

Computer Organization and Architecture

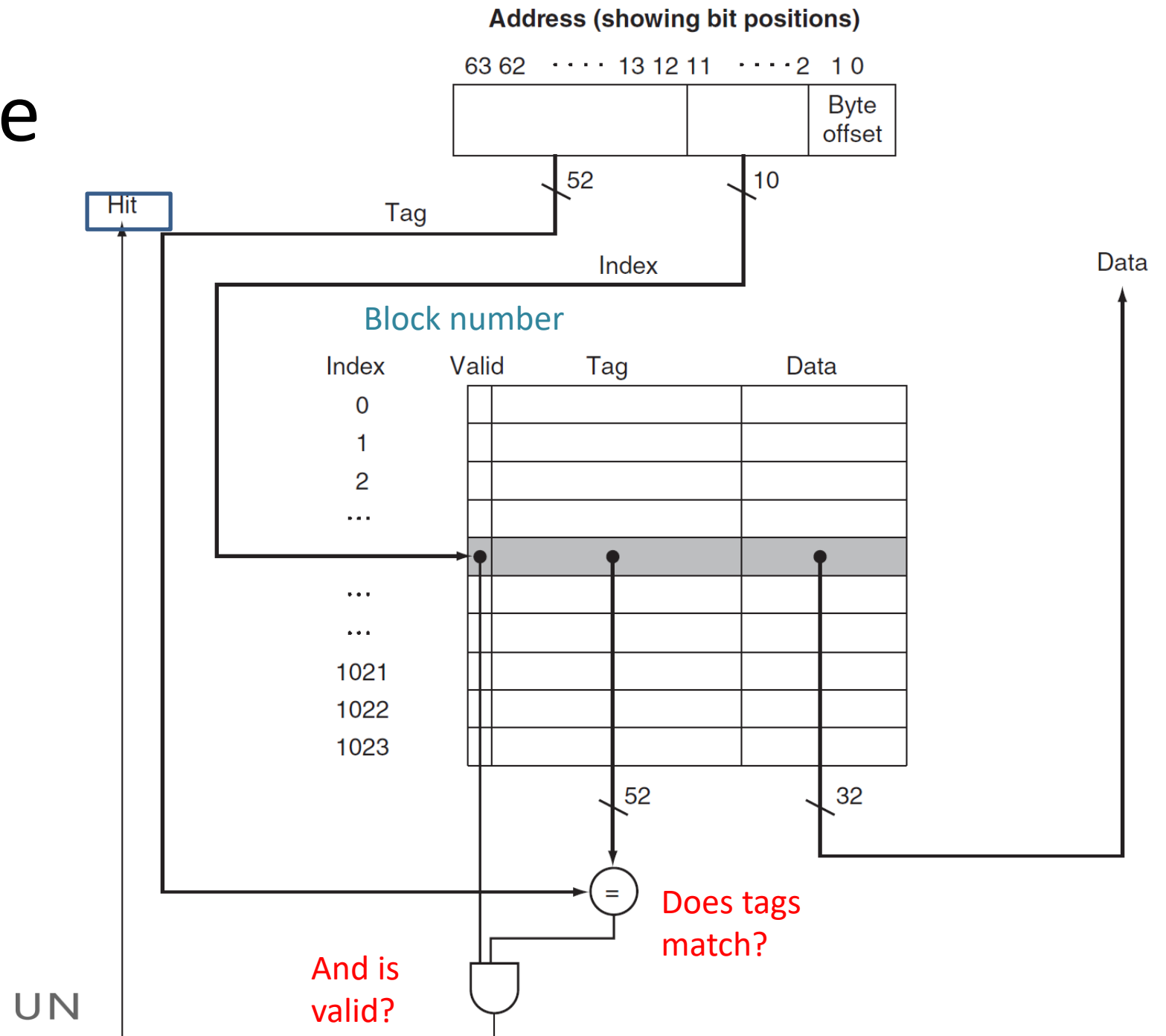
Lecture – 25

Nov 14th , 2022

- Handling Cache Hits
- Handling Cache Misses
- Handling Cache Writes

Cache Hardware

Easy to handle cache hits.
CPU proceeds normally

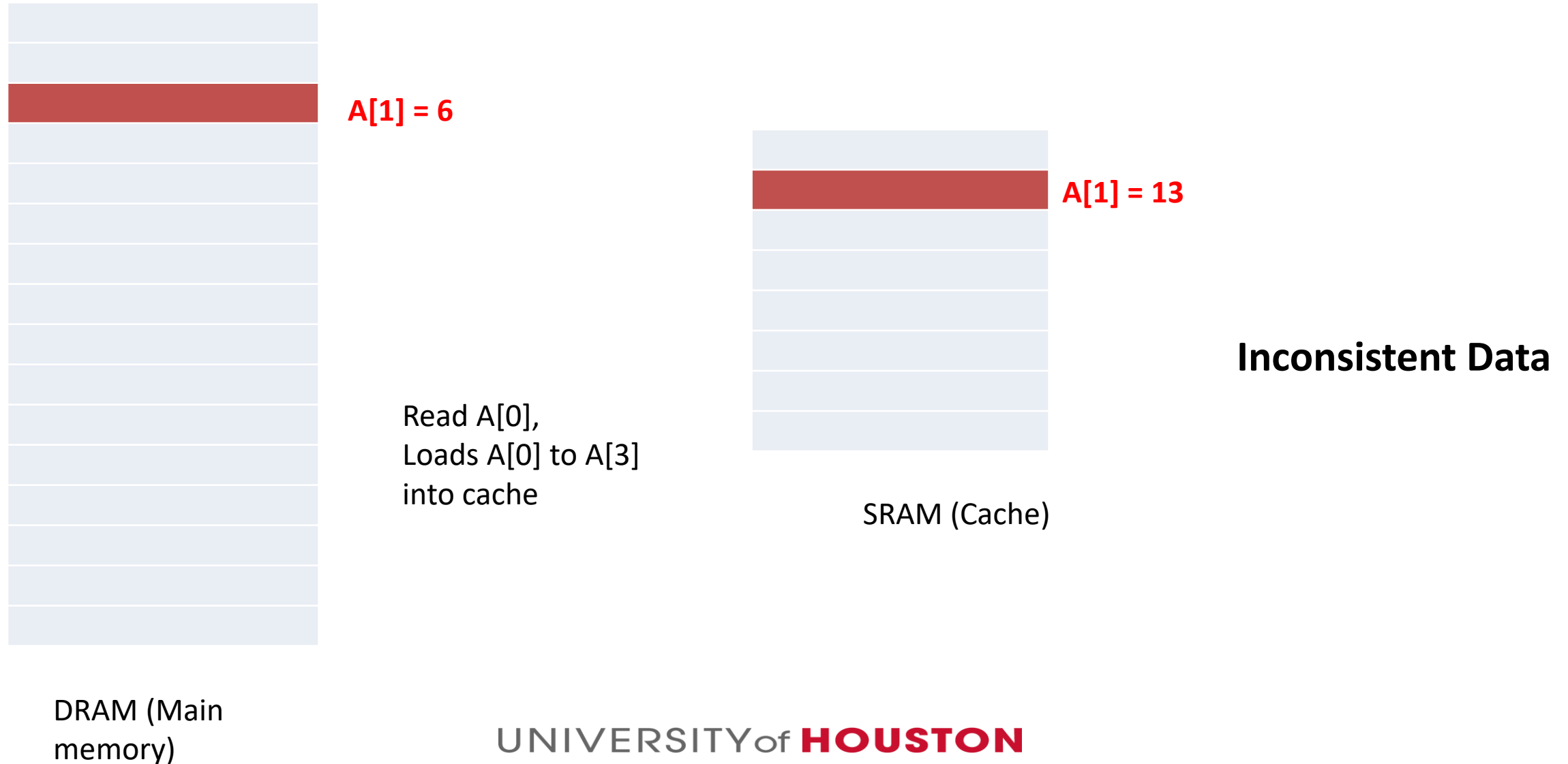


Handling Cache Misses

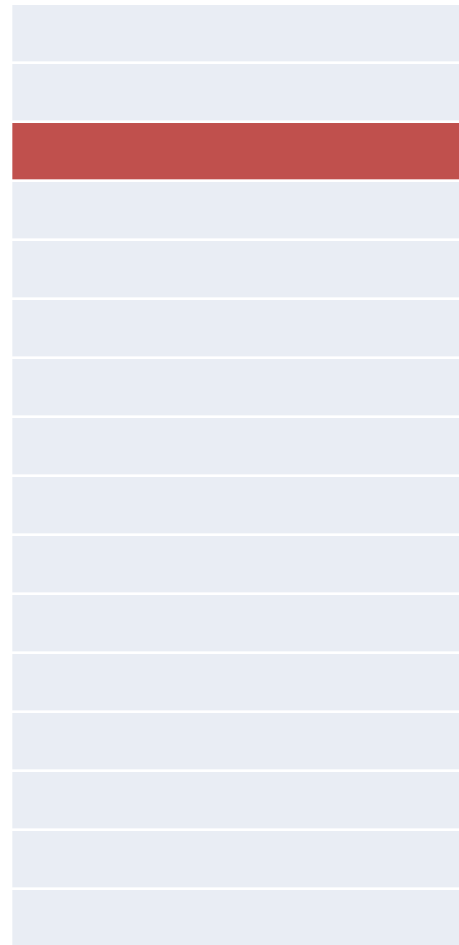
- On cache miss
 - Stall the CPU pipeline
 - Fetch block from next level of hierarchy
- Cache miss can occur in instructions and data access
 - Instruction cache miss
 - Restart instruction fetch
 - Data cache miss
 - Complete data access

Handling Writes

Write Example

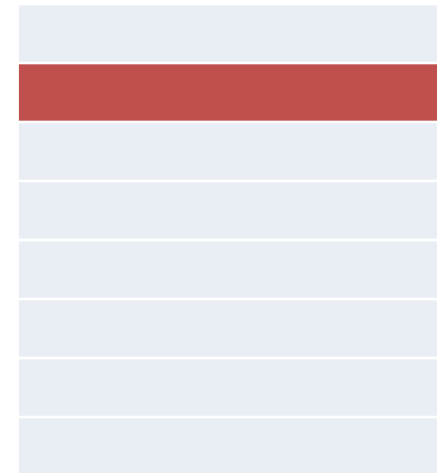


Write Example



A[1] = 13

Read A[0],
Loads A[0] to A[3]
into cache



A[1] = 13

SRAM (Cache)

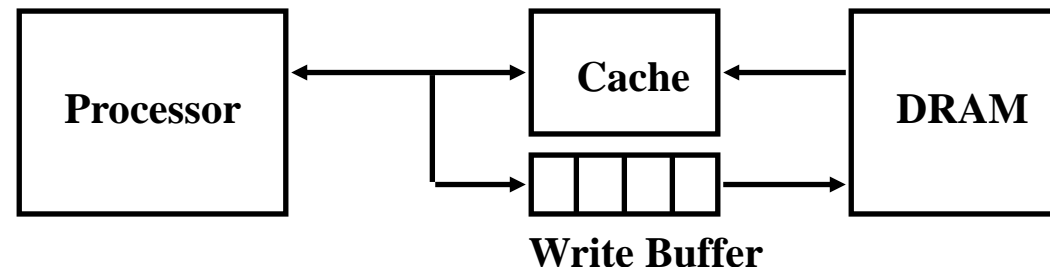
Inconsistent Data
Solution 1: Write to both Cache
and Memory
Write-Through strategy

Does not provide good
performance.
Every write is a write to memory
Each write can take up to 100
cycles

DRAM (Main
memory)

Write-Through

- On data-write hit, could just update the block in cache
 - But then cache and memory would be inconsistent
- Write-through: also update memory
- But makes writes take longer
 - e.g., write to memory takes 100 cycles
- Solution: write buffer
 - Holds data waiting to be written to memory
 - CPU continues immediately
 - Only stalls on write if write buffer is already full



Write-Back

- Alternative: On data-write hit, just update the block in cache
 - Keep track of whether each block is dirty
- When a dirty block is replaced
 - Write it back to memory

Cache Performance

- CPU time:
 - Time spent by CPU executing the program (include cache hits)
 - Time spent by CPU waiting (stalled) for memory (mainly cache misses)

$$\text{CPU time} = \text{cycles} \times \text{cycle time}$$

$$\text{CPU time} = (\text{execution cycles} + \text{mem stall cycles}) \times \text{cycle time}$$

- Memory stall cycles

- Either from read or write

$$\text{Mem stall cycles} = \text{Read stall cycles} + \text{write stall cycles}$$

- Miss penalty → cycles to fetch data from main memory into cache
- Miss rate → ratio of misses to total memory access (hits+misses)

$$\text{miss rate} = \frac{\text{no.of misses}}{\text{no.of mem access}}$$


Memory stall cycles= ?

Memory Stall Cycles

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$



Can be both read and write misses.

Average Access Time

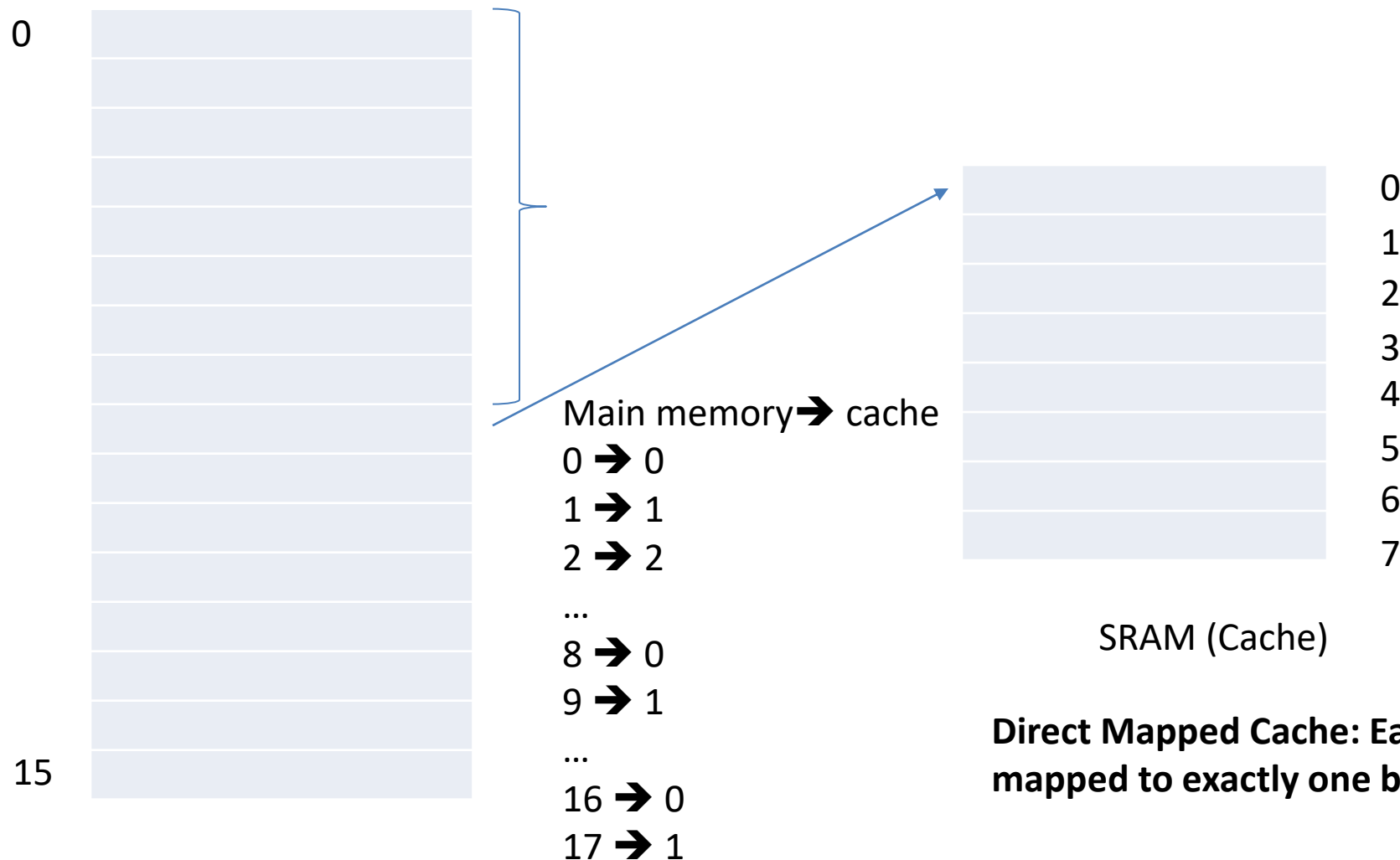
- Hit time is also important for performance
- Average memory access time (AMAT)
 - $\text{AMAT} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
- Example
 - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, l-cache miss rate = 5%
 - $\text{AMAT} = 1 + 0.05 \times 20 = 2\text{ns}$
 - 2 cycles per instruction

Reducing Cache Misses

Types of cache misses

- Compulsory Misses: first access to a block cannot be in the cache (cold start misses)
- Capacity Misses: cache cannot contain all blocks required for the execution
- Conflict Misses: cache block has to be discarded because of block replacement

Example



Fetch 9:

Not available in cache, so fetch
from main memory

Temporal locality:

Retain the latest accessed ones
and replace the oldest one

SRAM (Cache)

**Direct Mapped Cache: Each address in main memory is
mapped to exactly one block**

DRAM (Main
memory)

Associative Caches

- Direct Mapped Cache: Each address in main memory is mapped to exactly one block
- Each address in main memory can be mapped to
 - a set of cache blocks (Set associative)
 - any block (Full Associative)

Full Associative

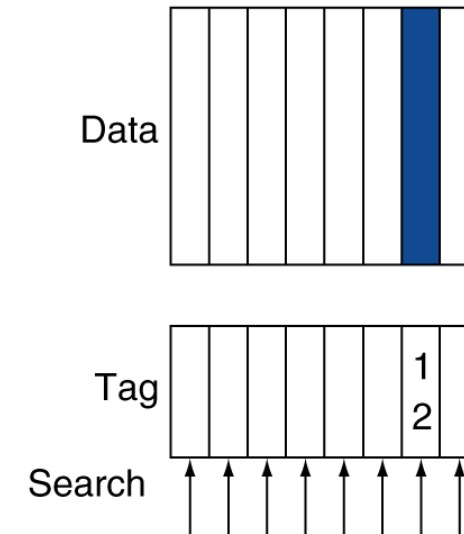
- Fully associative
 - Allow a given block to go in any cache entry

Associative Cache Example

Direct mapped



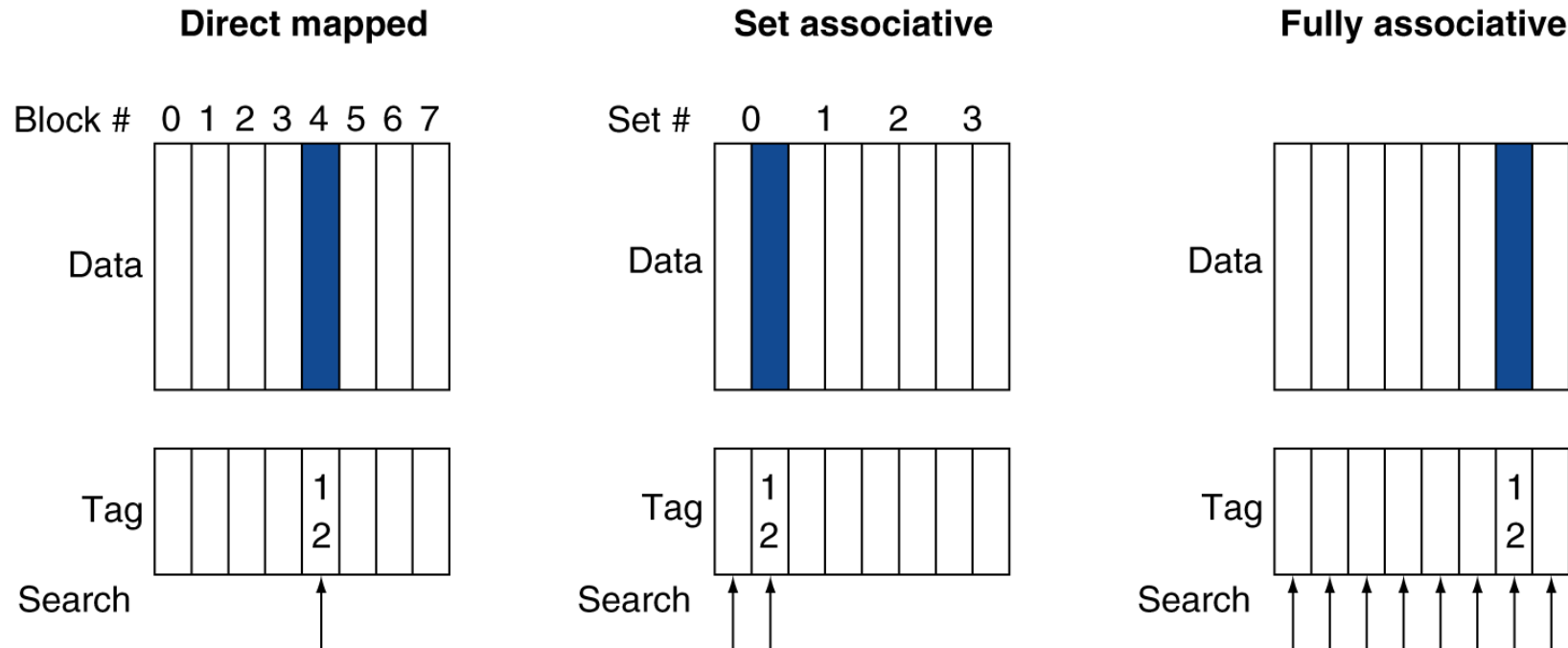
Fully associative



n -way set associative

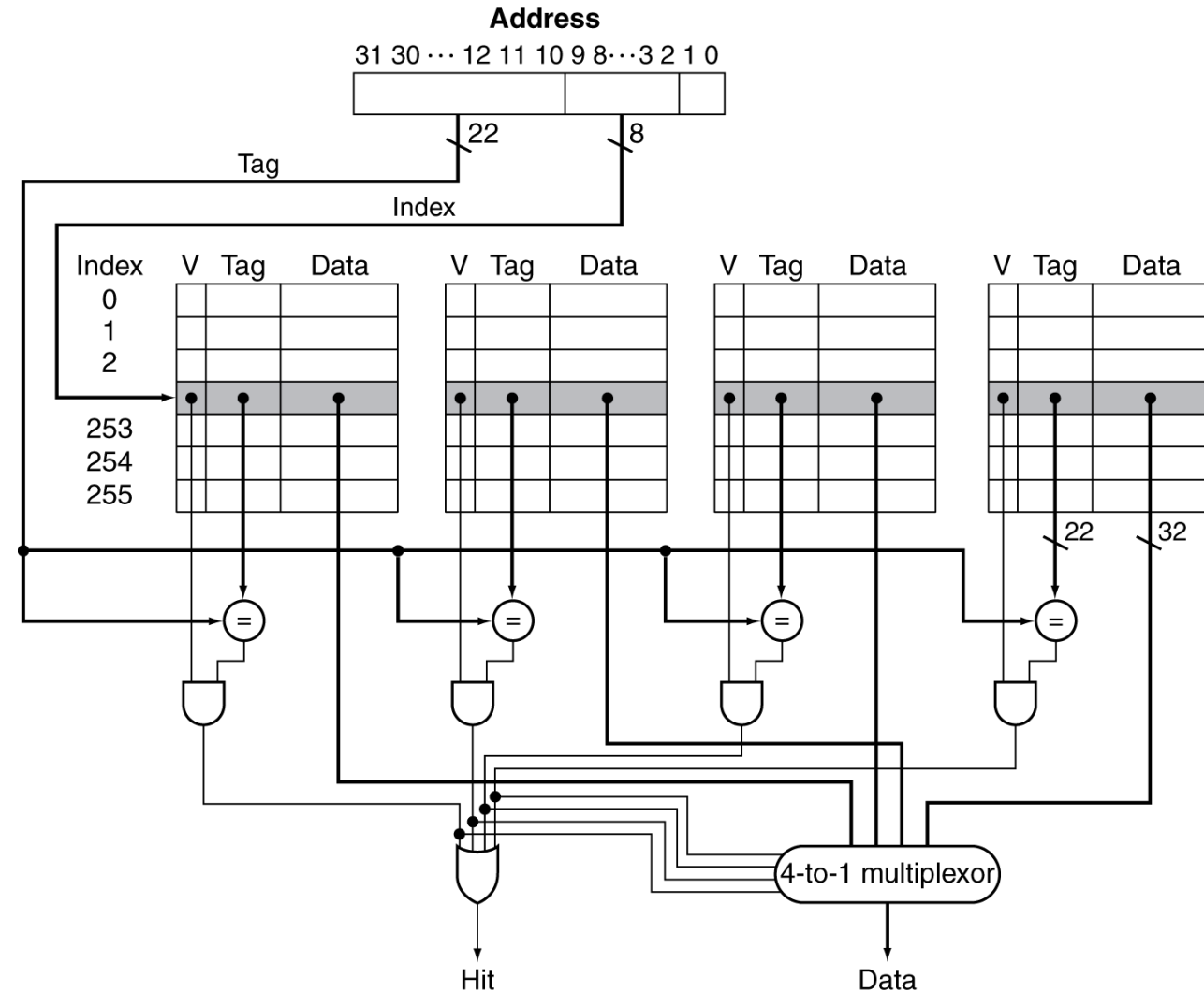
- Each set contains n entries
- Address determines which set
 - (Address) modulo (#Sets in cache)
- Search all entries in a given set at once
- n comparators (less expensive)

Associative Cache Example



Set Associative Cache Organization

Locating a block



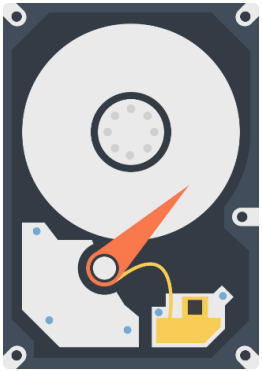
Replacement Policy

- Direct mapped: no choice
- Set associative
 - Prefer non-valid entry, if there is one
 - Otherwise, choose among entries in the set
- Least-recently used (LRU)
 - Choose the one unused for the longest time
 - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
 - Gives approximately the same performance as LRU for high associativity

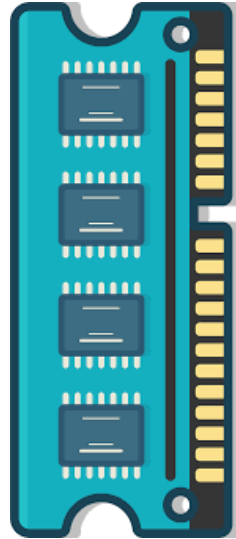
Reducing Cache Penalty

Hierarchy of memories

Stored in memory
HDD



RAM – Random
access memory
D(ynamic)RAM



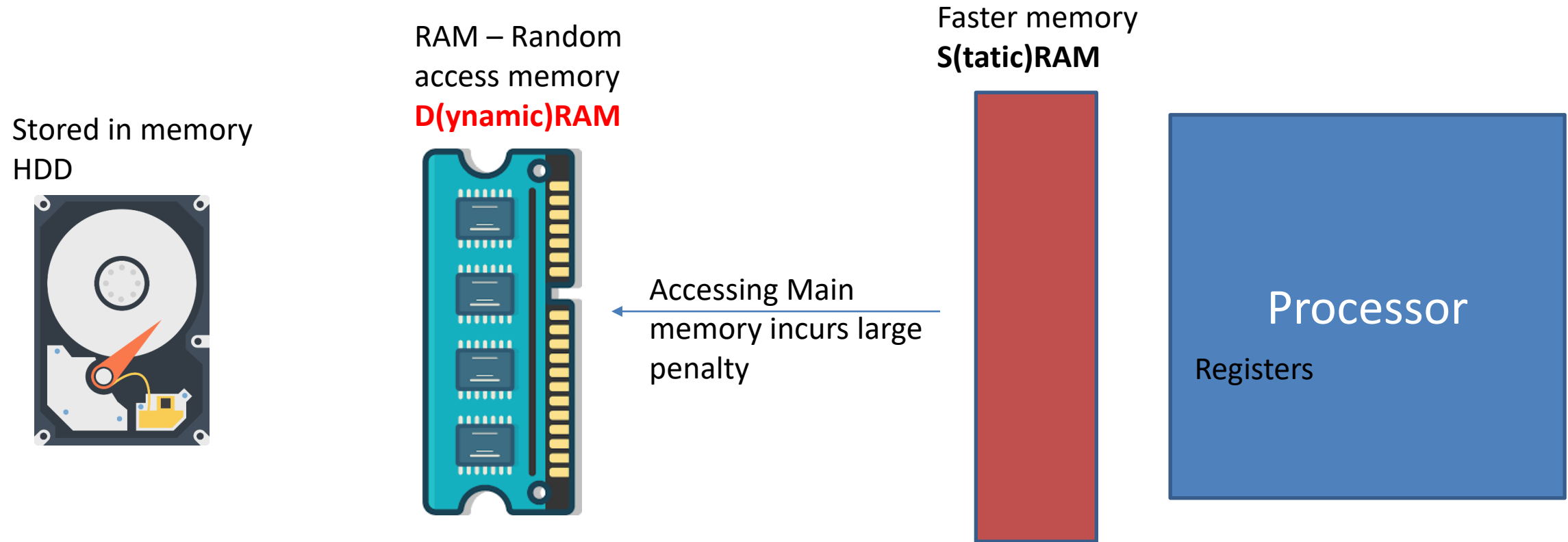
Faster memory
S(tatic)RAM



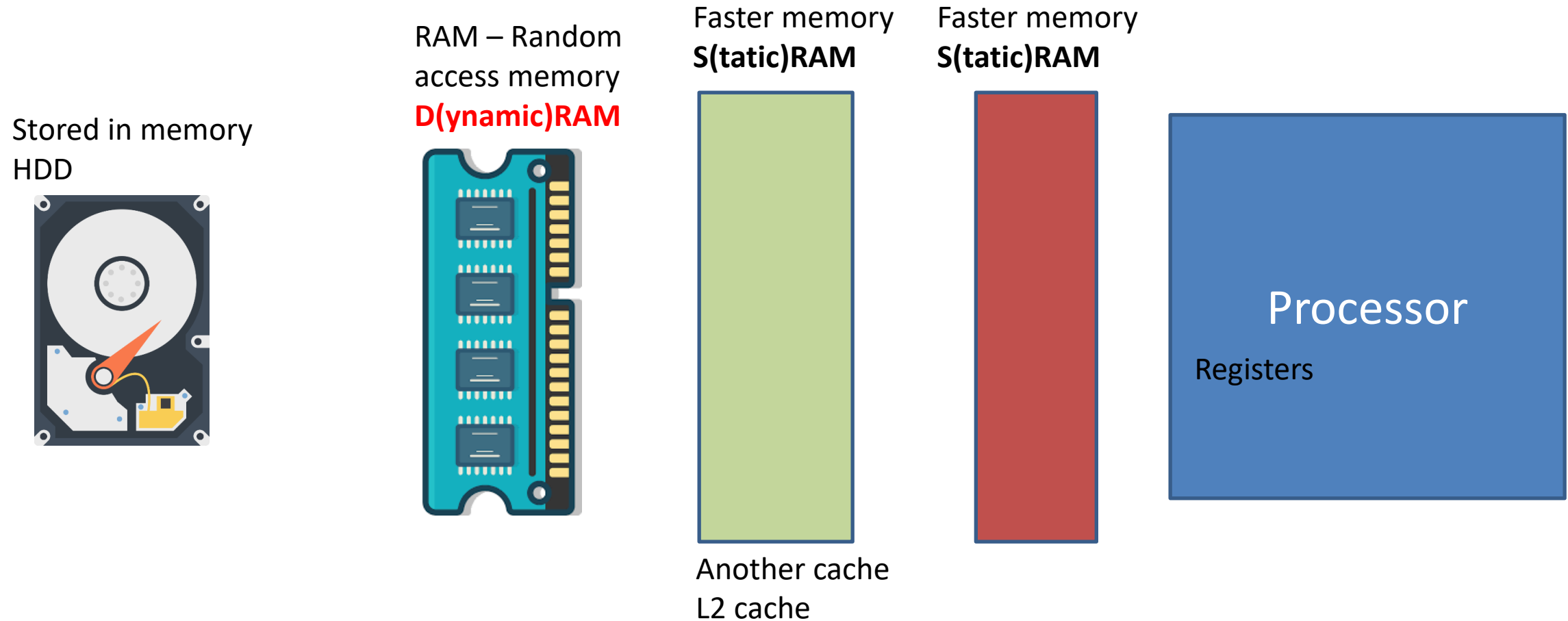
Processor

Registers

Hierarchy of memories



Hierarchy of memories



Multilevel Caches

- Primary cache attached to CPU
 - Small, but fast
- Level-2 cache services misses from primary cache
 - Larger, slower, but still faster than main memory
- Most systems include L-3 cache today
- Main memory services L-3 cache misses

Multilevel Caches

- If data not in L1- cache, but in L2-cache
 - Miss penalty = L2 cache access time
 - L2 access \ll Memory access time
- If data not in L2
 - Need to access memory
 - Large miss penalty

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - \Rightarrow clock cycle time = ? ns

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - \Rightarrow clock cycle time = 0.25ns
 - Miss rate/instruction = 2%
 - Main memory access time = 100ns
- With just primary cache
 - Miss penalty = ? cycles

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - \Rightarrow clock cycle time = 0.25ns
 - Miss rate/instruction = 2%
 - Main memory access time = 100ns
- With just primary cache
 - Miss penalty = $100\text{ns} / 0.25\text{ns} = 400$ cycles
 - Effective CPI

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - \Rightarrow clock cycle time = 0.25ns
 - Miss rate/instruction = 2%
 - Main memory access time = 100ns
- With just primary cache
 - Miss penalty = $100\text{ns} / 0.25\text{ns} = 400$ cycles
 - Effective CPI = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
- L1 miss with L-2 hit
 - Penalty = ? cycles

Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
- L1 miss with L-2 hit
 - Penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles
- L1 miss with L-2 miss
 - Extra penalty = 400 cycles
- CPI =

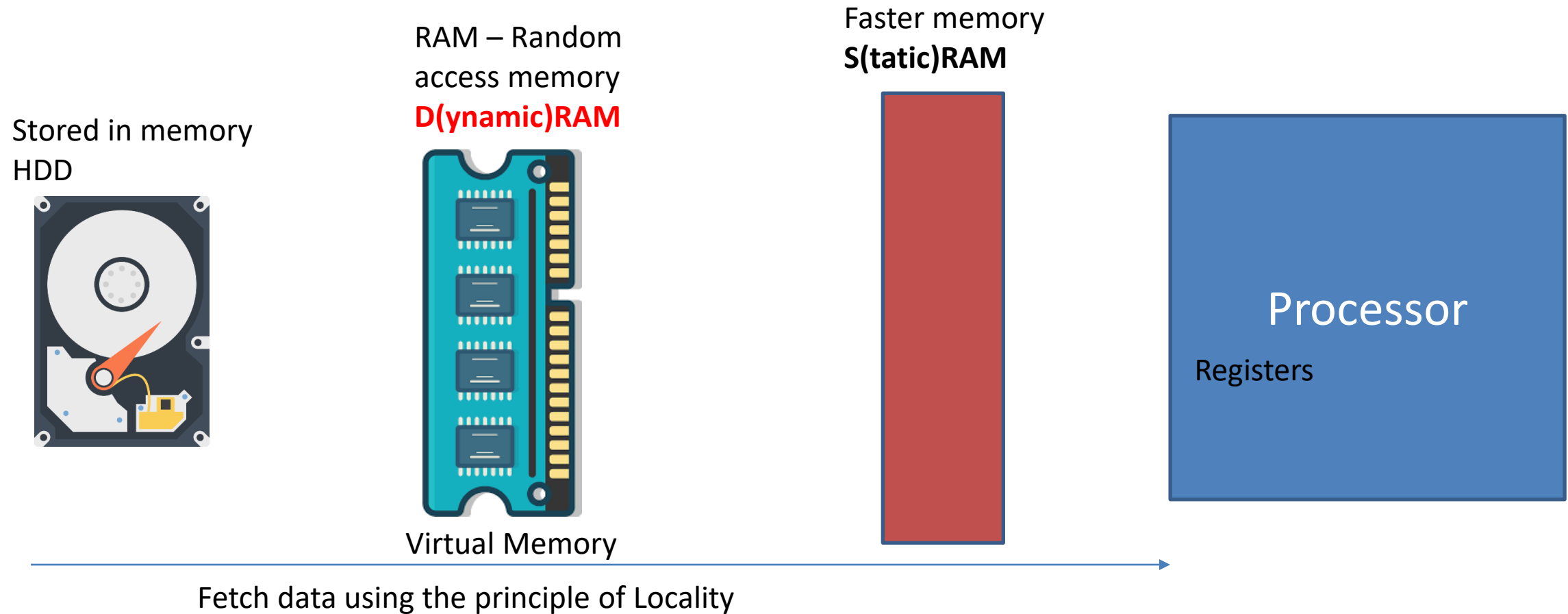
Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
- L1 miss with L-2 hit
 - Penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles
- L1 miss with L-2 miss
 - Extra penalty = 400 cycles
- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = $9/3.4 = 2.6$

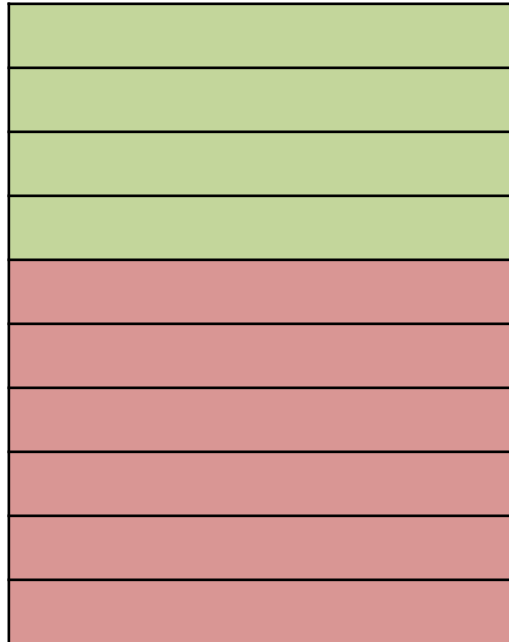
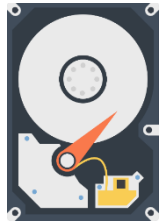
Multilevel Cache Considerations

- Primary cache
 - Focus on minimal hit time
- L-2 cache
 - Focus on low miss rate to avoid main memory access
 - Hit time has less overall impact

Hierarchy of memories



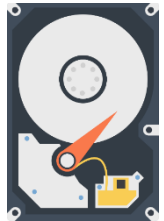
Stored in memory
HDD



Compile program 1

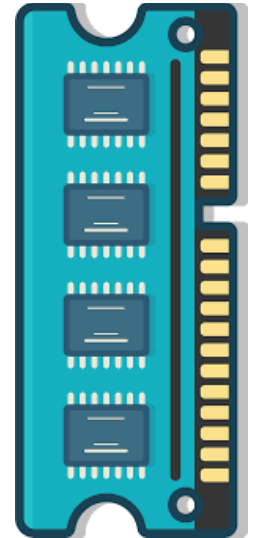
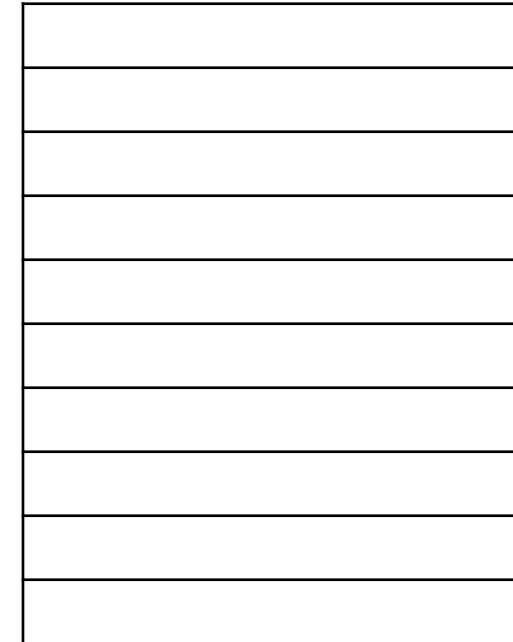
Compile program 2

Stored in memory
HDD



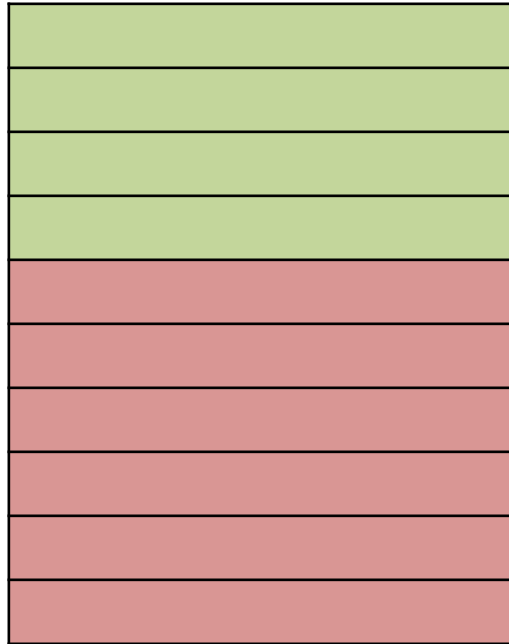
Compile program 1

Compile program 2



Load into Main memory, then
cache, registers, and then
execute

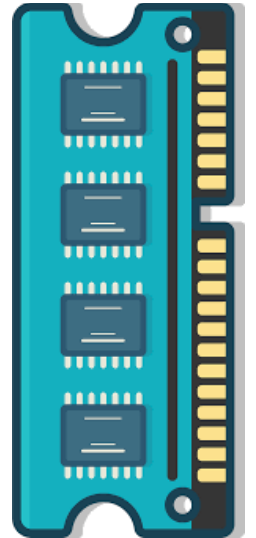
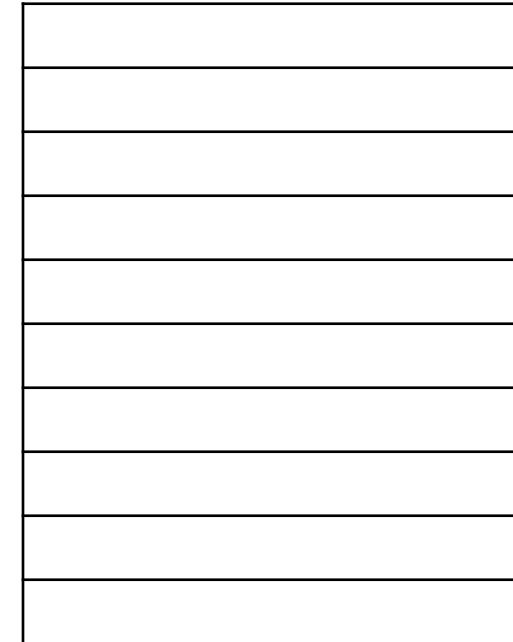
Stored in memory
HDD



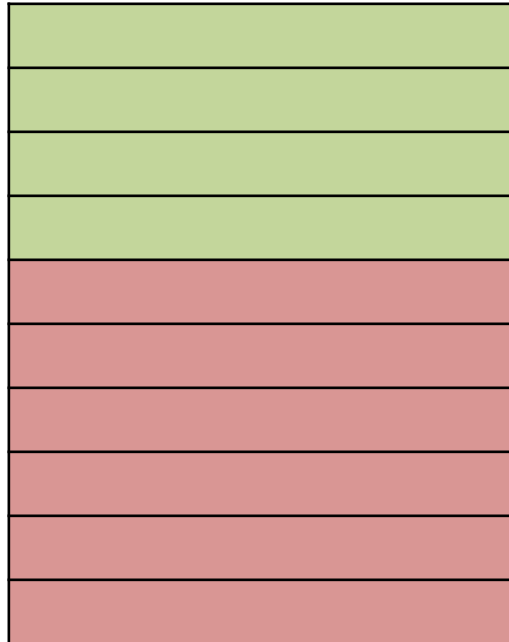
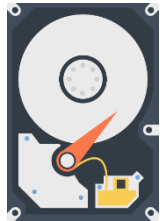
Compile program 1

Compile program 2

How to decide where to load the
data into?



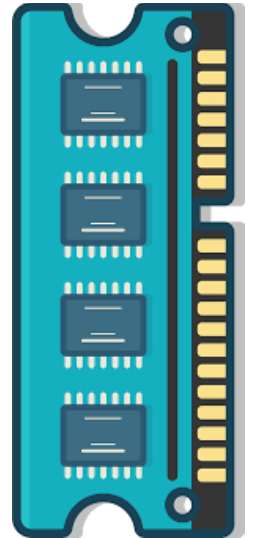
Stored in memory
HDD



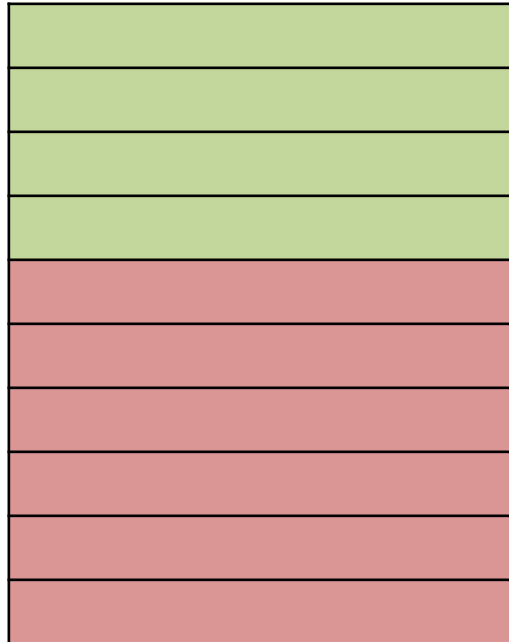
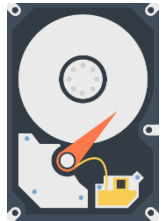
Compile program 1

Compile program 2

How to decide where to load the
data into?
Pre-define at compile time.



Stored in memory
HDD



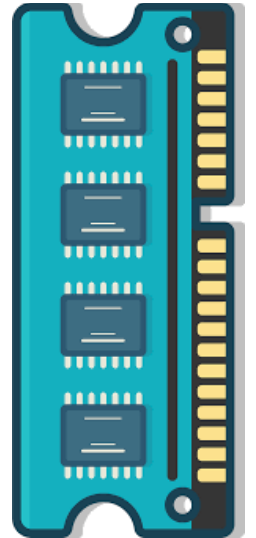
Compile program 1

Compile program 2

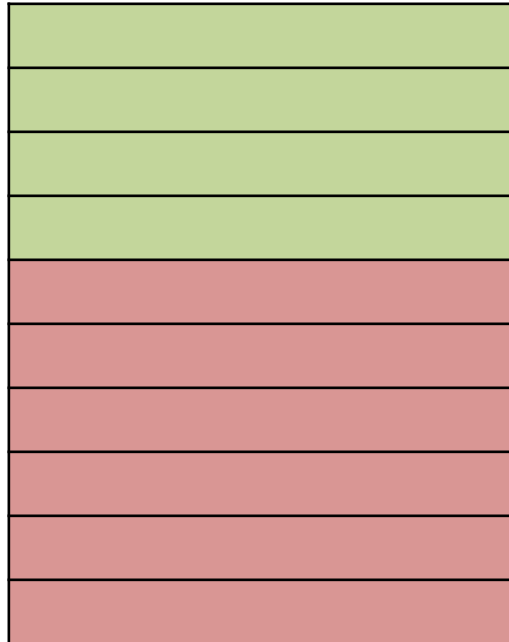
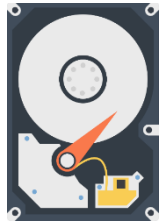
How to decide where to load the
data into?

Pre-define at compile time.

What if the mem locations are
used by another program



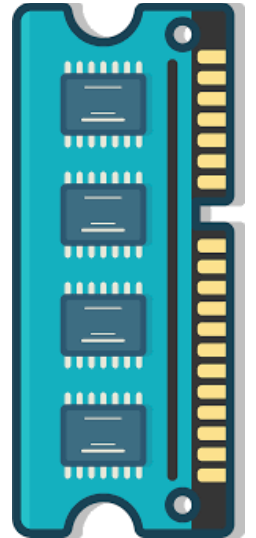
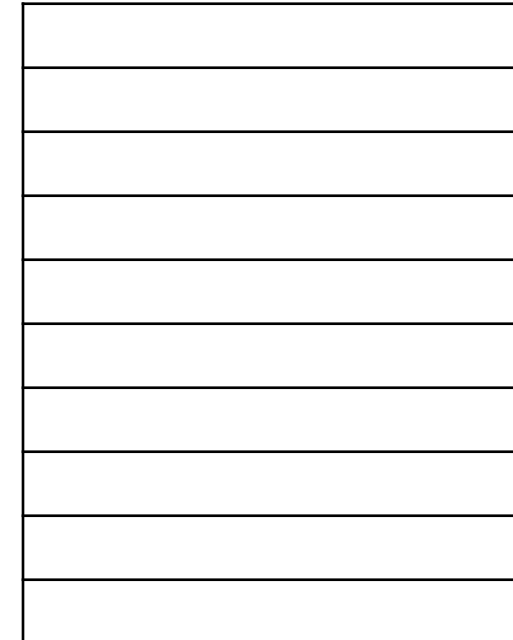
Stored in memory
HDD



Compile program 1

Compile program 2

Compile each program in its own
address space.
Independent of any physical
hardware address



Stored in memory
HDD



Compile program 1

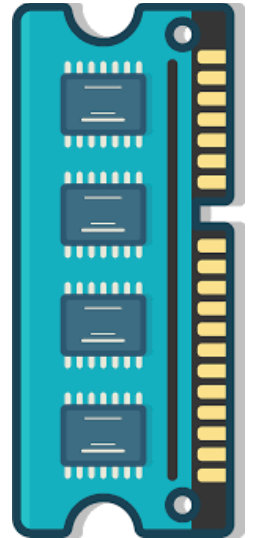
Compile program 2

Compile each program in its own
address space.

Independent of any physical
hardware address

Virtual Address/Memory

**Map/translate to main memory
locations**



Virtual Memory

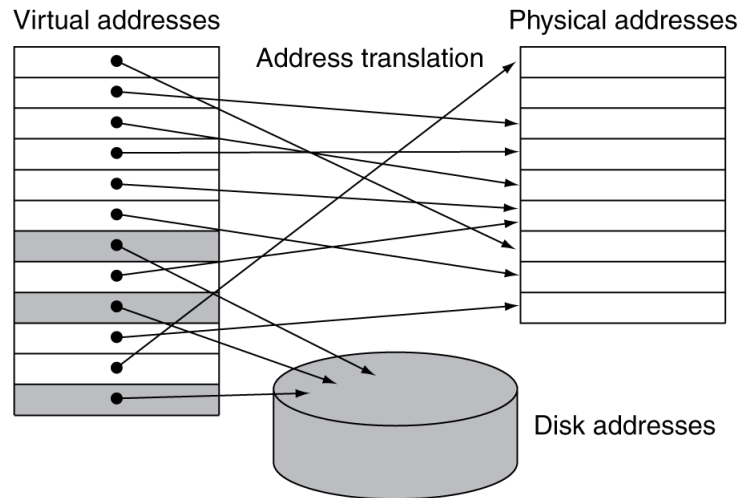
- Use main memory as a “cache” for secondary (disk) storage
 - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
 - Each gets a private virtual address space holding its frequently used code and data
 - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
 - VM “**block**” is called a **page**
 - VM translation “**miss**” is called a **page fault**

Advantages of Virtual Memory

- Extension:
 - Physical Memory can act as a cache for disk space
 - Process(es) can use more memory than actually available
 - Only a subset of program (“Working Set”) stored in physical memory
- Protection:
 - Different processes protected from each other
 - Kernel data protected from user programs
 - Different pages can be given special behavior (e.g. read only)
- Sharing:
 - Can map same physical page to multiple users (“Shared memory”)
- Relocation:
 - Map virtual addresses to physical main memory addresses

Address Translation

- Fixed-size pages (e.g., 4K)



Physical and Virtual memory are broken into pages. (ex. 4KiB)

Pages in virtual memory are mapped to pages in Main Memory (Via address translation)

Virtual address $\xrightarrow{\text{translation}}$ Physical Address

Virtual page not present in main memory but in secondary memory (storage/disk) → page fault

Address Translation

- Main memory size is 1 TiB (2^{40} Bytes)
- Each address stores 1 Byte
- How many address locations?

Address Translation

- Main memory size is 1 TiB (2^{40} Bytes)
- Each address stores 1 Byte
- How many address locations?
 - 2^{40}

Address Translation

- Main memory size is 1 TiB (2^{40} Bytes)
- Each address stores 1 Byte
- How many address locations?
 - 2^{40}
- How many bits per address

Address Translation

- Main memory size is 1 TiB (2^{40} Bytes)
- Each address stores 1 Byte
- How many address locations?
 - 2^{40}
- How many bits per address
 - 40

ARMv8 Address

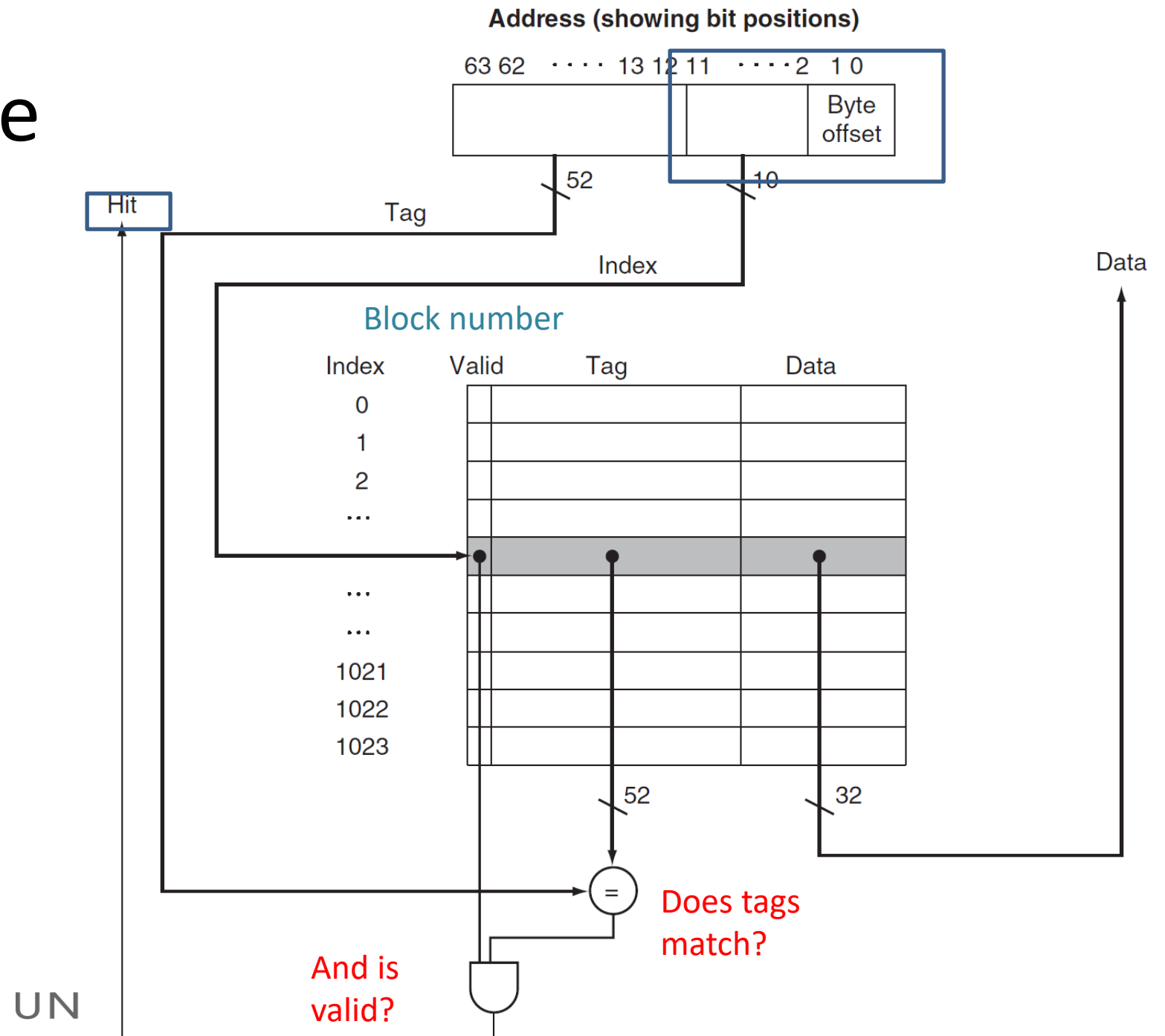
- Virtual addresses are 64 bits

ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address

Cache Hardware

Easy to handle cache hits.
CPU proceeds normally



ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address
- Virtual address broken into
 - Virtual page number
 - Page offset

ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address
- Virtual address broken into
 - Virtual page number
 - Upper portion of the address
 - Page offset
 - Lower portion, not changed
 - Size of offset, determines the page size

ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address
- Virtual address broken into
 - Virtual page number
 - Upper portion of the address
 - Page offset
 - Lower portion, not changed
 - Size of offset, determines the page size
 - Example 12-bits, what is the page size

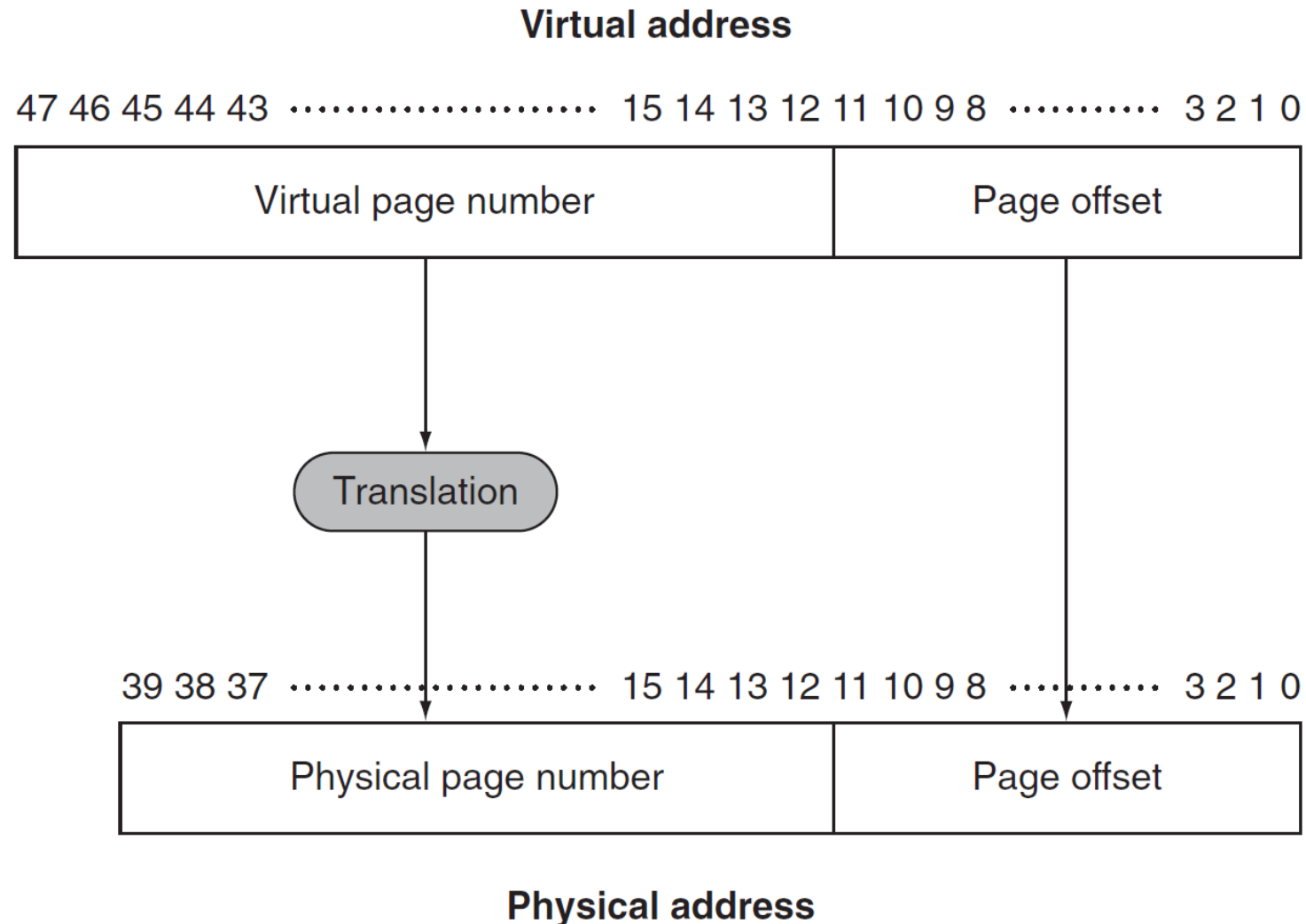
ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address
- Virtual address broken into
 - Virtual page number
 - Upper portion of the address
 - Page offset
 - Lower portion, not changed
 - Size of offset, determines the page size
 - Example 12-bits, what is the page size
 - 2^{12} bytes → 4KiB

ARMv8 Address

- Virtual addresses are 64 bits
 - 16 bits not used
 - Map 48 bit virtual address to 40 bit physical address
- Virtual address broken into
 - Virtual page number
 - Upper portion of the address
 - Page offset
 - Lower portion, not changed
 - Size of offset, determines the page size
 - Example 12-bits, what is the page size
 - 2^{12} bytes → 4KiB
 - Allows 2^{28} pages in main memory (total 2^{40})

Address Translation



Page Fault Penalty

- On page fault, the page must be fetched from disk
 - Takes millions of clock cycles
 - Handled by OS code
- Try to minimize page fault rate
 - Large page size: 4KiB to 64 KiB are common (as access time is high).
 - Fully associative placement: Reduces page fault rate.
 - Smart replacement algorithms
 - Write-back strategy: Write through too expensive, not feasible.

Placing a Page and Finding It

- Full associative scheme
 - Any virtual address can be mapped to any physical address
 - OS can choose what pages to replace

Placing a Page and Finding It

- Full associative scheme
 - Any virtual address can be mapped to any physical address
 - OS can choose what pages to replace
- Difficult to locate an entry
 - Use **Page Table**

Page Tables

- Table to index main memory
- Page table resides in main memory
- Virtual address is used as index
- Each program has its own page table

Page Tables

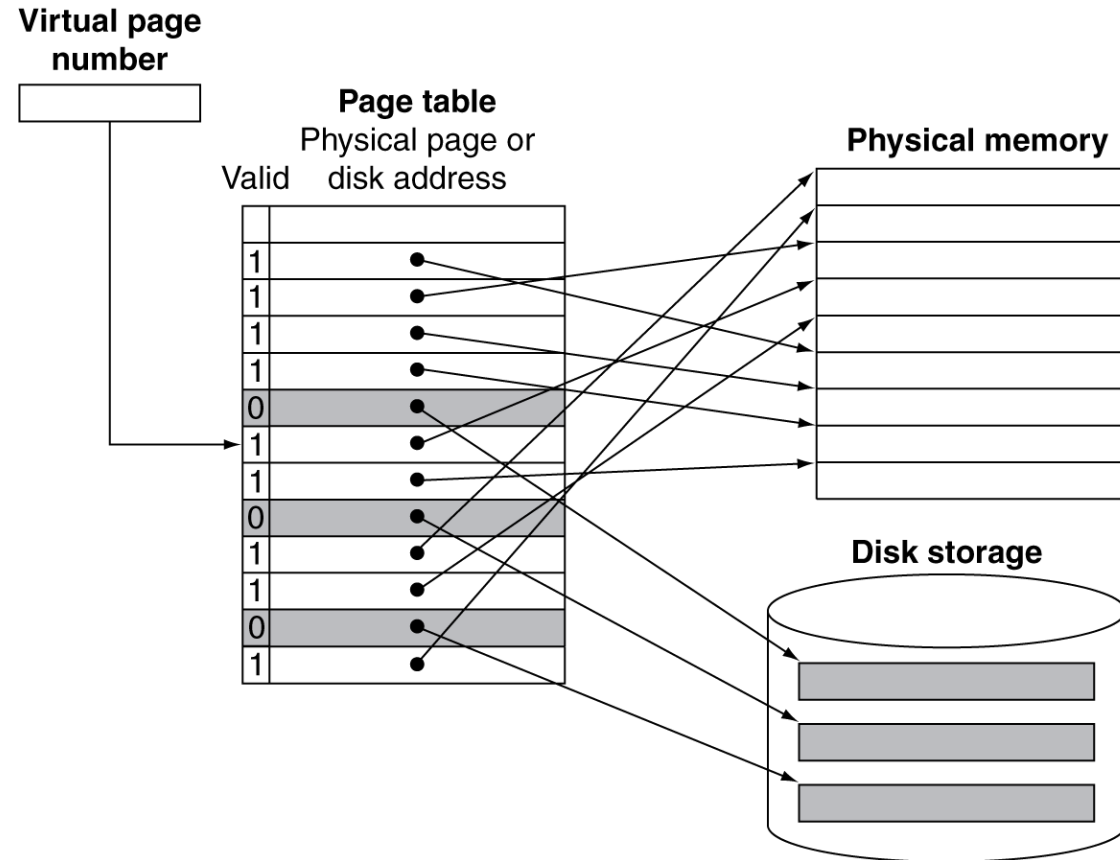
- Stores placement information
 - Page table register in CPU points to page table in physical memory
- If page is present in memory
 - PTE stores the physical page number
 - Plus other status bits (referenced, dirty, ...)
- If page is not present
 - PTE can refer to location in swap space on disk



Page Faults

- Virtual Address is not directly mapped to Secondary memory
- Track locations of pages in secondary memory
- Swap Space:
 - Space on secondary memory
 - Data structure to store where each page is located on the disk

Mapping Pages to Storage



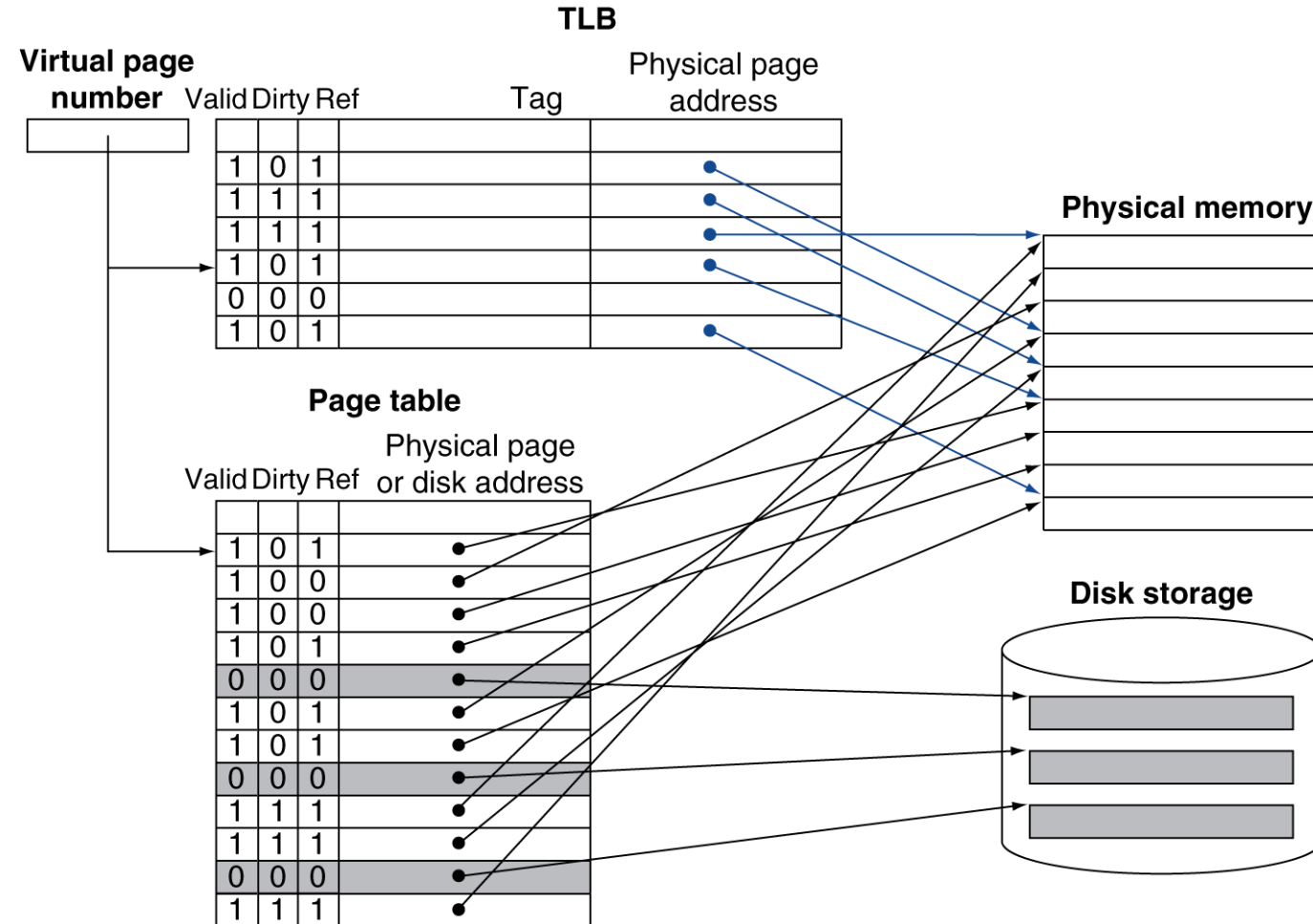
Page Tables can be Huge

- Problems with page tables
 - Page tables can be huge:
 - Consider a process using 1 GB of memory
Memory organized in 4k pages =>262,144 entries in page table
⇒translation step can be slow
⇒Break page tables into pages
⇒requires multiple memory pages to store the page table
 - Page table is stored in virtual memory
 - some memory pages storing the page table might be swapped onto disk
 - If using large number of processes, lots of memory consumed for storing page tables (1 per process)

Translation Look-Aside Buffer (TLB)

- Translation Look-Aside Buffer (TLB)
 - Cache for address translation containing the most frequently/recently used page frame addresses
 - TLB is hardware
 - Multi-level TLBs can be used similar to multi-level caches
 - TLBs typically much smaller than data/instruction caches
- If a virtual address is not found in TLB, use page table

Fast Translation Using a TLB



The Memory Hierarchy

The BIG Picture

- Common principles apply at all levels of the memory hierarchy
 - Based on notions of caching
- At each level in the hierarchy
 - Block placement
 - Finding a block
 - Replacement on a miss
 - Write policy

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?

Block Placement

- Determined by associativity
 - Direct mapped (1-way associative)
 - One choice for placement
 - n-way set associative
 - n choices within a set
 - Fully associative
 - Any location
- Higher associativity reduces miss rate
 - Increases complexity, cost, and access time

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
 - *Direct Mapped vs. Set-Associative vs. Fully Associative Caches*
- Q2: How is a block found if it is in the upper level?

Finding a Block

Associativity	Location method	Tag comparisons
Direct mapped	Index	1
n-way set associative	Set index, then search entries within the set	n
Fully associative	Search all entries	#entries
	Full lookup table	0

- Hardware caches
 - Reduce comparisons to reduce cost
- Virtual memory
 - Full table lookup makes full associativity feasible
 - Benefit in reduced miss rate

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
(Block placement)
 - *Direct Mapped vs. Set-Associative vs. Fully Associative Caches*
- Q2: How is a block found if it is in the upper level?
(Block identification)
 - *cache index, cache tag, cache offset*
- Q3: Which block should be replaced on a miss?

Replacement

- Choice of entry to replace on a miss
 - Least recently used (LRU)
 - Complex and costly hardware for high associativity
 - Random
 - Close to LRU, easier to implement
- Virtual memory
 - LRU approximation with hardware support

4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
(Block placement)
 - *Direct Mapped vs. Set-Associative vs. Fully Associative Caches*
- Q2: How is a block found if it is in the upper level?
(Block identification)
 - *cache index, cache tag, cache offset*
- Q3: Which block should be replaced on a miss?
(Block replacement)
 - *LRU vs. Random replacement*
- Q4: What happens on a write?

Write Policy

- Write-through
 - Update both upper and lower levels
 - Simplifies replacement, but may require write buffer
- Write-back
 - Update upper level only
 - Update lower level when block is replaced
 - Need to keep more state
- Virtual memory
 - Only write-back is feasible, given disk write latency