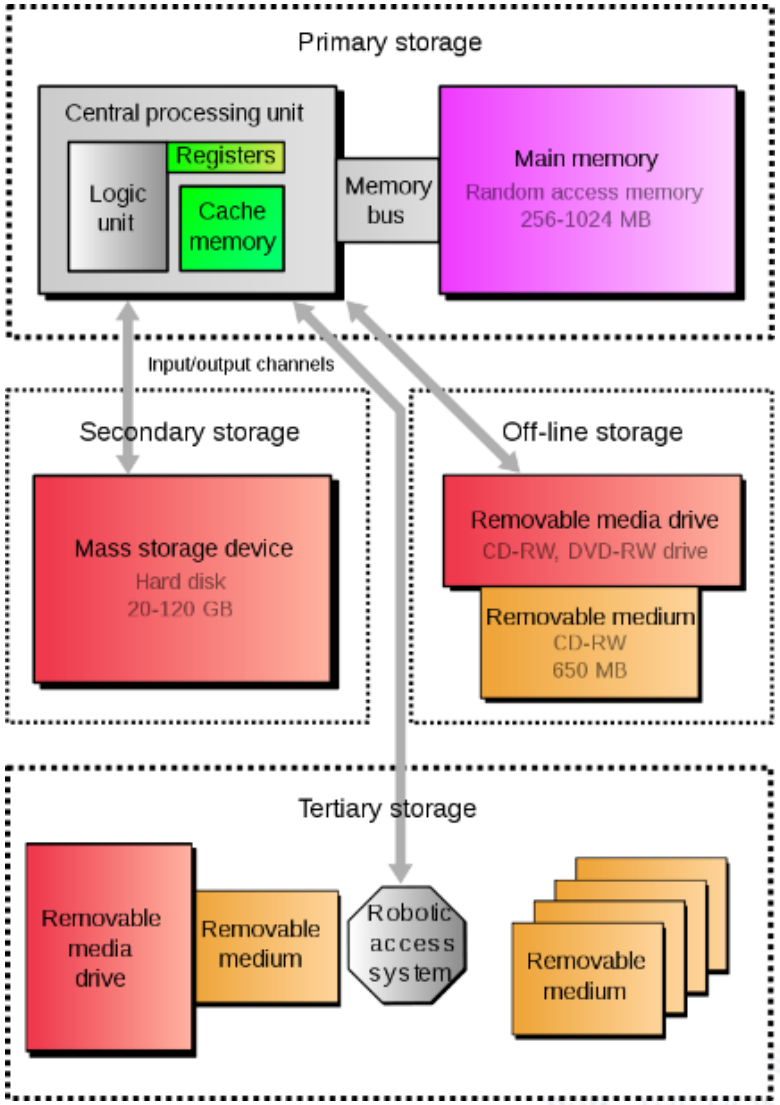
COMP2611 COMPUTER ORGANIZATION
MEMORY HIERARCHY

Major Goals

- How to build a **Large and Fast** memory system?
- Introduce the **memory hierarchy** and take advantage of the **principle of locality**
- Explain basic **cache** concepts
- Explain basic **virtual memory** concepts (optional)

Memory Hierarchy Overview



MEMORY TECHNOLOGY (OPTIONAL)

Memory Technology Revisit

- **Memory is the storage for instructions and data**

- **RAM (Random Access Memory) technology includes two types**
  - Static RAM (Static RAM) -- mostly used for **cache**
    - 0.5ns – 2.5ns, $2000 – $5000 per GB
  - Dynamic RAM (DRAM) – mostly used for **main memory**
    - 50ns – 70ns, $20 – $75 per GB
- **Hard Disk Drive (HDD) or Solid State Drive (SSD)**
  - Mostly used for **secondary storage**
  - 5ms – 20ms, $0.20 – $2 per GB

- **Ideal memory**
  - Fast: access time of SRAM
  - Large: capacity and cost/GB of disk

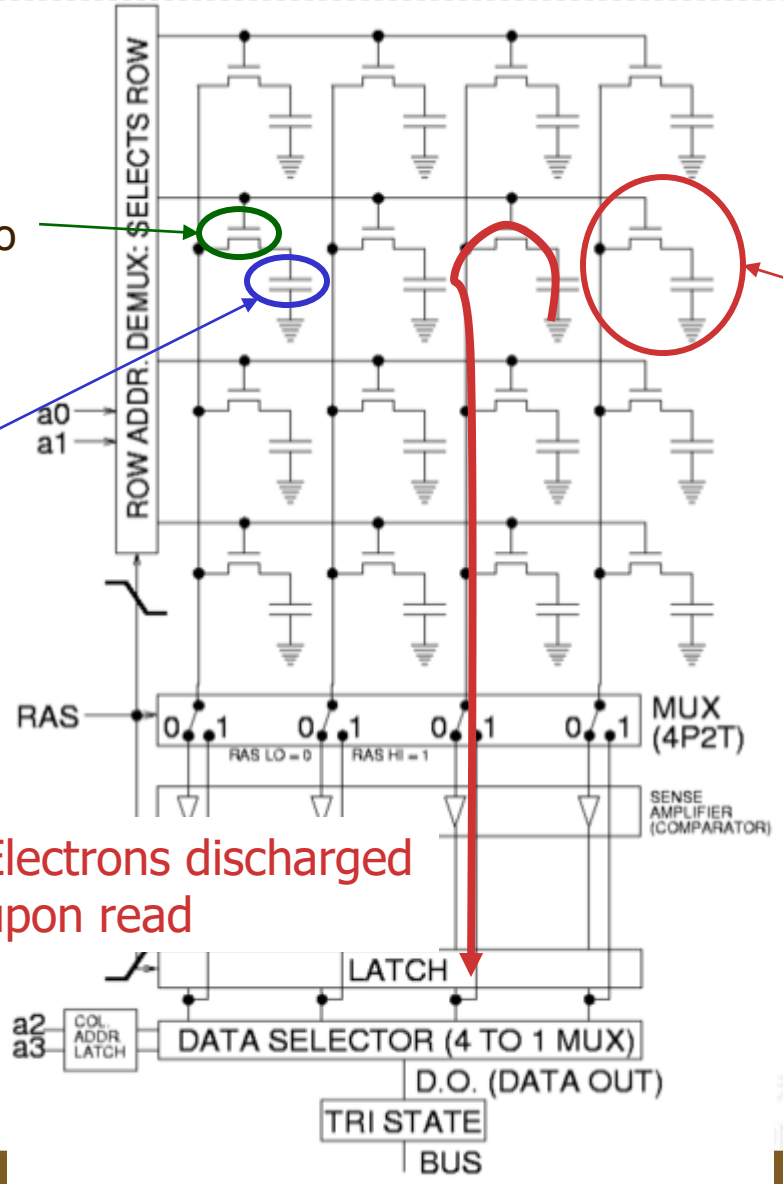
Example: 4x4 DRAM Array

Transistor:

To enable read/write to the capacitor

Capacitor:

To store the "value"



Electrons discharged upon read

A memory cell.

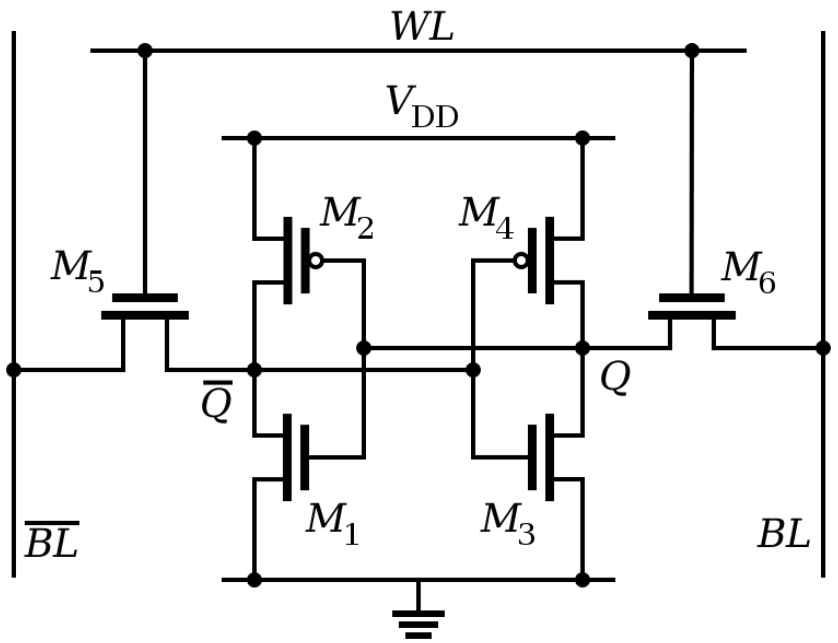
The first DRAM cell was invented in 1966 by Robert Dennard, a researcher at IBM's Thomas J. Watson Research Center

DRAM Technology

- **DRAM**
  - Consists of bits of data stored in a separate capacitors
  - Hold a logical “1” by having capacitor charged, and vice versa
- **Capacitors are**
  - Structurally simple, only one transistor and a one capacitor per bit
  - High density with low cost
- **Major problem of using capacitors:**
  - Real-world capacitors are not ideal, hence leak electrons
  - Information stored in capacitors eventually fades
  - Need to recharge the capacitors periodically to restore the “values”
  - Read operation discharges the electrons: read is destructive
- **Because of this refresh requirement, it is called dynamic memory**

SRAM Technology

- **SRAM is a type of semiconductor memory**
  - Each cell is constructed using transistors only (i.e. not capacitors)
  - Density not as high as DRAM, it is more expensive
  - Does not need to be periodically refreshed, the memory retains its contents as long as power remains applied
  - Read is not destructive in SRAM design; that's why called static



A six-transistor CMOS SRAM cell

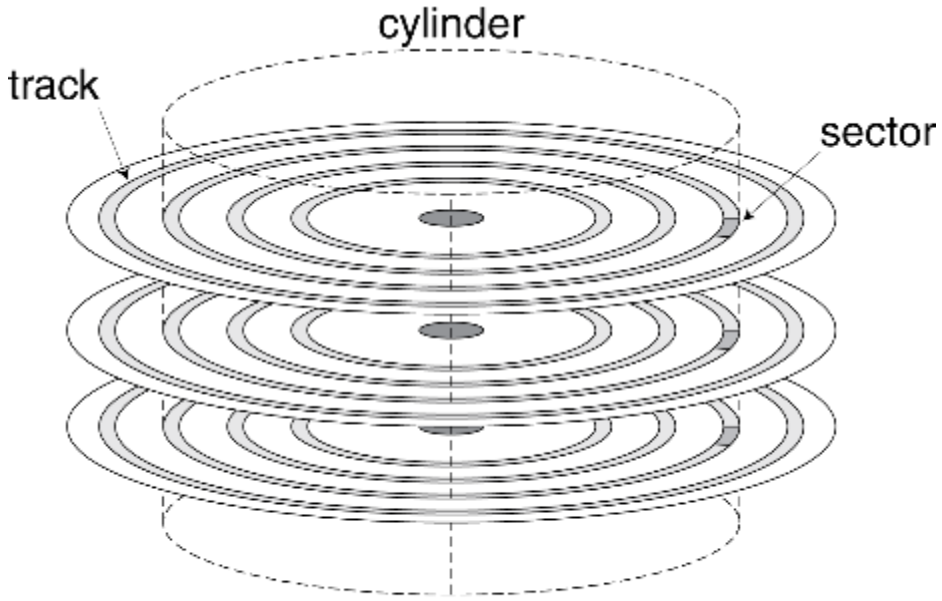
DRAM vs. SRAM

Comparison	SRAM	DRAM
Speed	Faster	Slower
Size	Small	Large
Cost	Expensive	Cheap
Used in	Cache memory	Main memory
Density	Less dense	Highly dense
Single block of memory	6 transistors	1 transistor
Charge leakage property	Not present	require power refresh
Power	Low	High

- **Before: discussions always assumed that memory access takes one cycle**
  From now on, this assumption is dropped
- **Implication: performance of processor looks bad with this reality**
- **In this chapter, we'll look into ways to minimize the impact of this**

Hard Disk Drive (HDD)

- Non-volatile, rotating magnetic storage



Disk Sectors and Accesses

- **Each sector records**
  - Sector ID
  - Data (512 bytes, 4096 bytes proposed)
  - Error correcting code (ECC)
    - Used to hide defects and recording errors
  - Synchronization fields and gaps
- **Access to a sector involves**
  - Queuing delay if other accesses are pending
  - Seek: move the heads
  - Rotational latency
  - Data transfer
  - Controller overhead

Disk Access Example

- **Given**
  - 512B sector, 15,000rpm, 4ms average seek time, 100MB/s transfer rate, 0.2ms controller overhead, idle disk

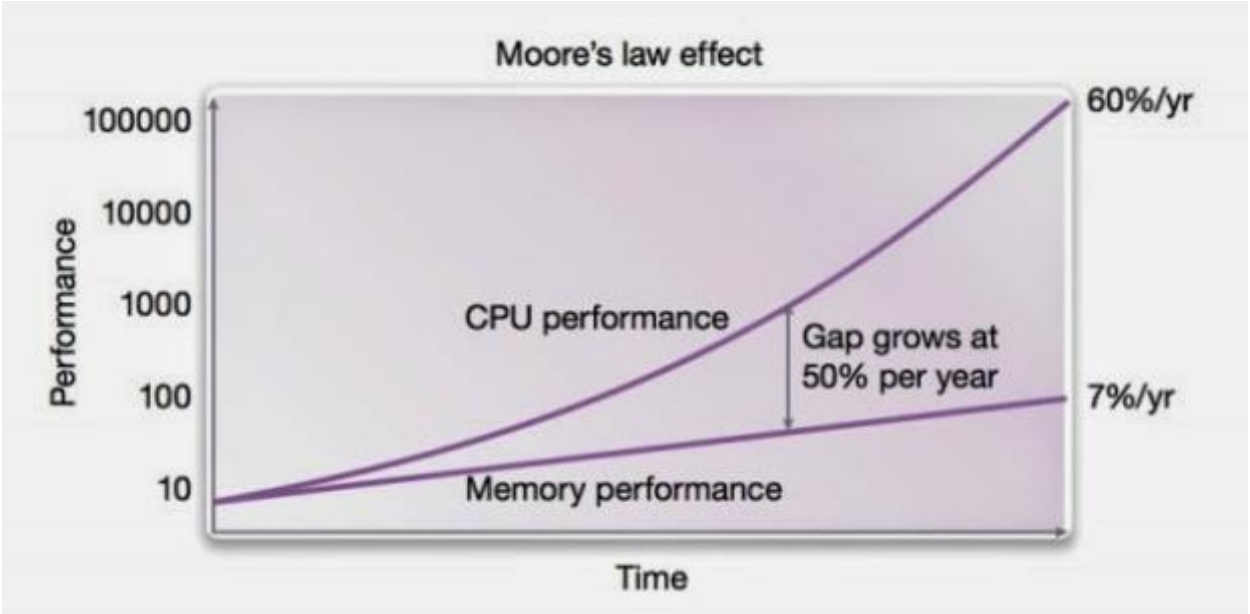
- **Average read time**
  - 4ms seek time
    + ½ / (15,000/60) = 2ms rotational latency
    + 512 / 100MB/s = 0.005ms transfer time
    + 0.2ms controller delay
    = 6.205ms
- **If actual average seek time is 1ms**
  - Average read time = 3.205ms

Solid-State Drive (SSD)

- **Non-volatile semiconductor storage**
  - 100x – 1000x faster than disk
  - Varying densities and capacities achieved in SSDs by stacking chips in a grid
  - Smaller, lower power, more robust
  - But more $/GB (between disk and DRAM)

Processor-Memory Bottleneck

- **DRAM improvement is not keeping up with processor improvement**
  - Processors are getting much faster than memories over time
  - The speed gap is widening!
- **Memory bottleneck**



- **Solution: Use a memory hierarchy exploiting the principle of locality**

PRINCIPLE OF LOCALITY

Principle of Locality

Programs access a small proportion of their address space at any time

- **Such execution pattern exhibits two types of localities**
- **Temporal locality**
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- **Spatial locality**
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

- **This property is the KEY to memory hierarchy!**

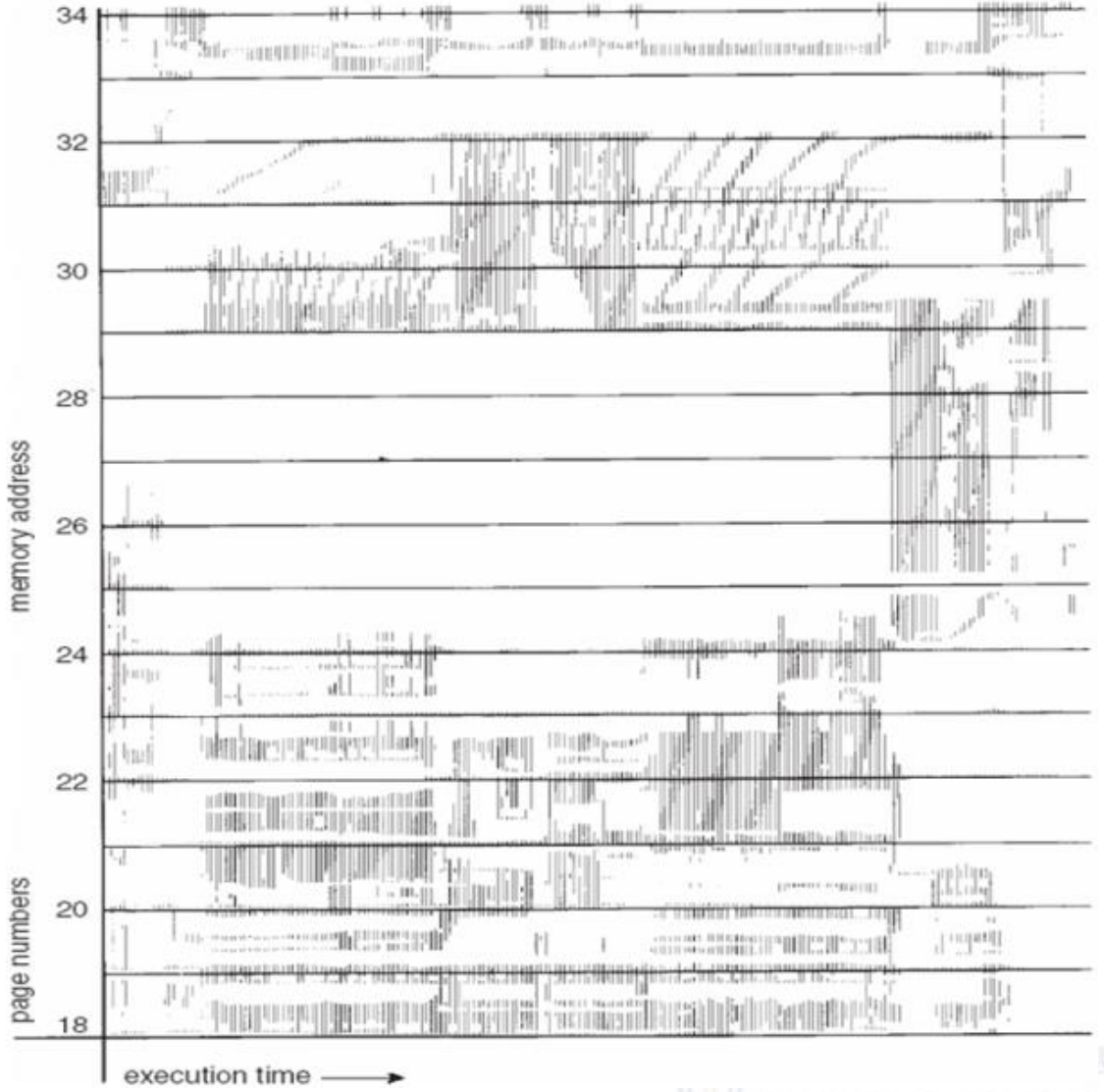
Example: Identifying Temporal and Spatial Locality

- Summing up 100 values stored in memory

```
        addi   $s0, $zero, 0      # $s0: accumulator
        addi   $s1, $zero, 100    # $s1: counter
L1:     lw     $t1, 0($s2)        # $s2: memory addr
        add    $s0, $s0, $t1
        addi   $s2, $s2, 4
        subi   $s1, $s1, 1
        bne    $s1, $zero, L1
```

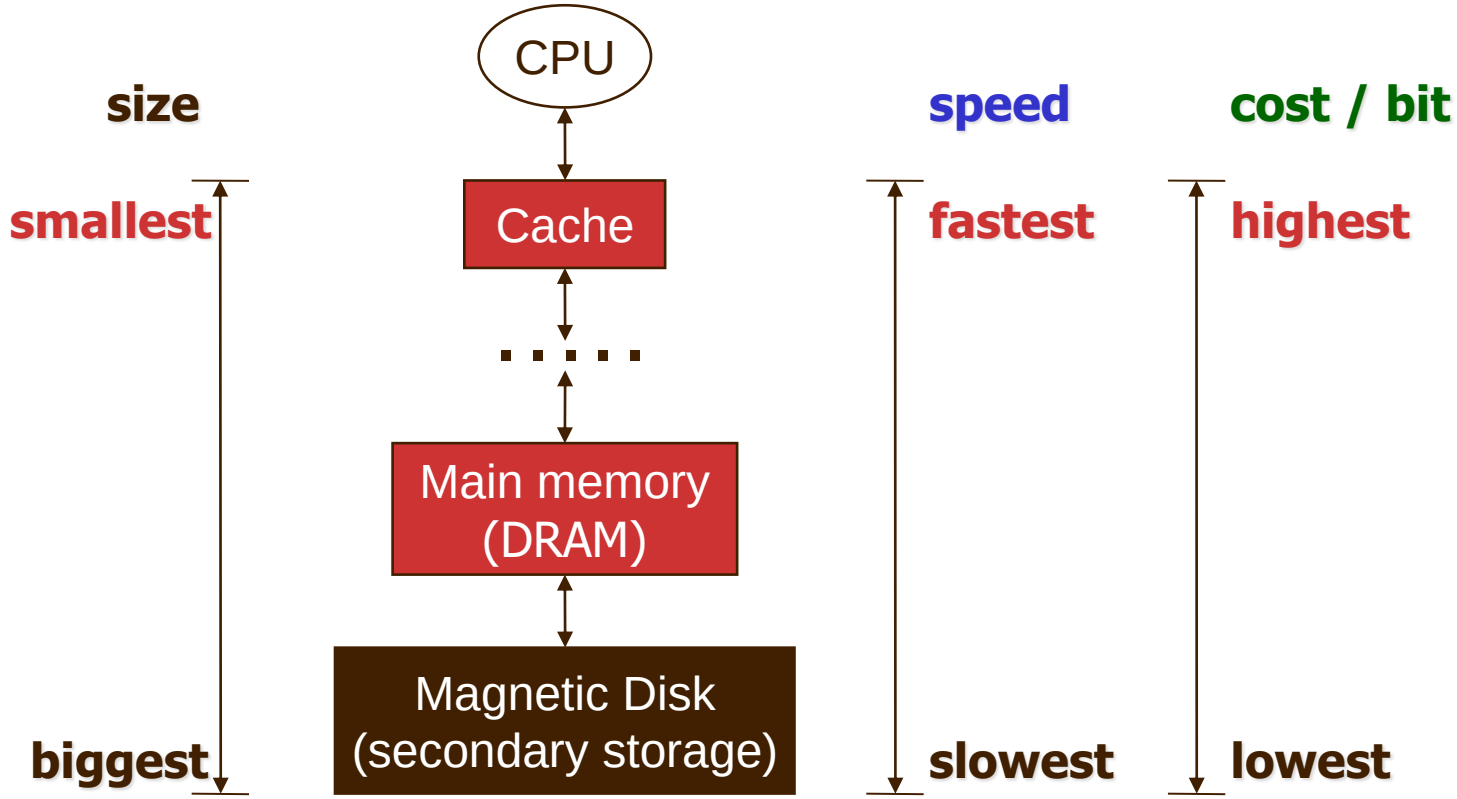
- Temporal and spatial locality can be observed in both instructions & data

Example: Locality During Program Execution



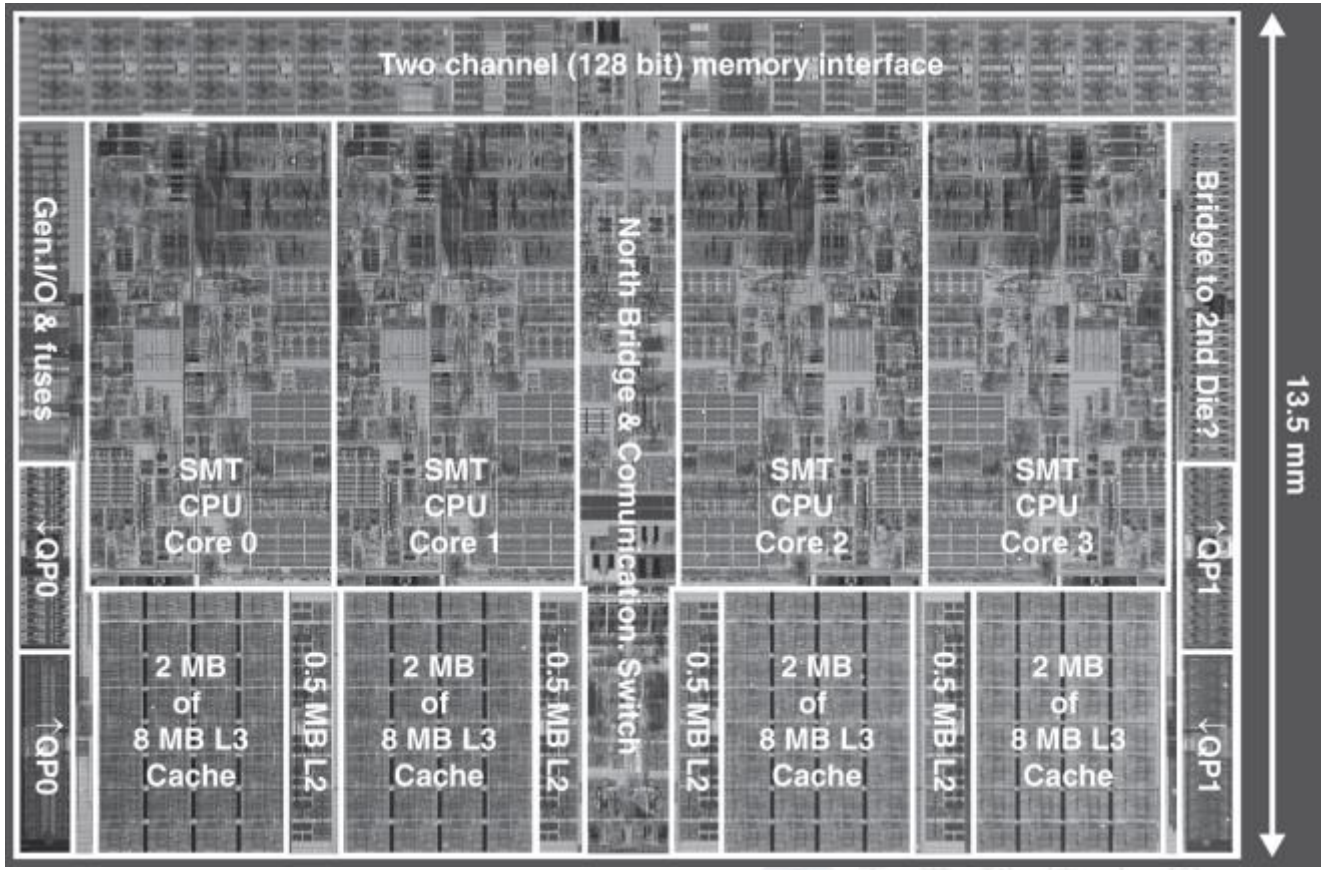
Memory Hierarchy

- A **memory hierarchy** consists of multiple levels of memory
- Users have the illusion of a memory
  - as large as the largest level
  - as fast as the fastest level
- A trade-off between performance and cost



Real Stuff

- Intel Nehalem 4-core processor
- Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache



How does memory hierarchy operate?

- **Store everything on hard disk**
- **When program starts to execute**
- **Initially,**
  - Instructions and data are loaded into memory (DRAM) from disk
- **Upon first access,**
  - A copy of the referenced instruction or data item is kept in cache
- **In subsequent accesses,**
  - First, look for the requested item in the cache
  - If the item is in the cache, return the item to the CPU
  - If NOT in the cache, look it up in the memory

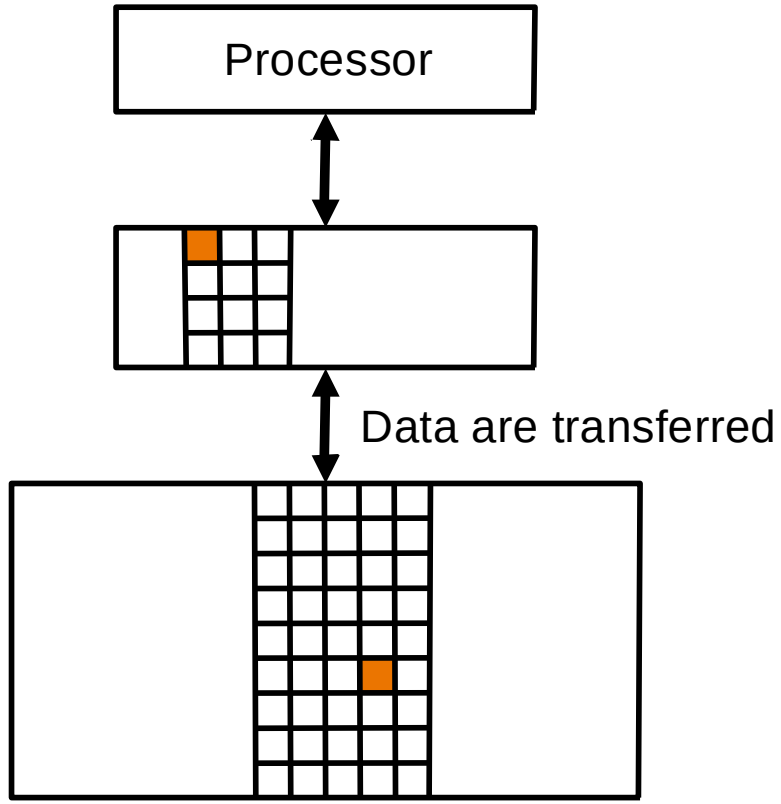
- **If not found in this level, look it up at next level until found**
- **Keep (or cache) a copy of the found item at this level after use**

Hit or Miss?

- Data are copied in **blocks** (aka **line, unit of copying, maybe multiple words**) between only two adjacent levels at a time

- If accessed data is present in upper level
- **Hit**: access satisfied by upper level
- Time to send data out: **hit time**

- If accessed data is absent
- **Miss**: block copied from lower level
- Data transfer occurs from the lower level to the upper level
- Time needed: **miss penalty**



Performance Measures

- **Hit rate (or hit ratio):**
  - Fraction of memory accesses found in the upper level
- **Miss rate: 1 - hit rate**

- **Cache hit time:**

  = time to determine miss or hit + time to access the cache
- **Cache miss penalty:**

  = time to bring a block from lower level to upper level

- **Hit time << Miss penalty**

Cache Performance

Average memory access latency = hit time + miss rate * miss penalty

For a configuration with a single level of cache, average memory latency

= hit time + miss ratio * miss penalty

= hit time + (1 - hit ratio) * miss penalty

For two levels of caches, average memory latency

= $\text{hit time}_1 + \text{miss ratio}_1 * \text{miss penalty}_1$

= $\text{hit time}_1 + \text{miss ratio}_1 * (\text{hit time}_2 + \text{miss ratio}_2 * \text{miss penalty}_2)$

= $\text{hit time}_1 +$

$\text{miss ratio}_1 * \text{hit time}_2 +$

$\text{miss ratio}_1 * \text{miss ratio}_2 * \text{miss penalty}_2$

The same idea can be extended to multiple levels of caches

Example: Performance with One Level of Cache

Example (assume everything else is ideal)

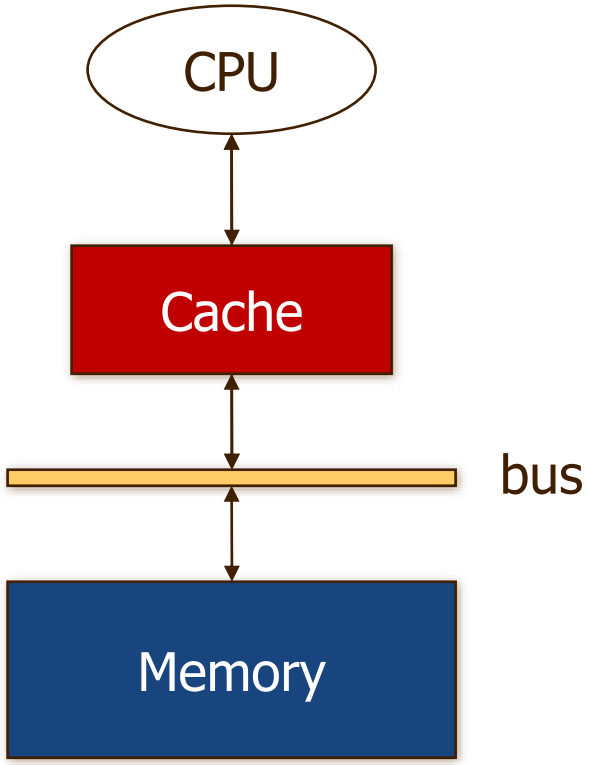
- 3-cycle cache access latency
- 100-cycle memory access latency
- Assuming 0-cycle for bus and cache lookup
- Data: 70% of the times in cache

Average memory latency

= 0.7 * 3 + 0.3 * (3 + 100)

= 3 + 0.3 * 100

= 33 cycles



Without cache, it is 100 cycles !!!

With caching, performance is better

We are closer to the ideal case!

Example: Performance with Two Levels of Caches

Example

- 3-cycle L1 latency
- 12-cycle L2 latency
- 100-cycle memory latency
- Assuming 0-cycle for bus and cache lookup
- Data: hit 70% in L1
- Data: hit 60% in L2

Average memory latency

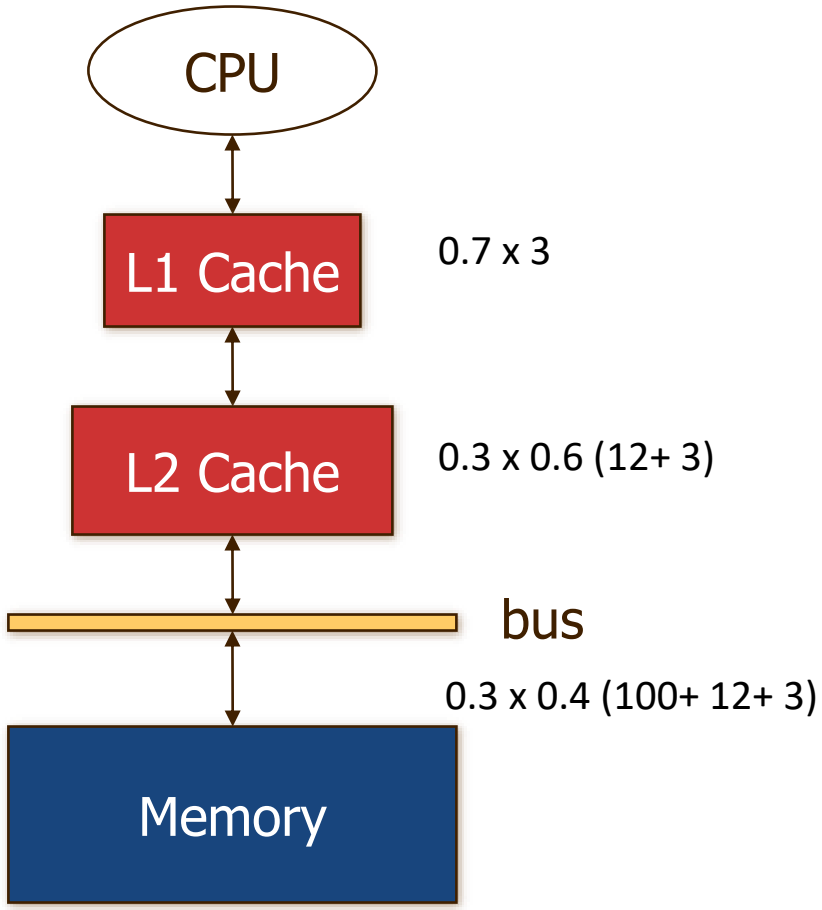
= 0.7 x 3 + 0.3 x 0.6 (12+3) + 0.3 x 0.4 (100+ 12+ 3)

Or alternative calculation

= 3 + 0.3 * 12 + 0.3 * 0.4 * 100

= 3 + 3.6 + 12.0

= 18.6 cycles (better than one level)



CACHE

Library Analogy

Library (main memory)

Book shelf (cache)

Cache Operations

Block placement

- Where is a block placed in the cache?

Block identification

- How can a block be found if it is in the cache?

Block replacement

- Upon miss, how the victim block in the cache is selected for replacement?

Write strategy

- When a write occurs, is the information written only to the cache?

Cache Organization

Cache is organized as an array of cache blocks

Each memory location is mapped to ONE location in the cache

Cache block

- A minimum unit of information that can be present in cache or not

Cache block sizes

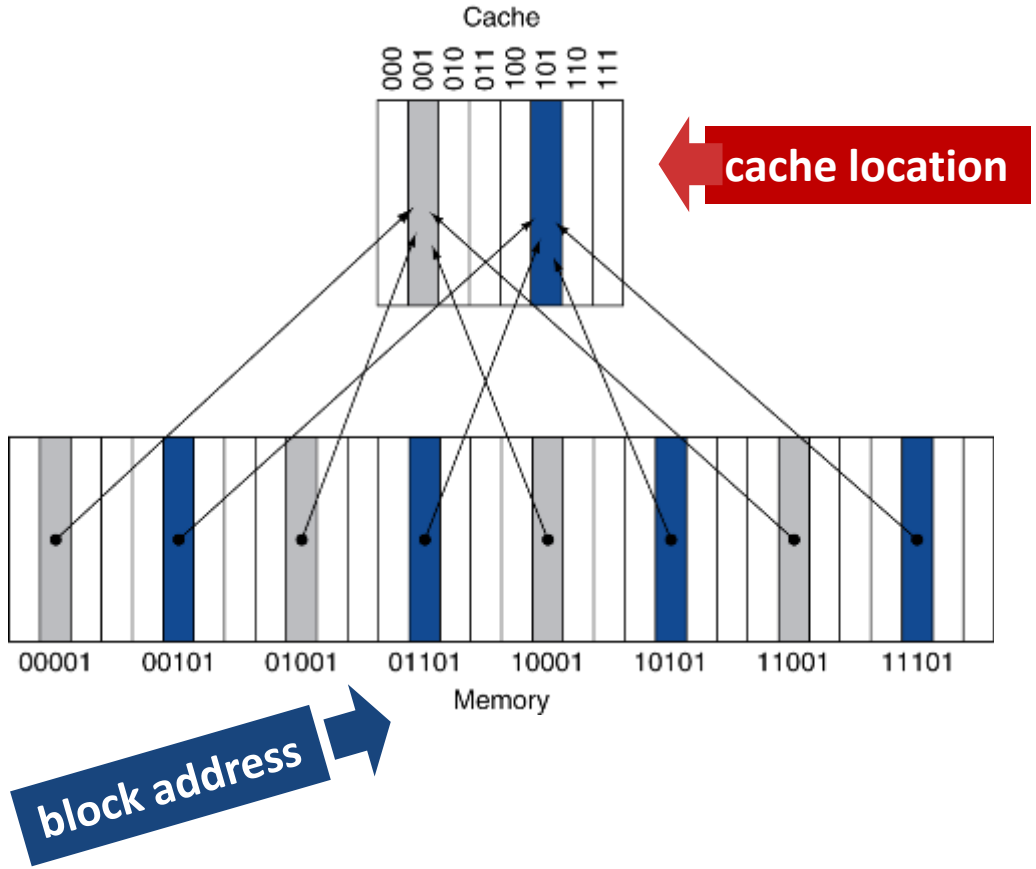
- Commonly used today: 32 bytes and 64 bytes

Three basic organizations

- Direct-mapped (one memory block to one possible cache block)
- Set-associative (one memory block to one set of possible cache blocks)
- Fully-associative (one memory block to all possible cache blocks)

Direct Mapped Cache

- **When transfer a block from main memory to cache**
  - Its location in cache is determined by memory address
- **Direct mapped: only one candidate location in cache**
  - (Block address) modulo (#Blocks in cache)
- **Many-to-one mapping**

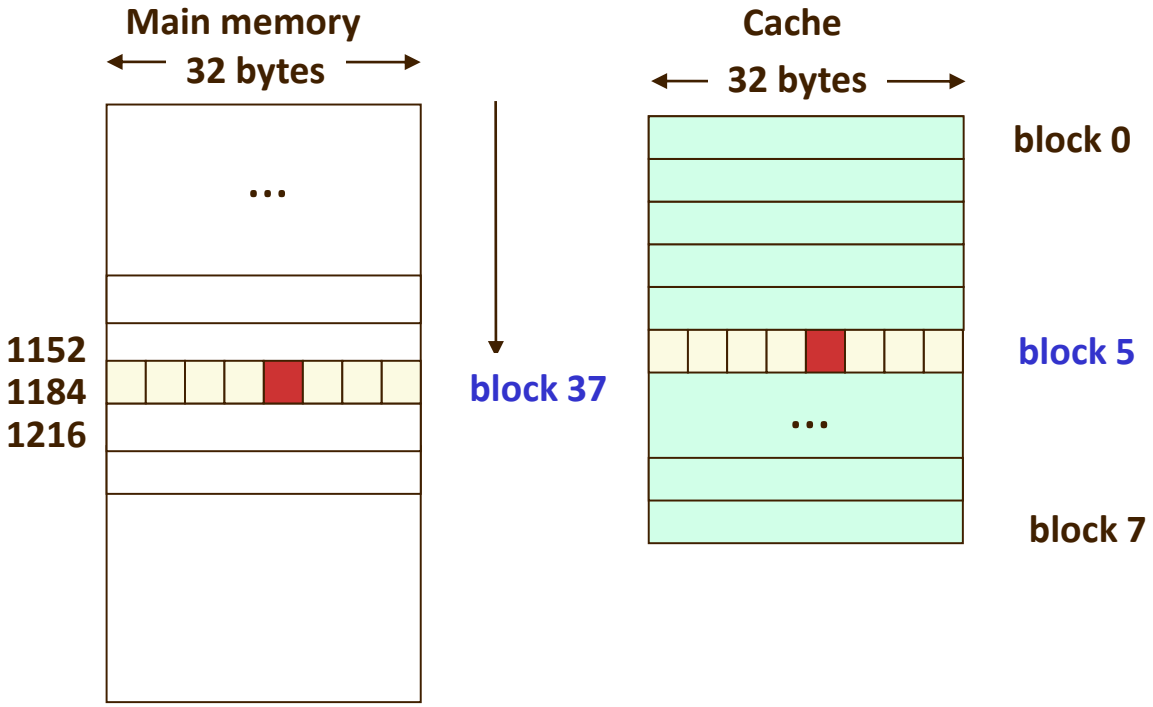


Block Placement Example

- Block size = 8 words (i.e. 32 bytes)
- Direct mapped cache has 8 blocks

Example: `lw $t0, 1200($zero)`

- Processor asks for a word from byte address 1200
- That word resides in block 37 in memory, e.g. floor(1200/32) = 37
- The whole block will be loaded to cache set 5, e.g. 37 mod 8 = 5
- Then that word will be sent to processor



Block Placement Shortcut

- It belongs to memory block 37: 0000 0000 0000 0000 0000 0100 101 (block address)
- Block 37 will be loaded to cache block 5: 101

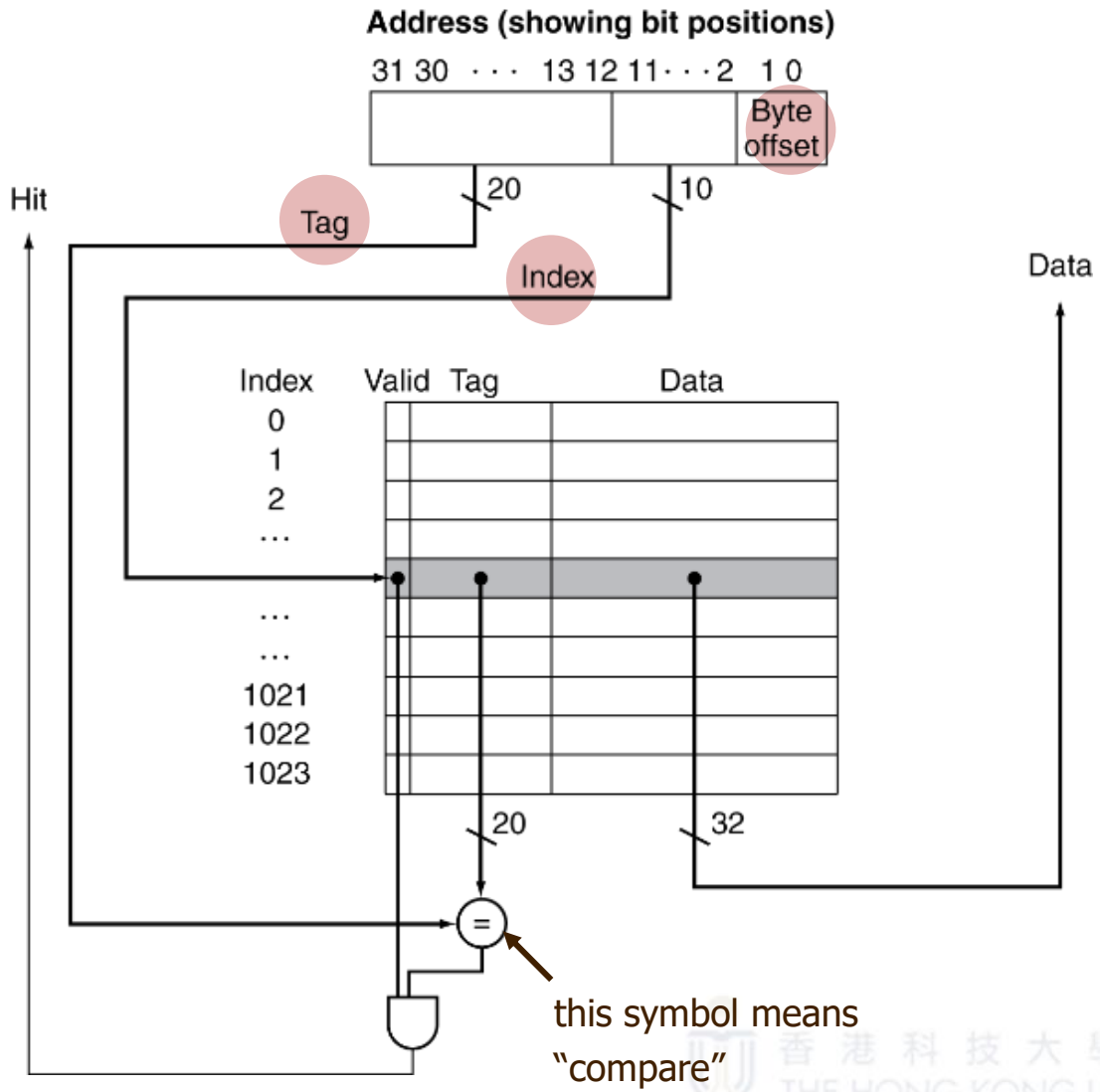
- Shortcut:
- If the number of cache sets (N) is a power of 2; $N = 2^m$
- Cache_location = the low-order m bits of block address

- Which cache set?
- 0000 0000 0000 0000 0000 0100 1011 0011
- 0011 0001 0010 0000 0000 0100 1011 0000

Block Identification: Tags and Valid Bits

- **How do we know which particular block is stored in a cache location?**
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the tag
- **What if there is no data in a location?**
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

Address Sub-divison



Block Identification Example

- block size = 8 words (i.e. 32 bytes)
- Direct-mapped cache with 8 blocks

Memory (byte) address generated by processor	Block address	Hit or miss in cache	Assigned cache block (where found or placed)
$0010\ 1100\ 0010_2$	$0010\ 110_2$	**Miss**	$(0010110_2 \bmod 8) = 110_2$
$0011\ 0100\ 0000_2$	$0011\ 010_2$	**Miss**	$(0011010_2 \bmod 8) = 010_2$
$0010\ 1100\ 0100_2$	$0010\ 110_2$	**Hit**	$(0010110_2 \bmod 8) = 110_2$
$0010\ 1100\ 0010_2$	$0010\ 110_2$	**Hit**	$(0010110_2 \bmod 8) = 110_2$
$0010\ 0000\ 1000_2$	$0010\ 000_2$	**Miss**	$(0010000_2 \bmod 8) = 000_2$
$0000\ 0110\ 0000_2$	$0000\ 011_2$	**Miss**	$(0000011_2 \bmod 8) = 011_2$
$0010\ 0001\ 0000_2$	$0010\ 000_2$	**Hit**	$(0010000_2 \bmod 8) = 000_2$
$0010\ 0100\ 0001_2$	$0010\ 010_2$	**Miss**	$(0010010_2 \bmod 8) = 010_2$

Cache BLK#	Tag
000	0010
001	
010	0010
011	0000
100	
101	
110	0010
111	

- Miss rate = (# of misses) / (# total memory accesses) = 5 / 8

Bits in Cache

- A direct-mapped cache with 16KB of data and 8-word (2^5 bytes) blocks
- Assume 32-bit memory address
- What is the actual cache size?

16KB = **4K** words = $\mathbf{2^{12}}$ words

Block size of **8** words -> $\mathbf{2^9}$ blocks

Each block has **8 x 32 = 256** bits of data

A tag has **32 - 9 - 5 = 18** bits

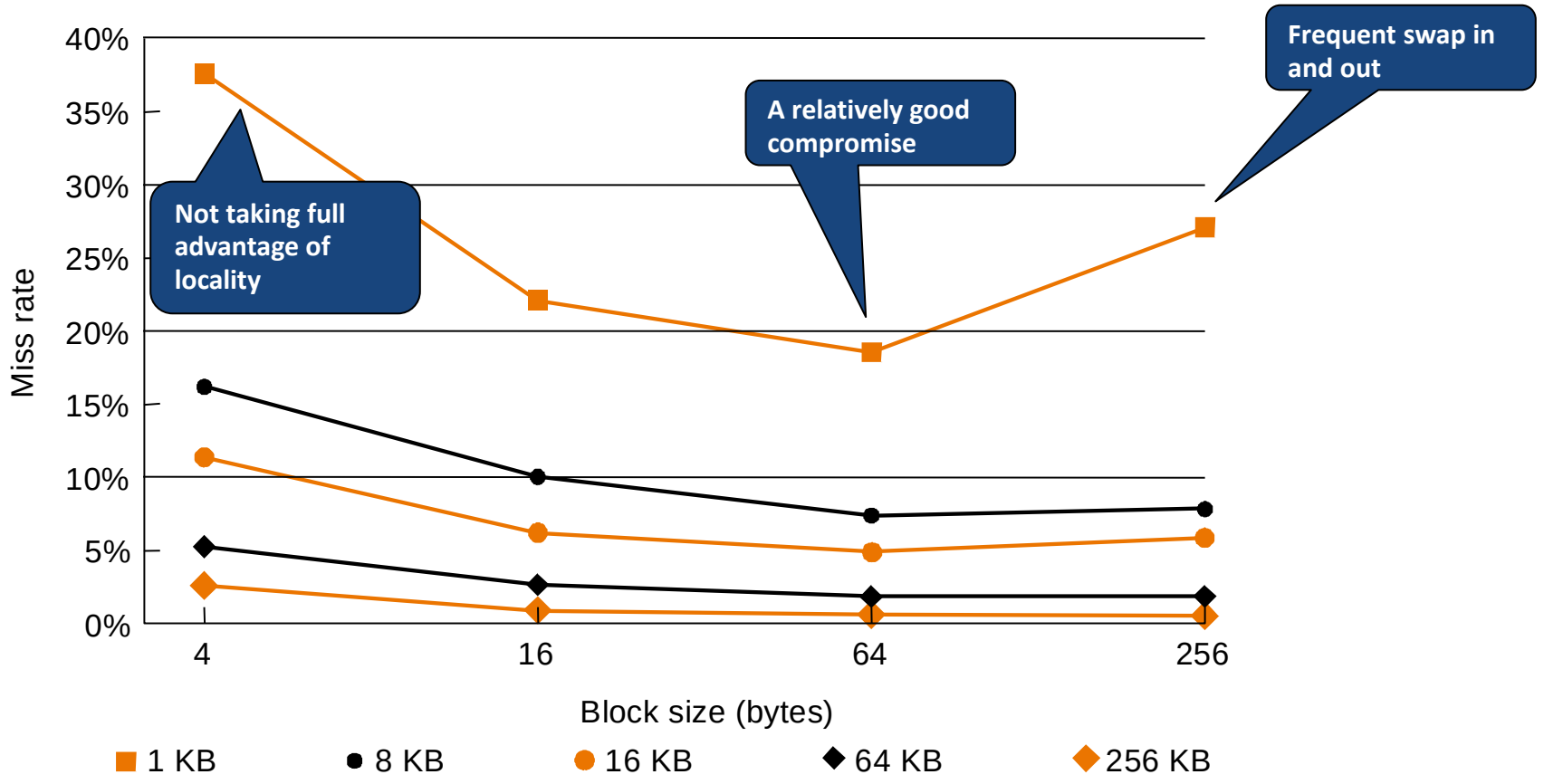
And, **1** valid bit

Total bits per entry = **(256 + 18 + 1)**

Total cache size

$\mathbf{2^9 \times (256 + 18 + 1) = 2^9 \times 275 = 140800 = 137.5}$ **Kbits**

Block Size vs. Miss Rate



Cache Misses

- **On cache hit, CPU proceeds normally**
- **On cache miss**
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Disadvantage of Direct Mapped Cache

- **Blocks mapped to same cache block can't be present simultaneously**
- **Example: Given a cache with 8 blocks**
  - Processor sends out the following memory block address to access: 100011, 001011, 100011
  - Block 100011 & 001011 are mapped to the same cache set 011
  - Arrival of block 001011 will kick (i.e. replace) 100011 out
    - i.e. both blocks can't be present simultaneously
  - Subsequent access to 100011 will not hit but miss
- **Less performance improvement with such cache conflict**
- **Can we alleviate this conflict problem to increase the cache hit rate?**

Associate Cache

Fully associative (FA)

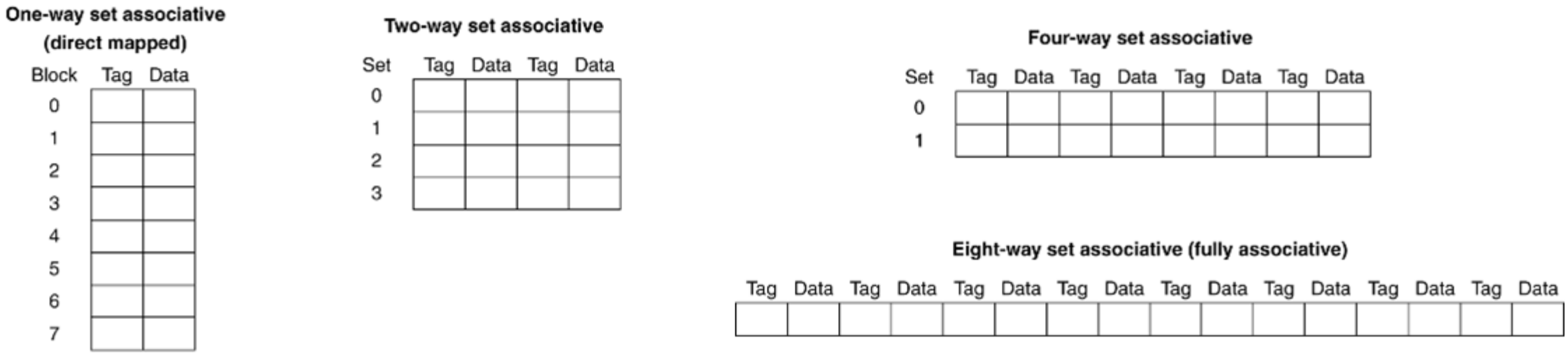
- Each block can be placed **anywhere** in the cache
- **Advantage:**
  - No cache conflict ⇒ better cache hit rate than **direct mapped**
  - But, still see misses due to size (capacity miss)
- **Disadvantage:**
  - Costly (hardware and time) to search for a block in the cache

N-Way Set associative (SA)

- Each block can be placed in a **certain number (N)** of cache locations (i.e. **set**)
- A good **compromise** between direct mapped & fully associative
  - In terms of cost and hit rate

Spectrum of Cache Associativity

- **Example: associativity in a cache with 8 blocks**



How much associativity?

- **Increase in degree of associativity**
  - Advantage: Decrease in miss rate
  - Disadvantage: Increase in hit time (due to longer search time)
  - But with diminishing returns
- **Simulation of a system with 64KB Data cache, 16-word blocks, SPEC2000**
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%

Associativity Example

- **Compare 4-block caches**
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- **Direct mapped**

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

Associativity Example (cont.)

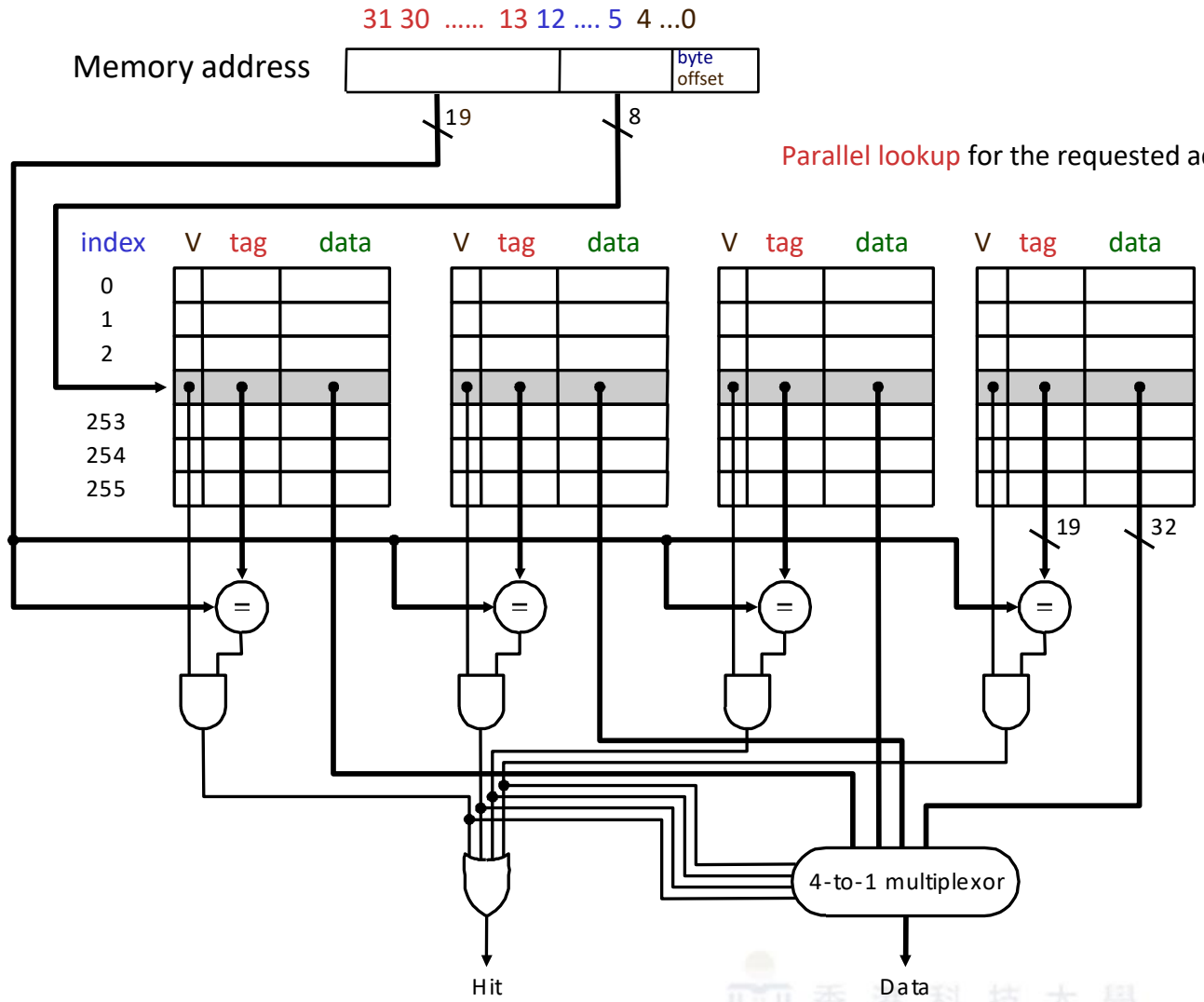
- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	**Mem[6]**	Mem[8]		
8	0	hit	Mem[6]	**Mem[8]**		

- Fully associative

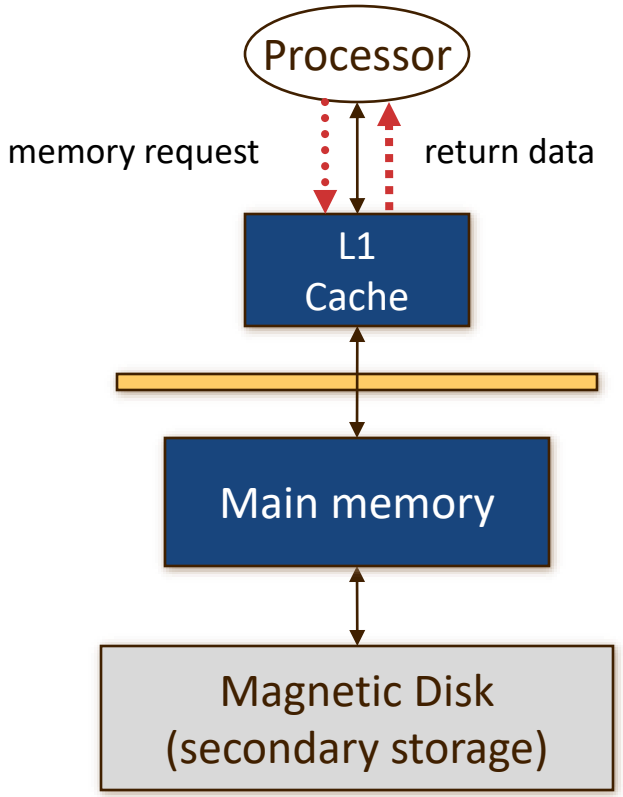
Block address		Hit/miss	Cache content after access			
0		miss	**Mem[0]**			
8		miss	Mem[0]	**Mem[8]**		
0		hit	**Mem[0]**	Mem[8]		
6		miss	Mem[0]	Mem[8]	**Mem[6]**	
8		hit	Mem[0]	**Mem[8]**	Mem[6]	

Block Identification in N-way Set Associative Cache

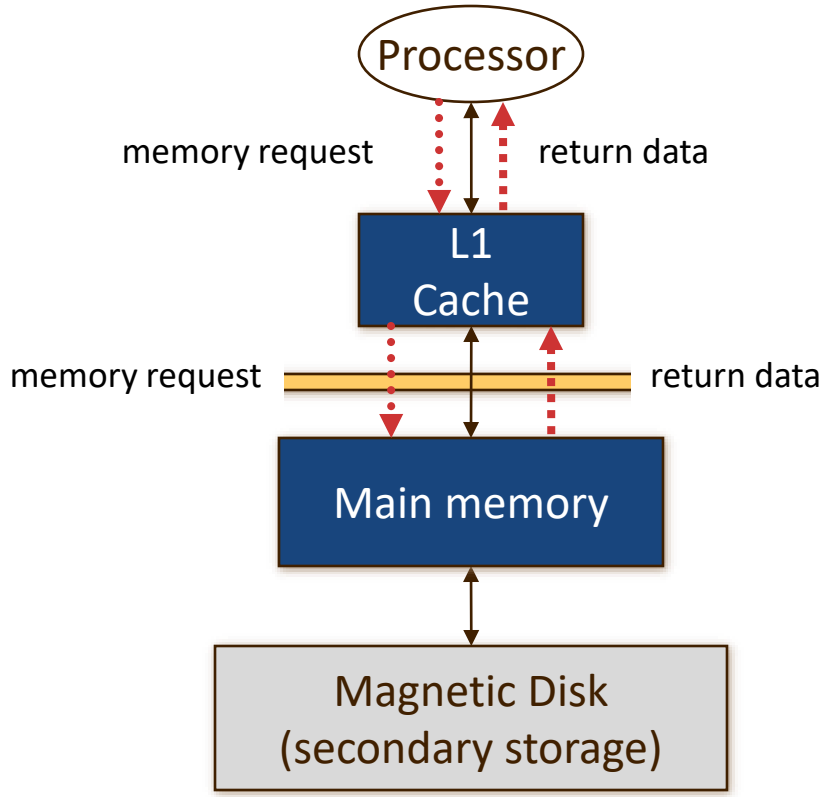


Parallel lookup for the requested address in the cache set

Handling Cache Accesses



Cache hit



Cache Miss

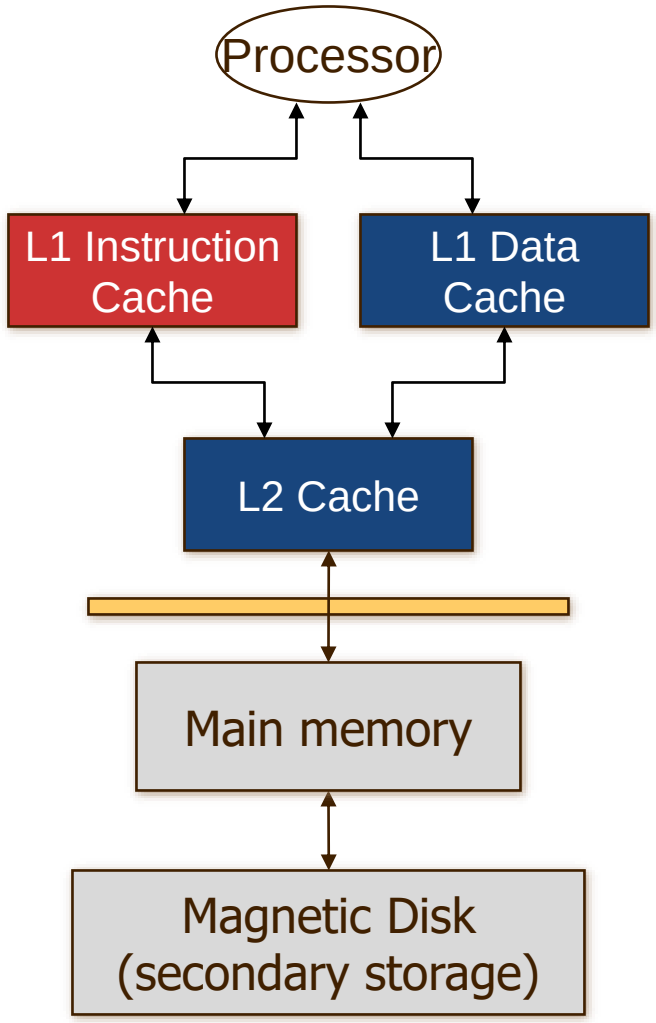
Caching the Instruction

Instructions are also stored in memory

- Upon execution of a program,
  CPU fetches instructions from memory

- To avoid long instruction fetch,
  same caching idea can be applied to instructions

➢ An example of instruction cache in a two-level cache hierarchy is shown in the figure



Example: Steps on an Instruction Cache Miss

- **Send the PC value to the memory**
- **Instruct main memory to perform a read and wait for completion**
- **Write the (instruction) cache entry, i.e.**
  - Put the data from memory in the data portion of the entry
  - Write the upper bits of the PC into the tag field
  - Turn the valid bit on
- **Restart the instruction execution at the first step**
- **Which will re-fetch the instruction, this time finding it in the cache**

- **Control of the cache on an instruction access is essentially identical:**
- **On a miss, simply stall the CPU until the memory responds with instr.**

Block Replacement

- When a cache miss occurs, we must decide which block to replace
- Why block replacement is needed?

Block replacement for

- Direct mapped: only one candidate (trivial case)
- Set associative:
  - Prefer non-valid entry (i.e. valid bit =0), if there is one
  - Otherwise, choose among entries in the set

Two primary replacement policies for associative caches

1. Random:
   - Candidate blocks are randomly selected to spread allocation uniformly
2. Least recently used (LRU):
   - The candidate is the block that has not been used for the longest time
   - Costly to implement for a degree of associativity higher than 2 or 4

LRU Replacement Policy

Upon miss:

- Replace the victim from the LRU position with miss address
- Move the miss address to the MRU position
- Heuristic: the LRU is not referenced for a long time, good candidate

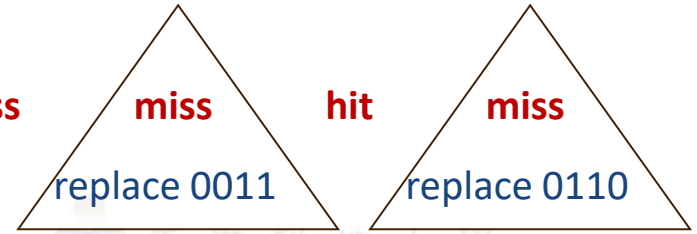
Upon hit:

- Move the hit address to the MRU position
- Then, pack the rest
- Heuristic: the one just got hit should be the last one to be replaced

Example of LRU Replacement Policy

- Assume the cache is 4-way set-associative
- Consider block address stream below (from left to right):
- 0011010, 0010010, 0110010, 1000010, 0001010, 0010010, 0100010
- All these block addresses mapped to same set “010”
- Question: which blocks remain in the set at the end?

MRU →	—	0011	0010	0110	1000	0001	0010	0100
	—	—	0011	0010	0110	1000	0001	0010
	—	—	—	0011	0010	0110	1000	0001
LRU →	—	—	—	—	0011	0010	0110	1000
		miss	miss	miss	miss	miss	hit	miss

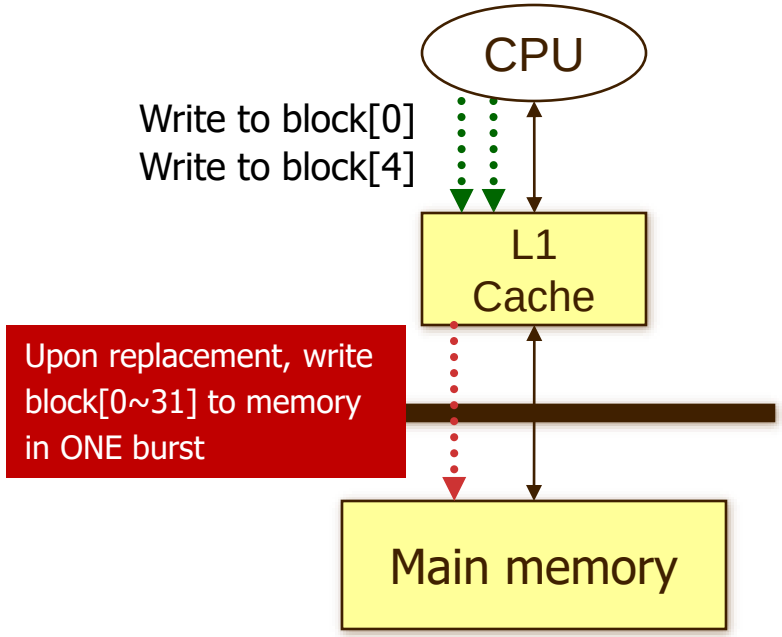


Dealing with Dirty Data

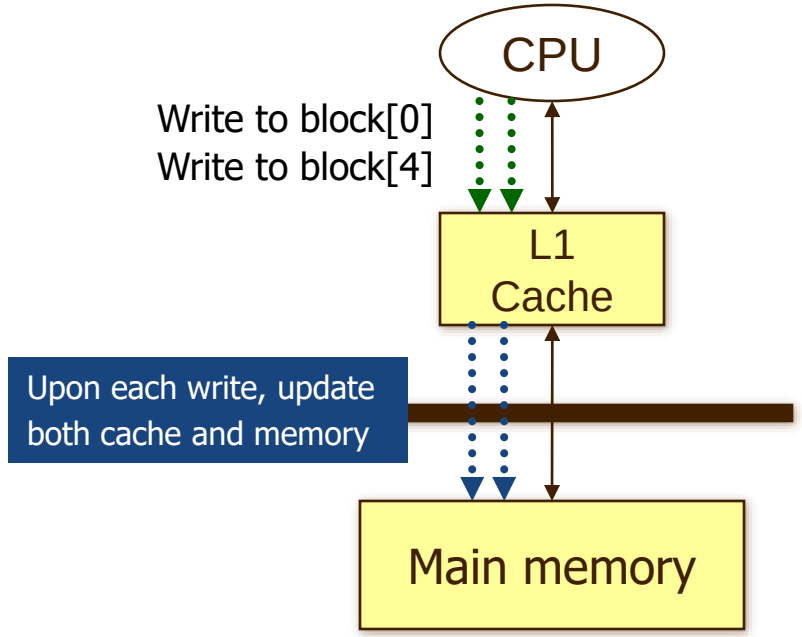
- If the content of a cache entry is modified, it is considered as **dirty**
- What should we do upon block replacement?
- It is a question about how we update the memory

- Two options:
  - Write-back
  - Write-through

Write-Back vs Write-Through



Write-Back



Write-Through

Write-Back Strategy

Upon write access by processor

- The information is written only to the block in the cache, not memory

When the block becomes the candidate of replacement

- The dirty block is written back to the main memory

Advantages

- Processor can write individual words at the rate of the cache (not memory)
- Multiple writes to a block are merged into one write to main memory
- Since the entire block is written, system can make effective use of a high bandwidth transfer

- As CPU speed increases at a rate faster than DRAM-based memory, more and more caches use the write-back strategy

Write-Through Strategy

Upon all memory accesses

- **The information is written to both the block in the cache and memory (memory is always up-to-date)**

Advantages

- **Handling of misses is simpler and cheaper**
  - Because they do not require a block to be written back to memory
- **Write-through is easier to implement than write-back**

Disadvantages

- **Multiple writes to the same location consume memory bandwidth**
- **Waste of bandwidth as compared to write-back strategy**
- **Potentially slow down the reads to memory**

Notes: A write buffer is needed for a high-speed system if write-through strategy is used
A write buffer is a queue to hold data while data are waiting to be written to memory

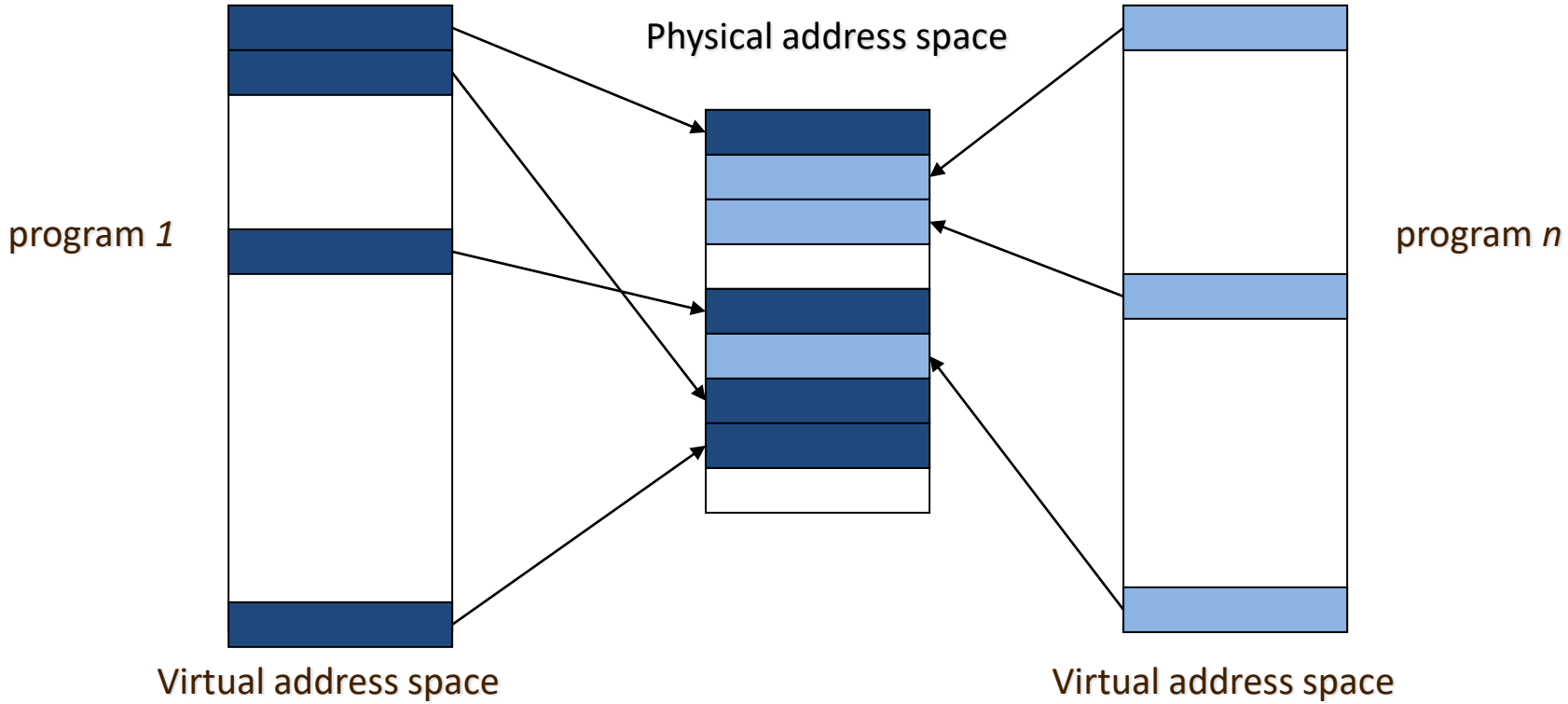
VIRTUAL MEMORY (OPTIONAL)

Motivation

- **Protection**
  - Ensure that programs cannot interfere with each others
- **Sharing of memory between programs to increase memory utilization**
  - As running programs only actively use a fraction of the memory
- **Allowing a program to exceed the size of the main memory**
  - Use secondary storage (e.g. magnetic disks) to backup
  - i.e. make use of the main memory as a "cache" for magnetic disks
- **In systems today, memory address in our programs is considered as virtual address**

Virtual memory is the technique to seamlessly map virtual addresses to physical addresses
(seamlessly = automatically map in hardware)

Basic Concept



Virtual Address vs. Memory Address

The processor generates virtual addresses

- While the memory is accessed using physical addresses

That means, programs see the virtual address space

- While the system sees the physical address space
- Virtual address space is as big as a register can address

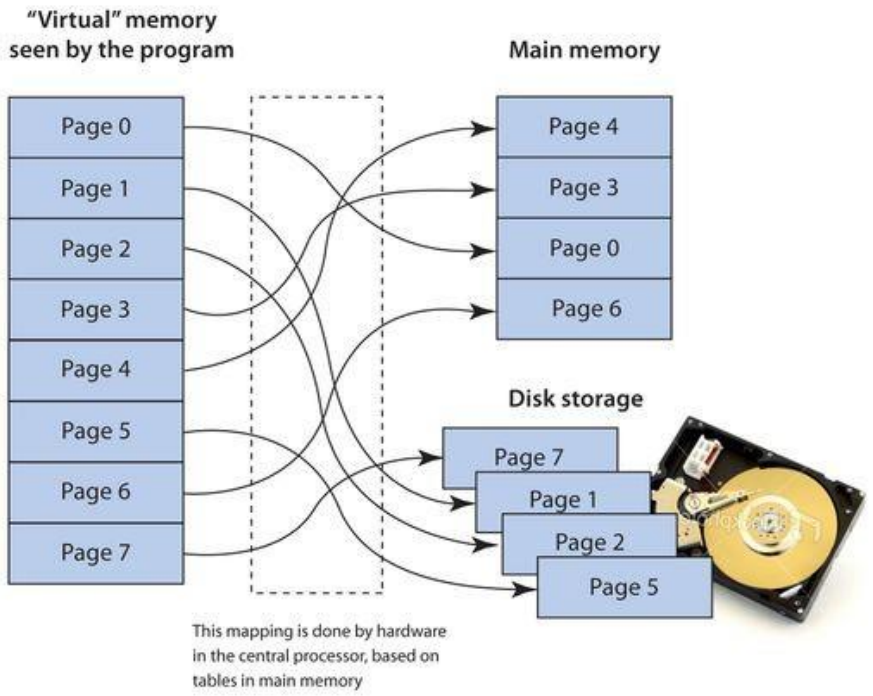
Ideally, physical address = mapping_function(virtual address)

- Mapping function can be implemented as a table in system memory
- But, if we map at word or block level, the table is too big!
- Instead, split the virtual and physical address space into pages

- A typical page size is 4Kbytes
- Then, the mapping is between virtual and physical pages

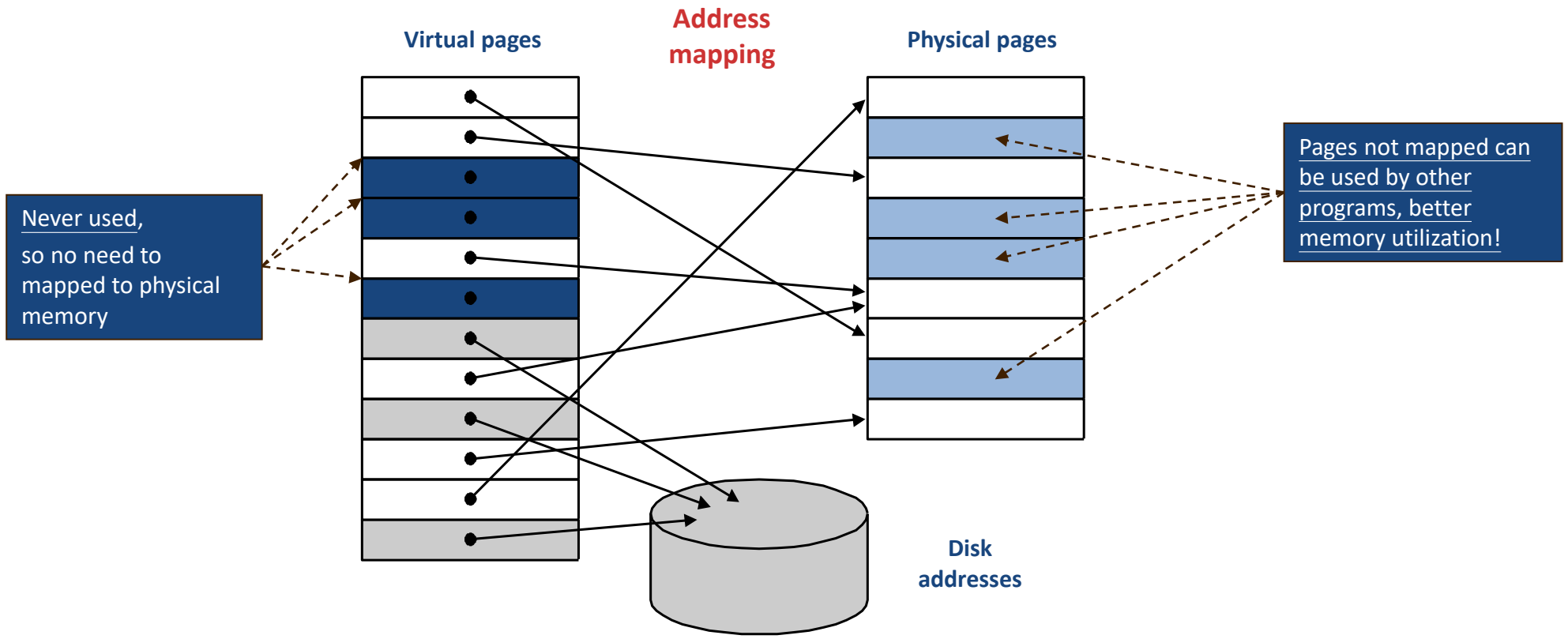
More precisely

- Portion of virtual memory, when not currently in use, is stored in secondary storage
- When it is requested later, OS **shuttles** it into the memory, **replacing** other portions not currently in use (replacement policy answers this part)



How does virtual memory work?

- Virtual page size = physical page size

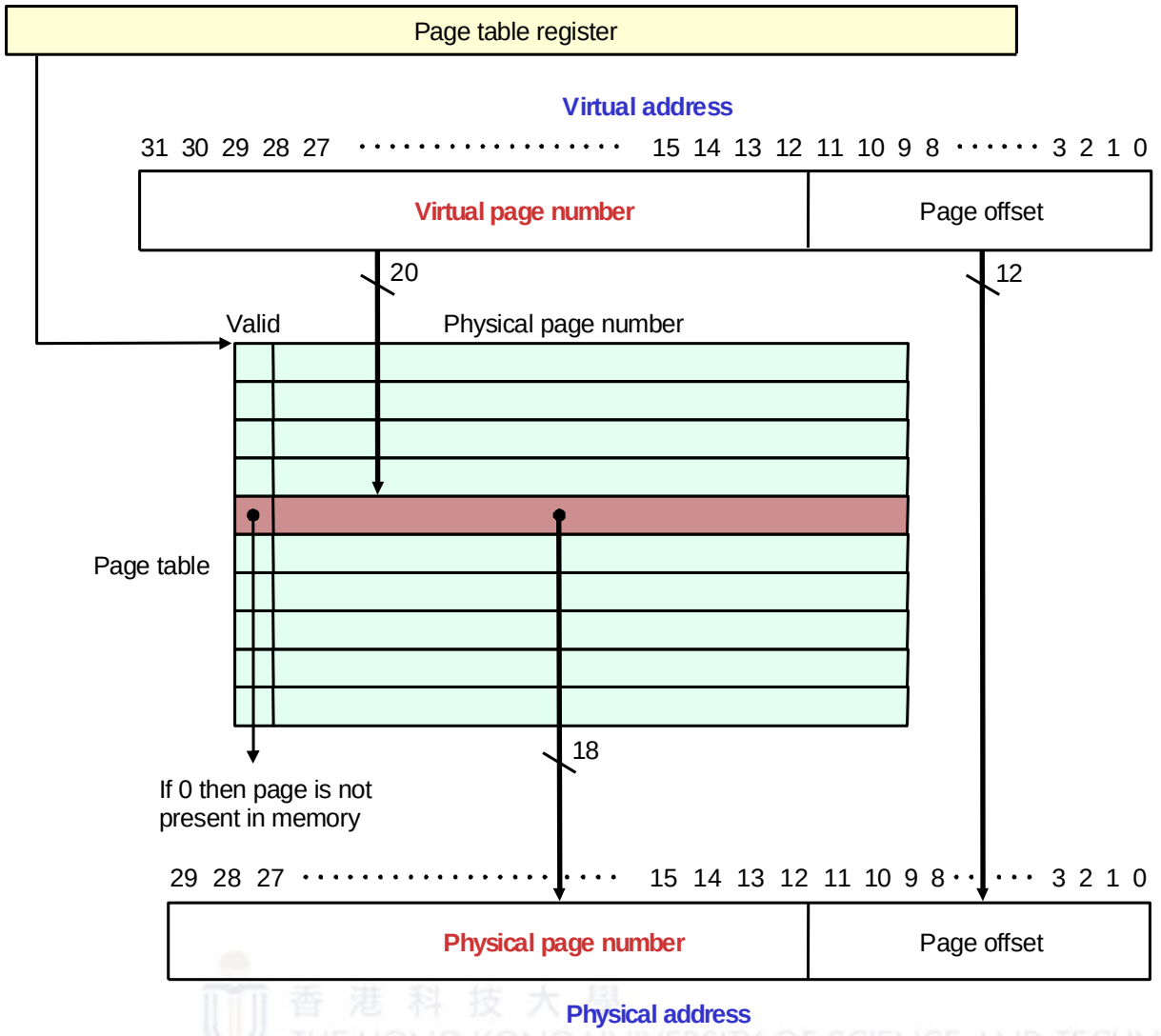


Page Table

- Mapping virtual addresses to physical addresses through a **page table**
- It is a structure that **resides in the memory**
- Starting address of the page table is stored in the **page table register**
- Each page table entry stores
  - a valid bit to indicate if the mapping exists, and
  - the corresponding physical page number
- Since every possible virtual page is represented in the page table,
- There is **no need to have a tag field**

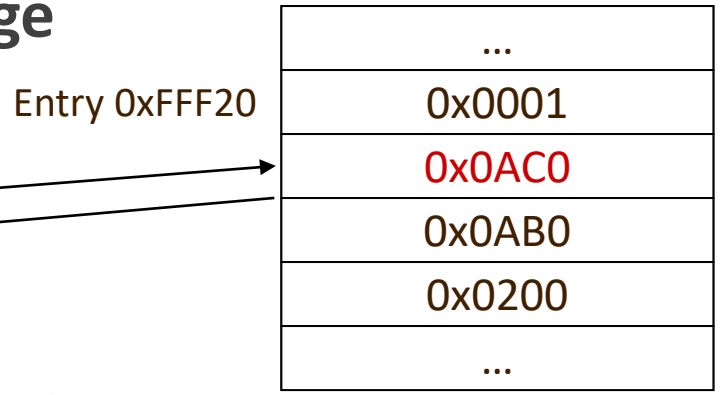
Address Translation

- Virtual address space: 4 GB (2^{32})
- Maximum main (physical) memory size: 1 GB (2^{30})
- Page size: 4 KB (2^{12})



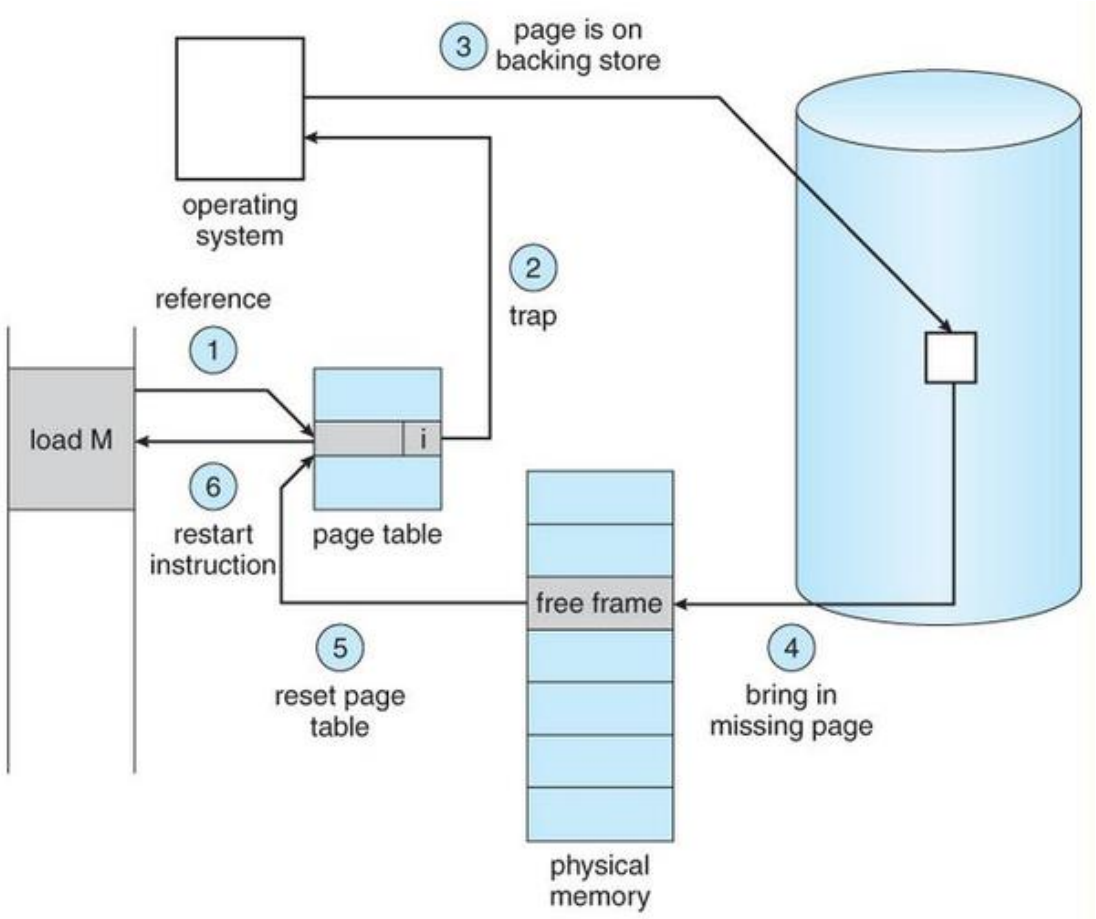
Example

- **Page size = 4Kbyte (2^{12})**
- **Virtual address space = 2^{32} bytes**
- **Physical address space = 2^{28} bytes**
- **What are the sizes of the virtual and physical page number?**
  - Size of virtual page number = 32 – 12 = 20 bits
  - Size of physical page number = 28 – 12 = 16 bit
- **What is the physical address for 0xFFF21340 using page table below?**
  - 0xFFF21340 = 1111 1111 1111 0010 0001 0011 0100 0000
  - Virtual page number = 0xFFF21
  - Physical address = 0x0AC0340



Page Faults

- It is possible that a mapping does not exist upon first access
- Try to map through the page table results in a page fault
- **Operating system is invoked to resolve the page fault**
  - Resolve means find the mapping or setup appropriate mapping
- **A page fault usually has an enormous penalty**
  - Page faults are handled in software
  - It can take millions of clock cycles to process
  - Dominated by the time to get the first word for typical page sizes
  - Program execution is stalled until page fault is resolved
- So, **pages should be large enough**
  - Too small the page size, too often we see page faults
  - Too large the page size, higher chances of page fragmentation



Fast Address Translation

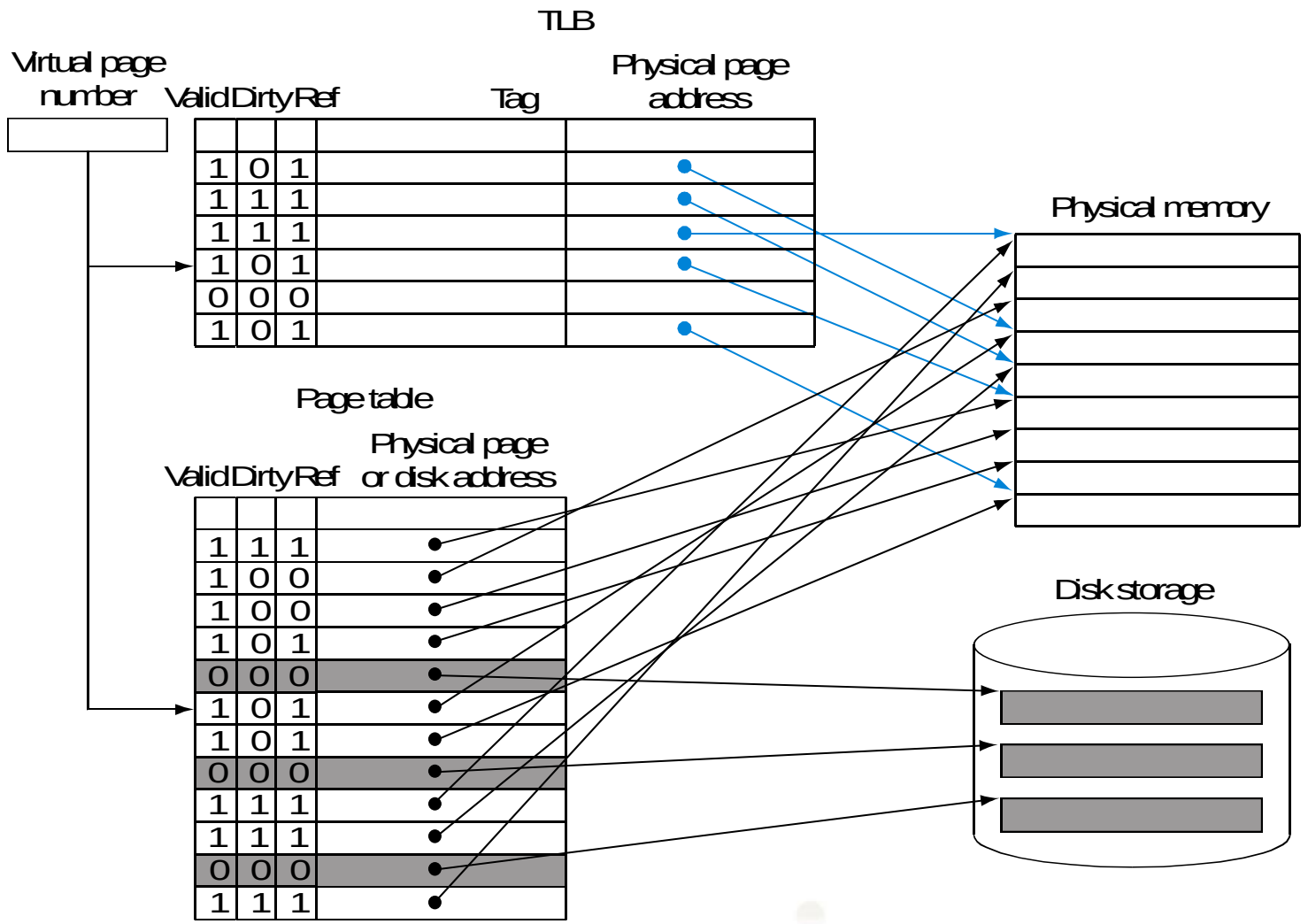
Problem with pure page table approach

- **Page tables are in main memory**
- **Every memory access by a program can take at least twice as long**
  - One memory access to obtain the physical address
  - The second access to get the data
- **Bad performance!**

Solution

- **Translation-lookaside buffer (TLB), a cache copy of the page table**
  - TLB relies on the locality of reference to the page table
  - When a translation for a virtual page number is used, it will probably be needed again in the near future as the references to the words on that page have both temporal and spatial locality
- **i.e. TLB is a special cache keeping track of recently used translations**
- **TLB is usually a small fully-associative cache, (e.g. 16~64 entries)**

Translation-Lookaside Buffer (TLB)



Working with TLB and Page Table

- **Upon each memory access**
  - Mapping for virtual address generated by CPU is first looked up in TLB
  - If found, do the translation and done
  - If not found, then looked up in the page table residing in memory

- **If mapping (i.e. translation) not found in the page table,**
- **Page fault!**
  - OS is invoked to handle the page fault
  - After OS resolve the page fault, the memory access is restarted

Key Knowledge Points

- **Ordinary programs exhibit two different notions of locality**
  - Temporal locality and spatial locality
- **Multilevel memory organizations achieve cost/performance tradeoff by exploiting the principle of locality**
- **Cache**
  - Data are transferred in blocks from main memory to cache upon misses
  - Block placement is direct-mapped, set-associative, or fully-associative
  - Block identification with tag and valid bit
  - Block replacement uses either random or least recently used (LRU)
  - The write strategy for caches is either write-through or write-back
- **Virtual memory**
  - The technique to seamlessly map virtual addresses to physical addresses
  - Needed for protection and efficient sharing of memory among programs
  - Mapping is implemented via a page table
  - TLB is cache copy of page table for the sake of performance