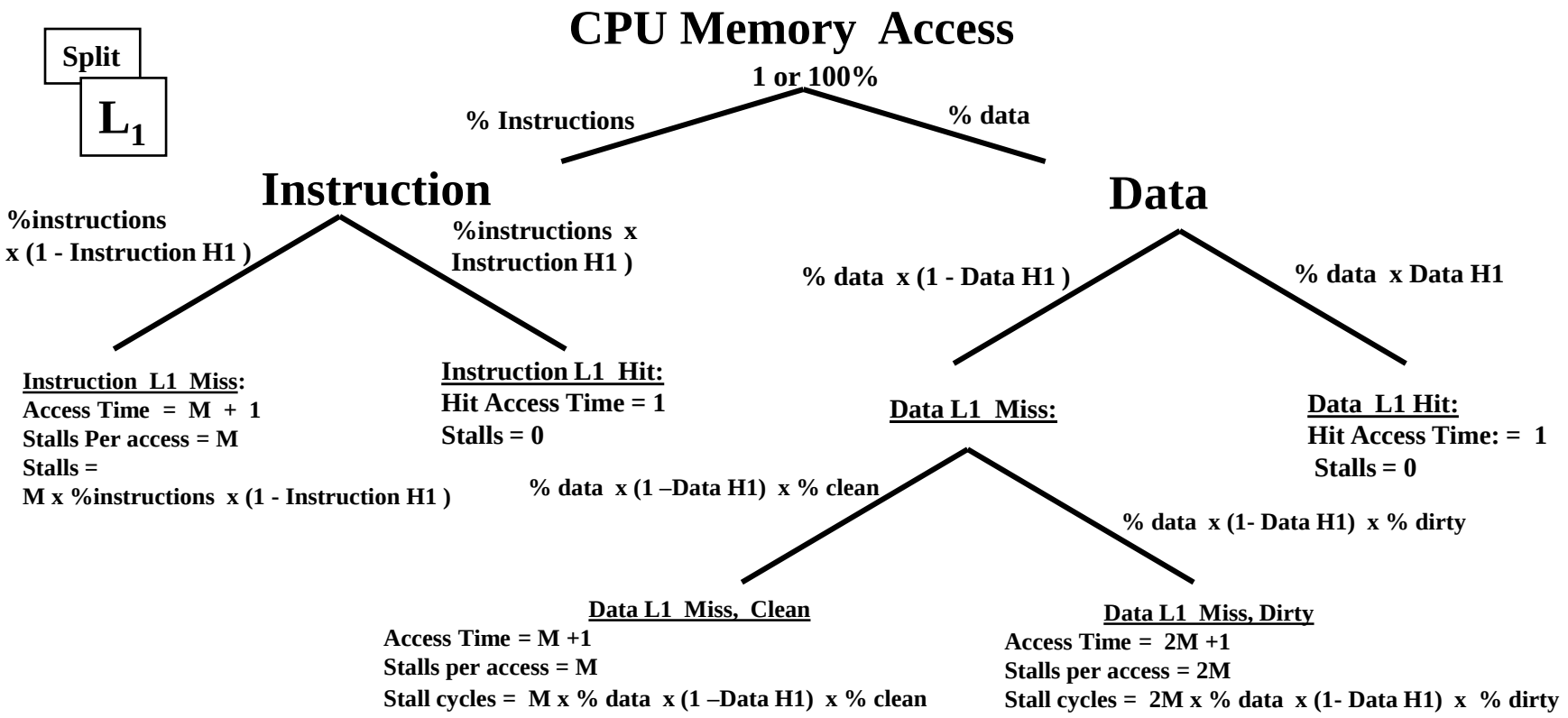
**Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access = 1.3 x 0.825 = 1.07 cycles**

**CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 1.07 = 2.17**

Given as 1.1

**M = Miss Penalty = 50 cycles**
**M + 1 = Miss Time = 50 + 1 = 51 cycles L1 access Time = 1 cycle**
**H1 = 0.985 or 98.5% 1- H1 = 0.015 or 1.5%**

# Memory Access Tree Structure
# For Separate Level 1 Caches, Write Back, With Write Allocate

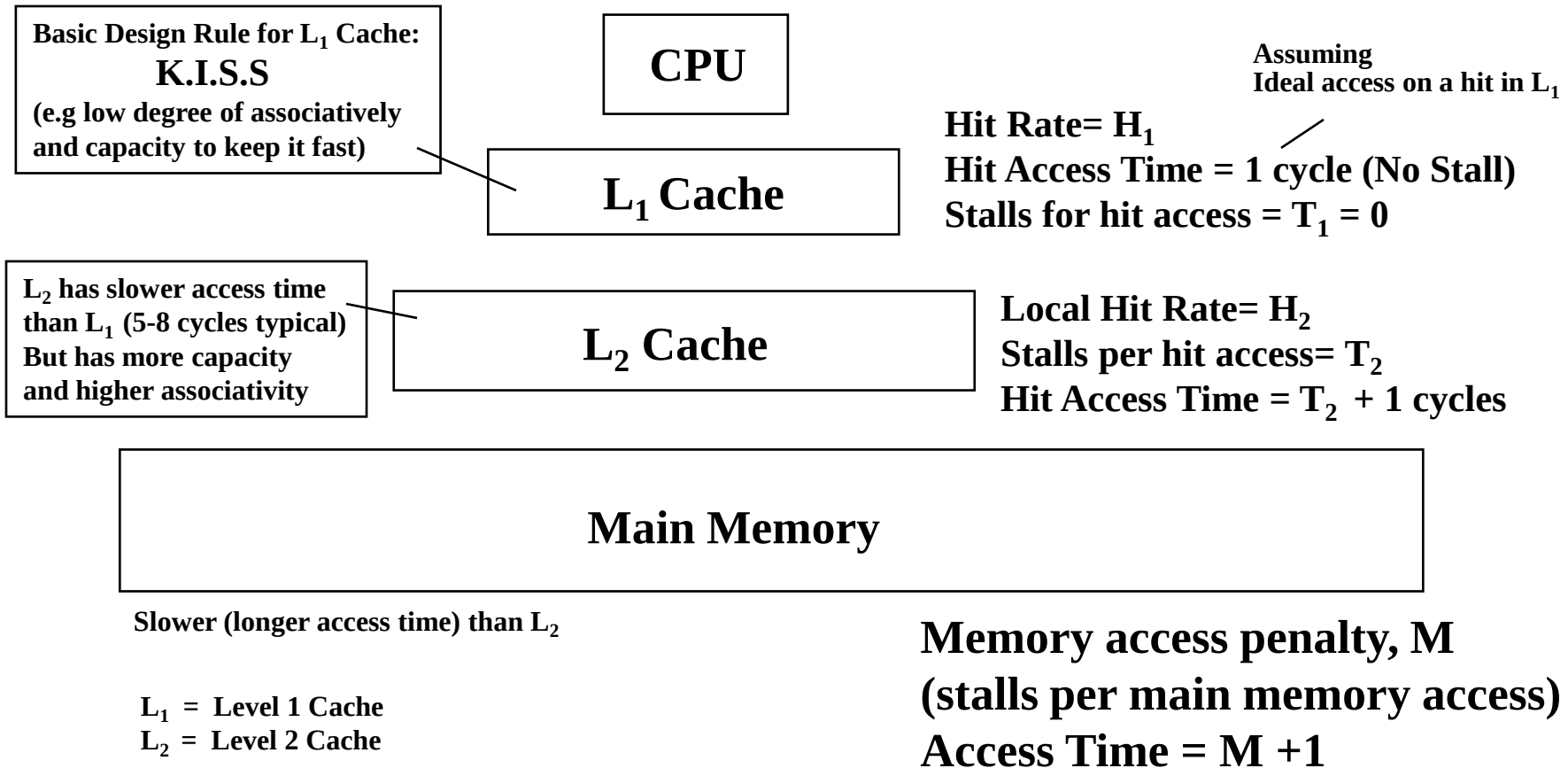



**Exercise: Find expression for: Stall cycles per average memory access, AMAT**

M = Miss Penalty = stall cycles per access resulting from missing in cache
M + 1 = Miss Time = Main memory access time
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses
% Clean = Percentage or fraction of data L1 misses that are clean
% Dirty = Percentage or fraction of data L1 misses that are dirty = 1 - % Clean

**Assuming:**
Ideal access on a hit in $L_1$

# 2 Levels of Cache: $L_1$, $L_2$




**Goal of multi-level Caches:**

**Reduce the effective miss penalty incurred by level 1 cache misses by using additional levels of cache that capture some of these misses.**

**Thus hiding more main memory latency and reducing AMAT further**

# Miss Rates For Multi-Level Caches

- Local Miss Rate: This rate is the number of misses in a cache level divided by the number of memory accesses to this level (i.e those memory accesses that reach this level).

  Local Hit Rate = 1 - Local Miss Rate

- Global Miss Rate: The number of misses in a cache level divided by the total number of memory accesses generated by the CPU.
- Since level 1 receives all CPU memory accesses, for level 1:

  **Local Miss Rate = Global Miss Rate = 1 - H1**

- For level 2 since it only receives those accesses missed in 1:

  **Local Miss Rate = Miss rate$_{L2}$ = 1- H2**

  **Global Miss Rate = Miss rate$_{L1}$ x Local Miss rate$_{L2}$**

  **= (1- H1) x (1 - H2)**

# 2-Level Cache (Both Unified) Performance (Ignoring Write Policy)

CPUtime = IC x (CPI$_{execution}$ + Mem Stall cycles per instruction) x C

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

- For a system with 2 levels of unified cache, assuming no penalty when found in $L_1$ cache: **($T_1 = 0$)**

Stall cycles per memory access =

[miss rate $L_1$] x [ Hit rate $L_2$ x Hit time $L_2$

+ Miss rate $L_2$ x Memory access penalty ] =

(1-H1) x H2 x T2 + (1-H1)(1-H2) x M

**L1 Miss, L2 Hit**

**Here we assume T1 =0 (no stall on L1 hit)**

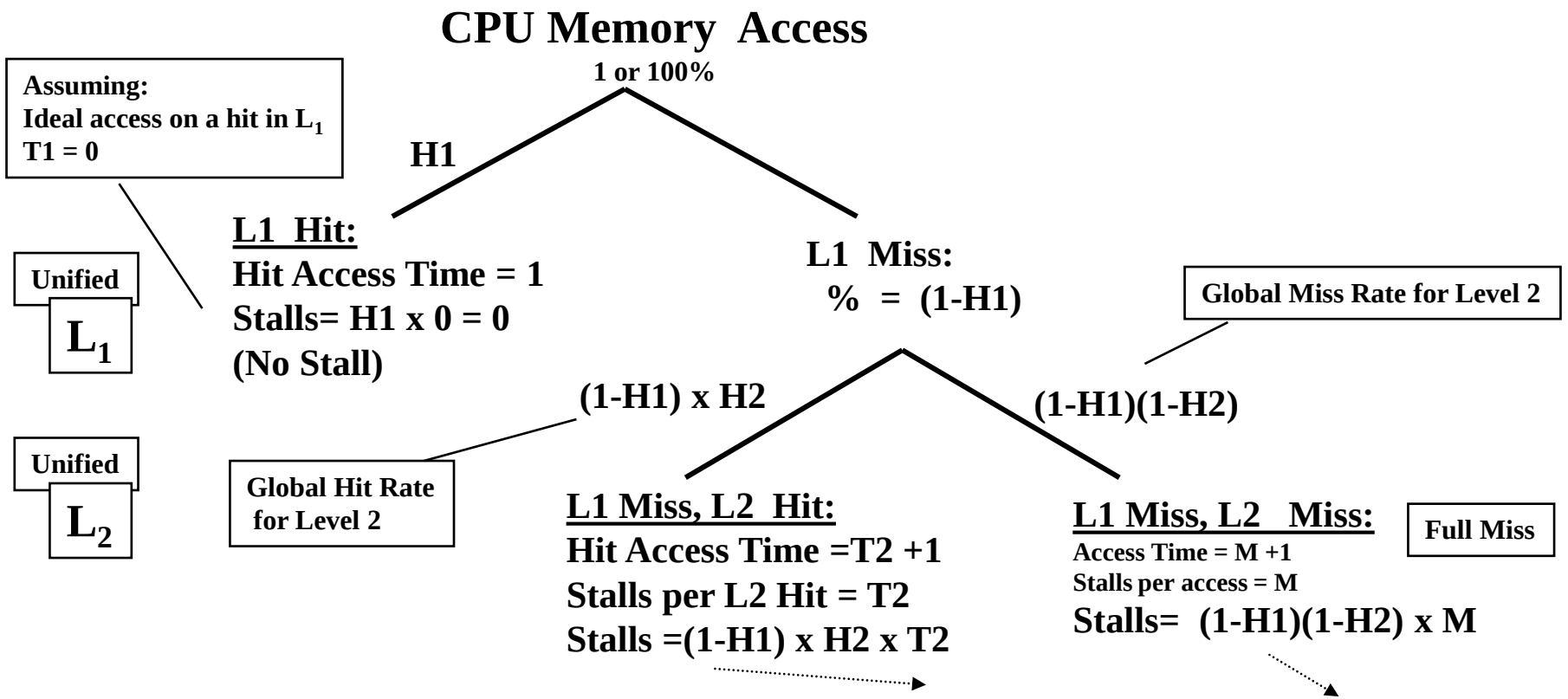
**H1 = L1 Hit Rate**
**T1 = stall cycles per L1 access hit**
**H2 = Local L2 Hit Rate**
**T2 =stall cycles per L2 access hit**

**L1 Miss, L2 Miss: Must Access Main Memory**

**Full Miss**

**CPI = CPI$_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access**
**= CPI$_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)**

# 2-Level Cache (Both Unified) Performance Memory Access Tree (Ignoring Write Policy) CPU Stall Cycles Per Memory Access




**Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M**

**AMAT = 1 + (1-H1) x H2 x T2 + (1-H1)(1-H2) x M**

$CPI = CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access

$= CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

# Unified Two-Level Cache Example

*(Ignoring Write Policy)*

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction.
- With two levels of cache (both unified)
- $L_1$ hit access time = 1 cycle (no stall on a hit, T1= 0), a miss rate of 5%
- $L_2$ hit access time = 3 cycles (T2= 2 stall cycles per hit) with local miss rate 40%,
- Memory access penalty, M = 100 cycles (stalls per access). Find CPI ...

CPI = $CPI_{execution}$ + Mem Stall cycles per instruction

With No Cache, CPI = 1.1 + 1.3 x 100 = 131.1

With single $L_1$, CPI = 1.1 + 1.3 x .05 x 100 = 7.6

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M

= 0.05 x .6 x 2 + 0.05 x 0.4 x 100

= 0.06 + 2 = 2.06 cycles

AMAT = 2.06 + 1 = 3.06 cycles

Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

= 2.06 x 1.3 = 2.678 cycles

CPI = 1.1 + 2.678 = 3.778

Speedup = 7.6/3.778 = 2

**CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access**
**= $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)**

# Memory Access Tree For 2-Level Cache (Both Unified) Example CPU Stall Cycles Per Memory Access

*(Ignoring Write Policy)*

**CPU Memory Access**

1 or 100%

H1 = 0.95 or 95%

Unified $L_1$

**L1 Hit:**
Hit Access Time = 1
Stalls per L1 Hit = T1 = 0
Stalls= H1 x 0 = 0
(No Stall)

Ideal access on a hit in $L_1$
T1 = 0

**L1 Miss:**
(1-H1)= 0.05 or 5%

(1-H1) x H2
= 0.05 x 0.6
= 0.03 or 3%

Unified $L_2$

Global Hit Rate for $L_2$

**L1 Miss, L2 Hit:**
Hit Access Time =T2 +1 = 3 cycles
Stalls per L2 Hit = T2 = 2 cycles
Stalls =(1-H1) x H2 x T2
= 0.03 x 2 = 0.06 cycles

(1-H1)(1-H2)
= 0.05 x 0.4
= 0.02 or 2%

Global Miss Rate for $L_2$

**L1 Miss, L2 Miss:**
Access Time = M +1 = 100 + 1 =101 cycles
Stalls per access = M = 100 cycles
Stalls= (1-H1)(1-H2) x M
= 0.02 x 100 = 2 cycles

Full Miss

**Given Parameters:**
H1 = 95% T1 = 0 cycles
H2 = 60% T2 = 2 cycles
M = 100 cycles
Stalls on a hit
$CPI_{execution}$ = 1.1
Memory accesses per instruction = 1.3

Stall cycles per memory access = (1-H1) x H2 x T2 + (1-H1)(1-H2) x M
= 0.06 + 2 = 2.06 cycles

AMAT = 1 + Stall cycles per memory access = 1 + 2.06 = 3.06 cycles

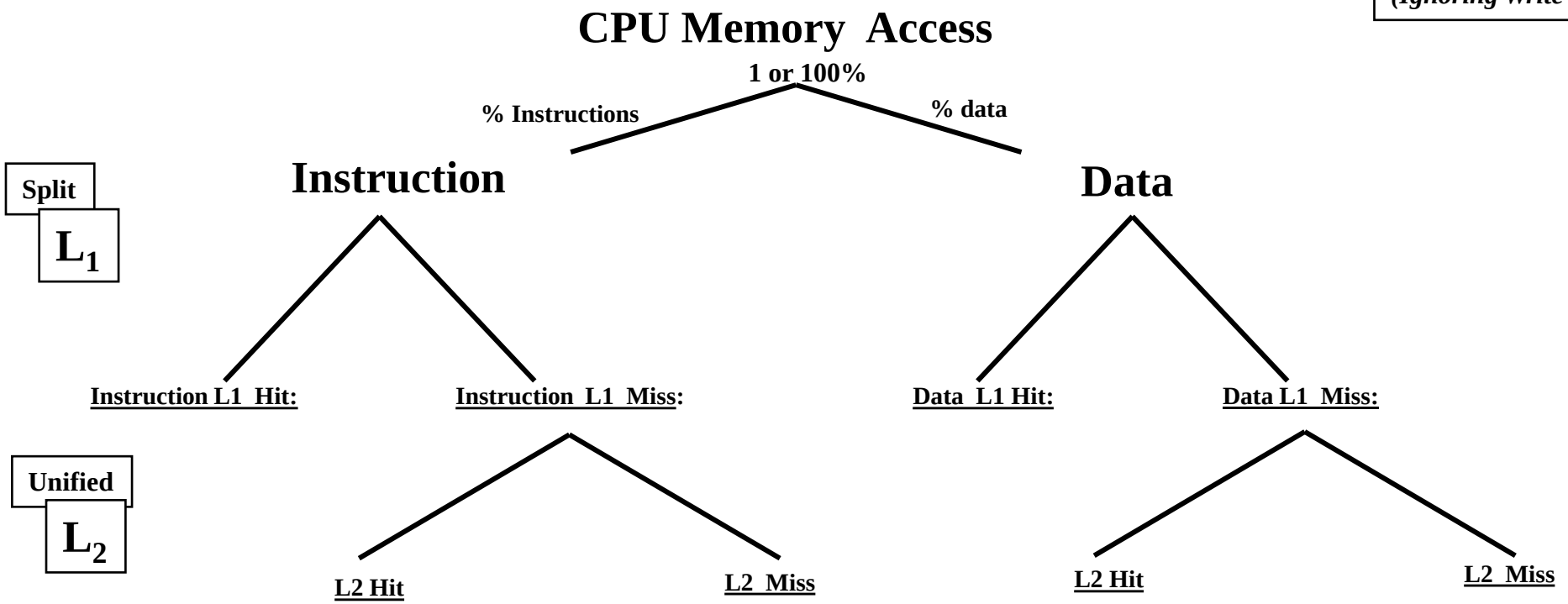
Stall cycles per instruction = (1 + fraction of loads/stores) x Stall Cycles per access
= 1.3 x 2.06 = 2.678 cycles

CPI = $CPI_{execution}$ + Stall cycles per instruction = 1.1 + 2.678 = 3.778

$CPI = CPI_{execution}$ + (1 + fraction of loads and stores) x stall cycles per access
$= CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT – 1)

# Memory Access Tree Structure For 2-Level Cache (Separate Level 1 Caches, Unified Level 2)

*(Ignoring Write Policy)*




% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

For L1: T1 = Stalls per hit access to level 1
Data H1 = Level 1 Data Hit Rate 1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate 1- Instruction H1 = Level 1 Instruction Miss Rate
For L2: T2 = Stalls per access to level 2
H2 = Level 2 local hit Rate 1-H2 = Level 2 local miss rate

M = Miss Penalty = stall cycles per access resulting from missing in cache level 2
M + 1 = Miss Time = Main memory access time

**Exercise:** In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access

# Common Write Policy For 2-Level Cache

$L_1$

- Write Policy For Level 1 Cache:
  - **Usually Write through to Level 2.** (not write through to main memory just to L2)
  - **Write allocate is used to reduce level 1 read misses.**
  - **Use write buffer to reduce write stalls to level 2.**

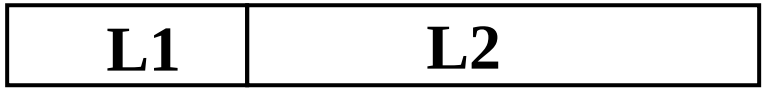
$L_2$

- Write Policy For Level 2 Cache:
  - **Usually write back with write allocate is used.**
    - To minimize memory bandwidth usage.

- The above 2-level cache write policy results in inclusive L2 cache since the content of L1 is also in L2
  - Common in the majority of all CPUs with 2-levels of cache
  - As opposed to exclusive L1, L2 (e.g AMD Athlon XP, A64)

*As if we have a single level of cache with one portion (L1) is faster than remainder (L2)*

**i.e what is in L1 is not duplicated in L2**

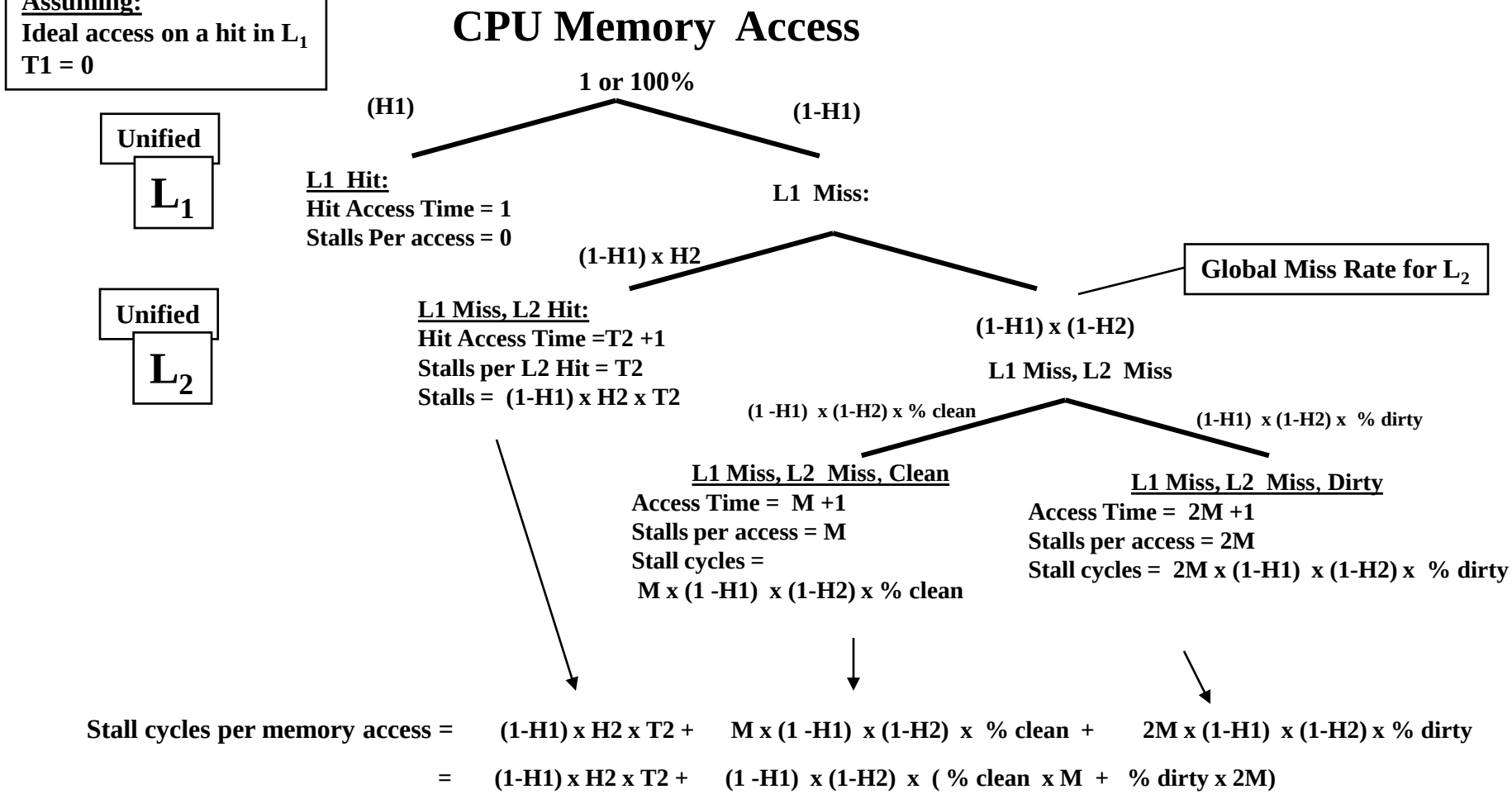


# 2-Level (Both Unified) Memory Access Tree

## L1: Write Through to L2, Write Allocate, With Perfect Write Buffer

## L2: Write Back with Write Allocate

**Assuming:**
**Ideal access on a hit in $L_1$**
**T1 = 0**




**Stall cycles per memory access = (1-H1) x H2 x T2 + M x (1 -H1) x (1-H2) x % clean + 2M x (1-H1) x (1-H2) x % dirty**

**= (1-H1) x H2 x T2 + (1 -H1) x (1-H2) x ( % clean x M + % dirty x 2M)**

**AMAT = 1 + Stall Cycles Per Memory Access**
**CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x Stall Cycles per access**

# Two-Level (Both Unified) Cache Example With Write Policy

- CPU with $CPI_{execution}$ = 1.1 running at clock rate = 500 MHz
- 1.3 memory accesses per instruction. Two levels of cache (both unified)
- For $L_1$:
  - **Cache operates at 500 MHz (no stall on L1 Hit, T1 =0) with a miss rate of 1-H1 = 5%**
  - **Write though to $L_2$ with perfect write buffer with write allocate**
- For $L_2$:
  - **Hit access time = 3 cycles (T2= 2 stall cycles per hit) local miss rate 1- H2 = 40%**
  - **Write back to main memory with write allocate**
  - **Probability a cache block is dirty = 10%**
- Memory access penalty, M = 100 cycles.
- Create memory access tree and find, stalls per memory access, AMAT, CPI.
- Stall cycles per memory access = (1-H1) x H2 x T2 +

  (1 -H1) x (1-H2) x ( % clean x M + % dirty x 2M)

  = .05 x .6 x 2 + .05 x .4 x ( .9 x 100 + .1 x200)

  = .06 + 0.02 x 110 = .06 + 2.2 = 2.26

- AMAT = 2.26 + 1 = 3.26 cycles

  Mem Stall cycles per instruction = Mem accesses per instruction x Stall cycles per access

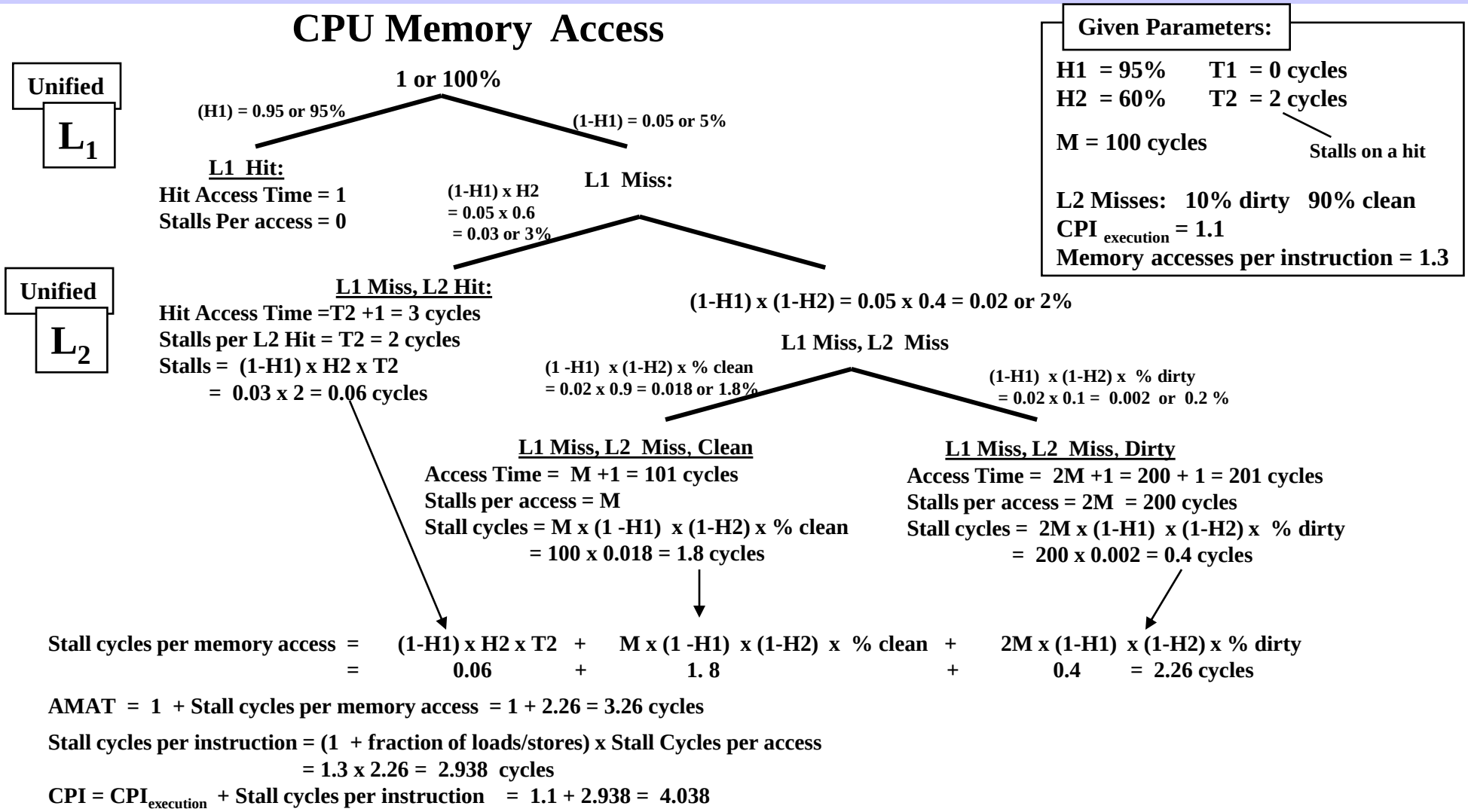
  = 2.26 x 1.3 = 2.938 cycles

  CPI = 1.1 + 2.938 = 4.038 = 4

**CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x (AMAT –1)**

# Memory Access Tree For Two-Level (Both Unified) Cache Example With Write Policy

## L1: Write Through to L2, Write Allocate, With Perfect Write Buffer
## L2: Write Back with Write Allocate

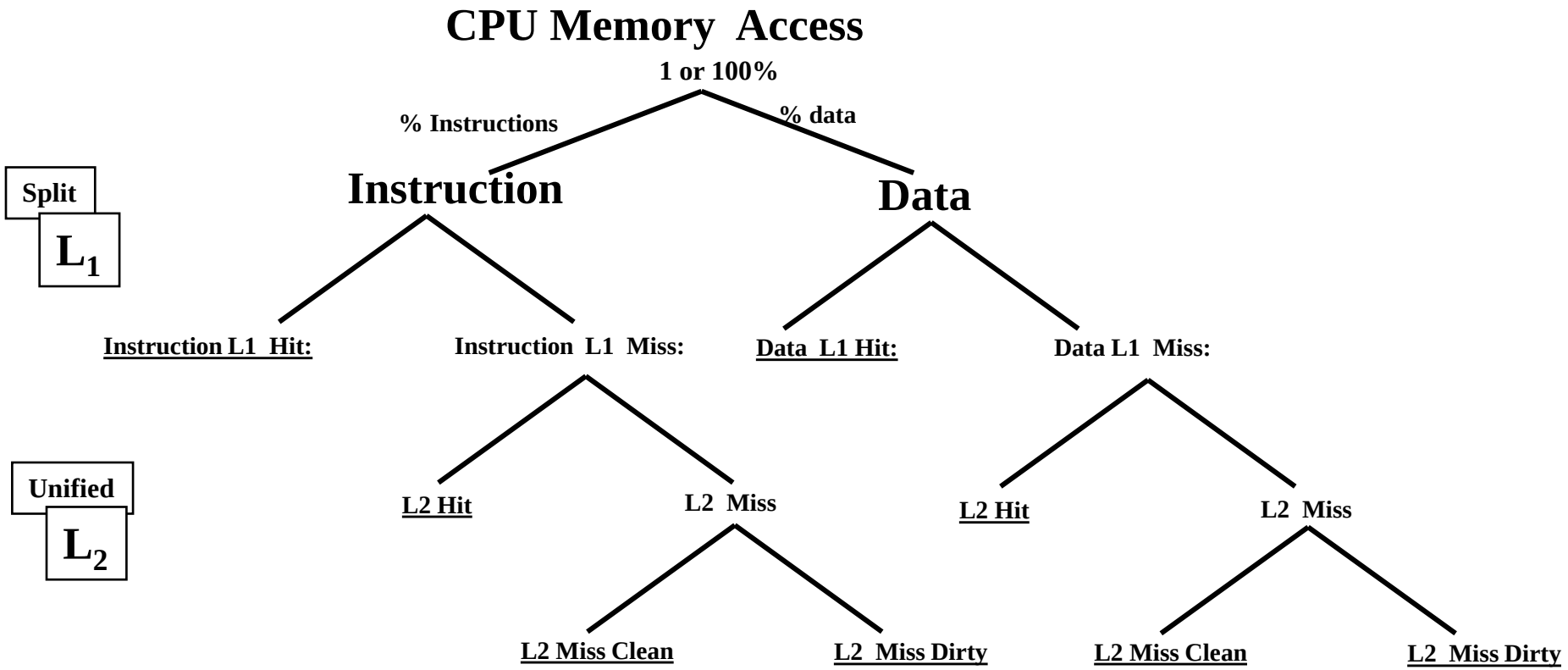



**AMAT = 1 + Stall Cycles Per Memory Access**

**CPI = $CPI_{execution}$ + (1 + fraction of loads and stores) x Stall Cycles per access**

# Memory Access Tree Structure For 2-Level Cache(Separate Level 1 Caches, Unified Level 2)

## L1: Write Through to L2, Write Allocate, With Perfect Write Buffer  L2: Write Back with Write Allocate




**Exercise:** In terms of the parameters below, complete the memory access tree and find the expression for stall cycles per memory access

% Instructions = Percentage or fraction of instruction fetches out of all memory accesses
% Data = Percentage or fraction of data accesses out of all memory accesses

For L1: T1 = Stalls per hit access to level 1
Data H1 = Level 1 Data Hit Rate  1- Data H1 = Level 1 Data Miss Rate
Instruction H1 = Level 1 Instruction Hit Rate  1- Instruction H1 = Level 1 Instruction Miss Rate
For L2: T2 = Stalls per access to level 2
H2 = Level 2 local hit Rate  1-H2 = Level 2 local miss rate
% Clean = Percentage or fraction of data L2 misses that are clean
% Dirty = Percentage or fraction of L2 misses that are dirty = 1 - % Clean

M = Miss Penalty = stall cycles per access resulting from missing in cache level 2
M = L2 Miss Time = Main memory access time