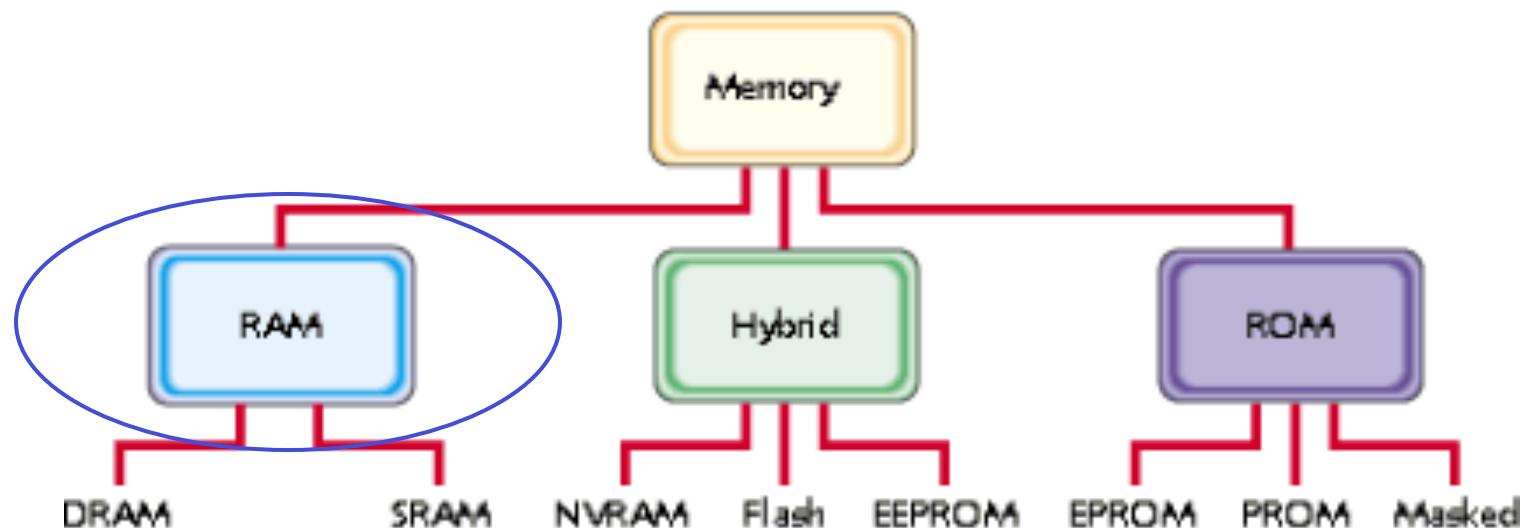
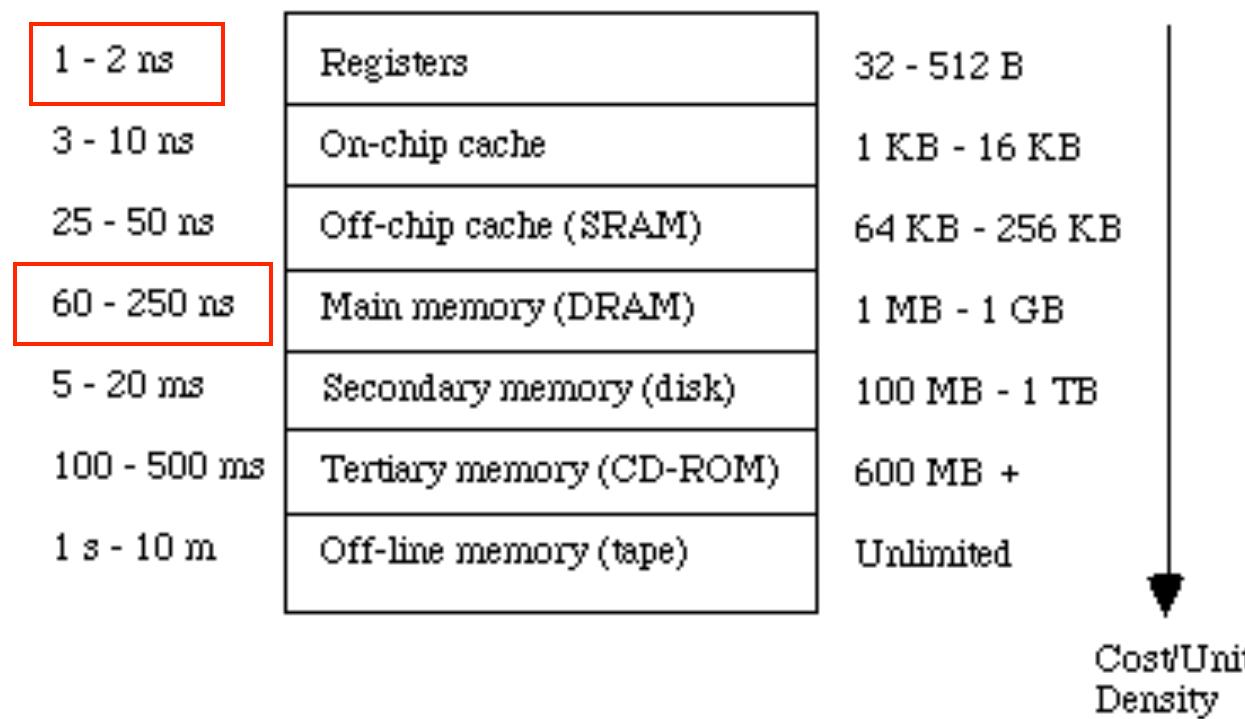


COMP2611: Computer Organization

Memory System



- To introduce the **memory hierarchy** and the **principle of locality**
- To introduce basic **cache** concepts
- To introduce basic concepts of **virtual memory**



RAM (Random Access Memory); two types of them:

- ❑ DRAM (Dynamic RAM) — people often refer to it as **memory**
- ❑ SRAM (Static RAM) — mostly used for **cache** (explained later)

From now on, we explicitly distinguish memory and cache

Why memory?

- ❑ Large storage for instructions and data of an executing program

Why do we choose DRAM, not SRAM, for memory?

- ❑ Because memory has to be large and cheap (so as to be affordable)
 - The implementation has to allow a **high density**
- ❑ DRAM provides this feature

How does a DRAM look like?

- Consists of bits of data stored in a separate **capacitors**
- Hold a logical “1” by having capacitor charged, and vice versa

Why capacitors?

- **Structurally simple**, only one transistor and a one capacitor per bit
- Hence, we are able to reach **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”
 - ⇒ To read the bit, discharge the electrons ⇒ read is destructive
- Because of this refresh requirement, we call it a **dynamic** memory

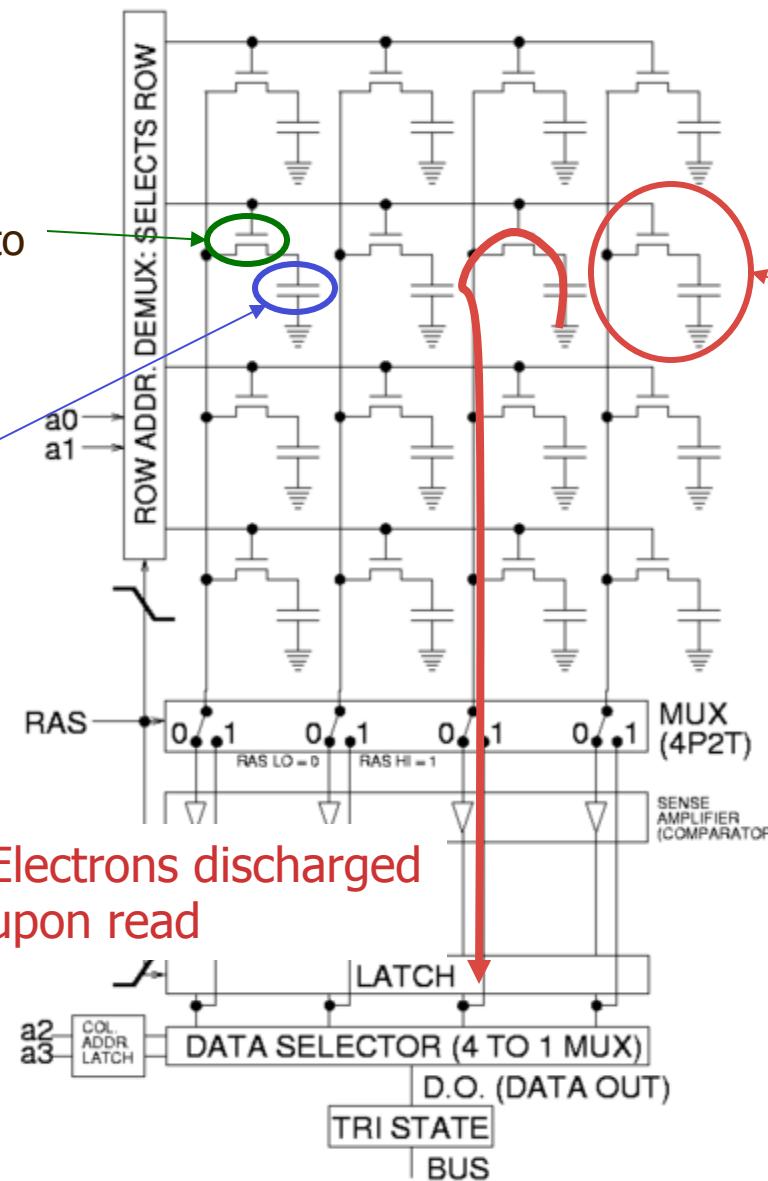
Example of a 4x4 DRAM Array

Transistor:

To enable read/write to the capacitor

Capacitor:

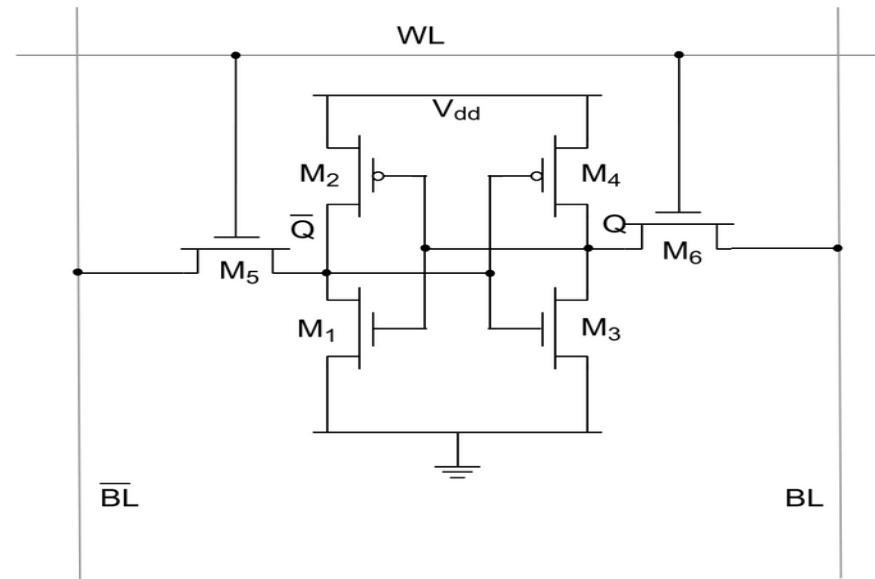
To store the “value”



A memory cell.

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

- SRAM is a type of **semiconductor memory**
 - Each cell is constructed using transistors only (i.e. no capacitor)
 - So, density not as high as DRAM \Rightarrow it is more expensive
- + Unlike DRAM, SRAM 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



- SRAM \neq SDRAM; SDRAM \equiv Synchronous DRAM

DRAM

- ❑ **Slow**; because capacitor charging and discharging times are long
- ❑ Can be as long as hundreds of cycles w.r.t. processor's speed

SRAM

- ❑ **Fast**; because switching time at transistor level is short
- ❑ May take a few cycles to a few tens of cycles, depending on the size
- ❑ Mostly used as caches inside the CPU

Before: discussions always assumed that memory access takes one cycle

From now on, this assumption is dropped

- ❑ A memory access instruction NO LONGER takes 5 cycles
 - ❑ Implication ⇒ performance of processor looks bad with this reality
-
- In this chapter, we'll look into ways to **minimize** the impact of this

Example

- 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
addi    $s1, $s1, -1
bne    $s1, $zero, L1

```

- Multicycle design: 4 cycles for **addi** & **add**; 3 cycles for **bne**
- Before, in **the ideal case**, each memory access took **1** cycle
 - Execution time = $4 + 4 + \{(1 + 4) + 4 + 4 + 4 + 3\} * 100 = 2008$
- Now, **in reality**, assume that each memory access takes **100** cycles
 - Execution time = $4 + 4 + \{(100 + 4) + 4 + 4 + 4 + 3\} * 100 = 11908$
- With “real” memory, slowdown by $1 - 2008 / 11908 = \boxed{-83\%}$
 - **So, it is critical to find a way to bridge this performance gap**

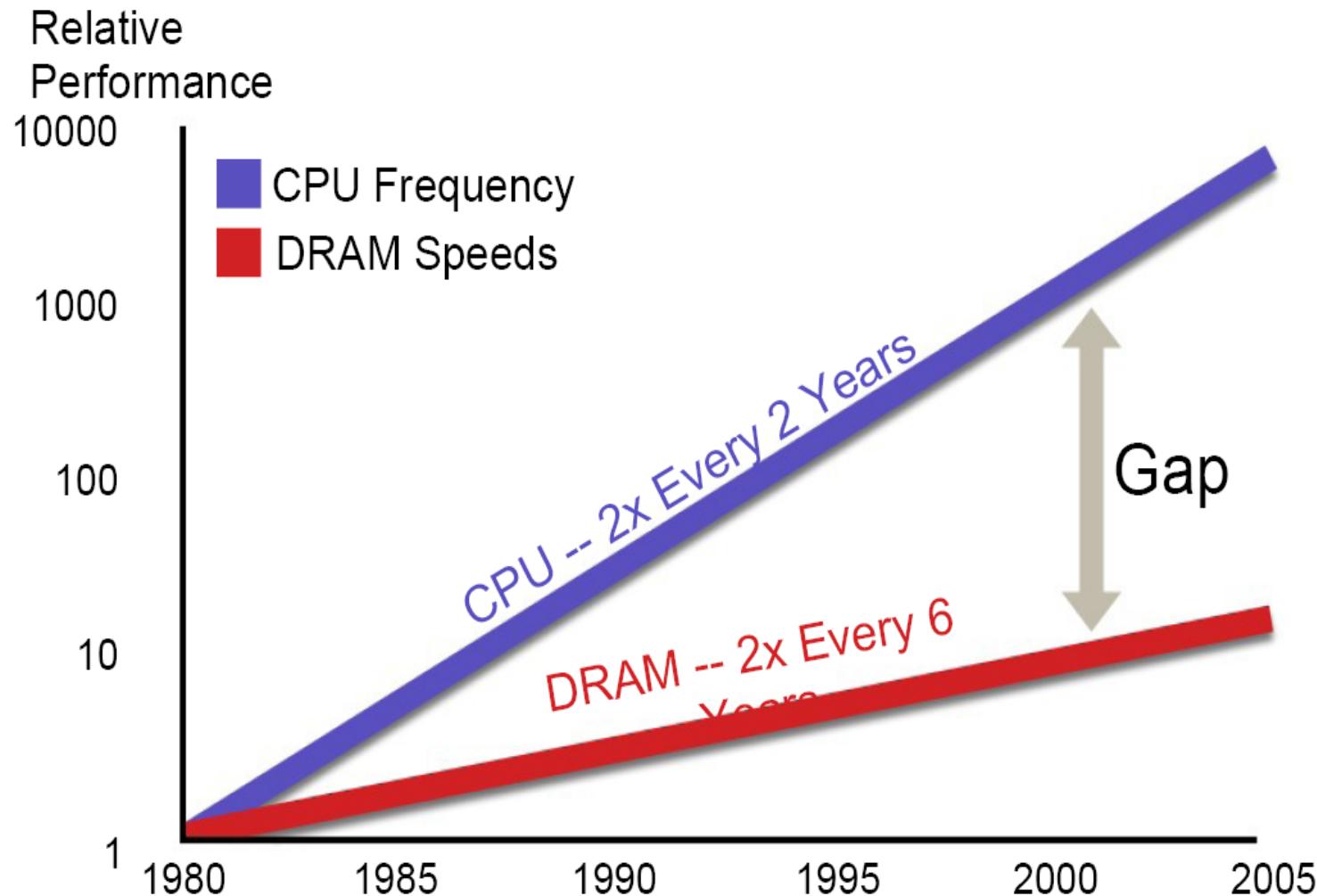
Today, what is worse:

- ❑ DRAM improvement is not keeping up with processor improvement
 - Processors are getting much faster than memories over time
 - The speed gap is widening!
-
- ❑ How about using SRAM?
 - Impractical & uneconomical to have a large amount of SRAM inside the computer
-
- ❑ As a result, memory access can be a bottleneck
 - Because processor spends long time waiting for memory access

Solution

Use a **memory hierarchy** exploiting the **principle of locality**

Memory Bottleneck



Programs usually access a relatively small portion of their address space (for instructions or data) at any instant of time

Such execution pattern exhibits two types of localities:

□ **Temporal locality:**

If an instruction or data item is referenced

⇒ it will tend to be referenced again soon

e.g. loops in programs contribute to temporal locality

□ **Spatial locality:**

If an instruction or data item is referenced

⇒ items whose addresses are close by will tend to be referenced soon

e.g. data arrays, sequential instructions contribute to spatial locality

This property is the KEY to memory hierarchy!

- Summing up 100 values stored in memory:

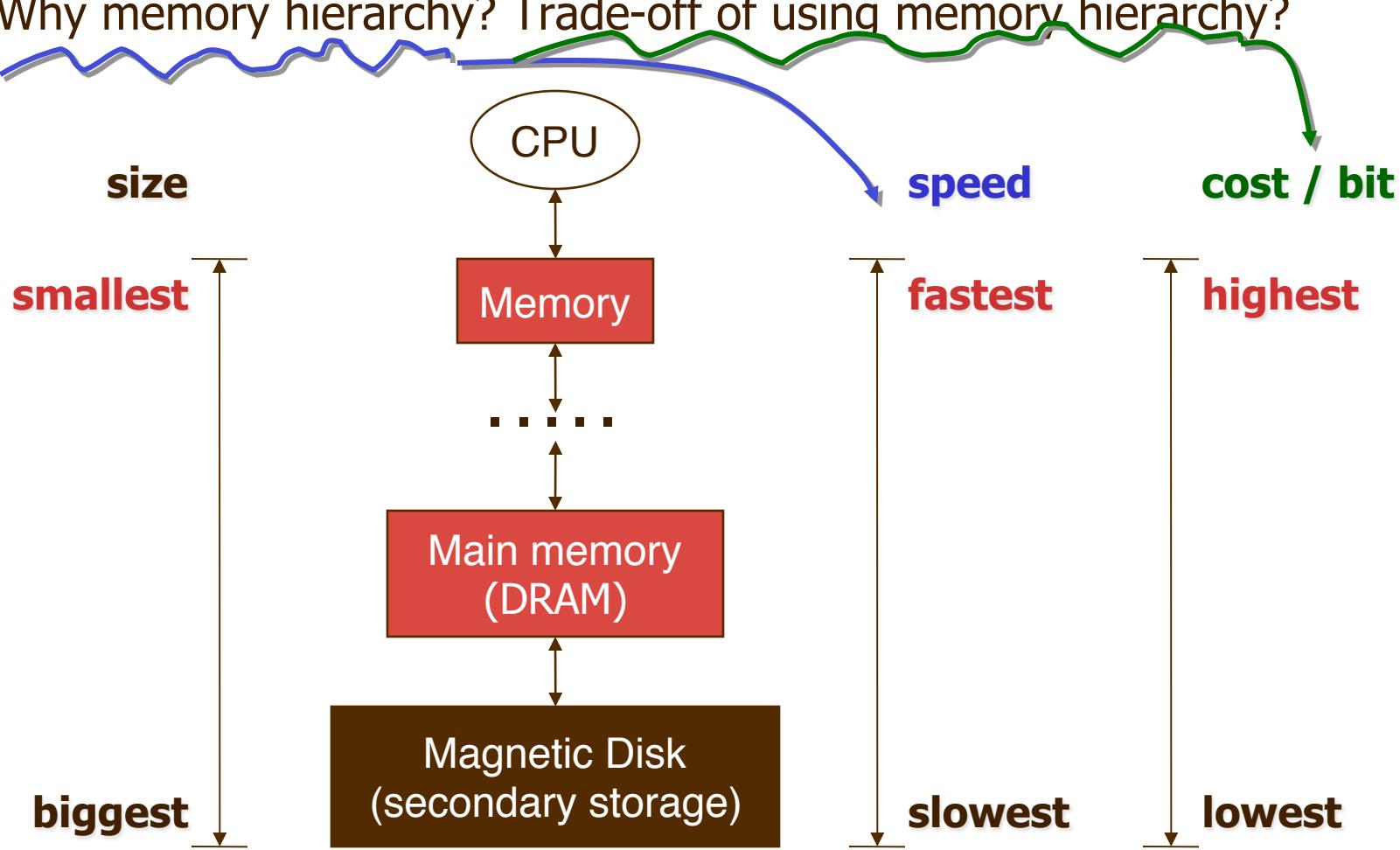
```
addi    $s0, $zero, 0      # $s0: accumulator
addi    $s1, $zero, 100    # $s1: counter
L1:   lw     $t1, 0($s2)    # $s2: memory addr
      add   $s0, $s0, $t1
      addi   $s2, $s2, 4
      subi   $s1, $s1, 1
      bne   $s1, $zero, L1
```

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

Memory Hierarchy

13

- A **memory hierarchy** consists of multiple levels of memory
 - Users have the **illusion** of a memory as large as the largest level
- Why memory hierarchy? Trade-off of using memory hierarchy?



Memory Technology	Typical access time	\$ per GB in 2008
SRAM	0.5-2.5 ns	\$2000-\$5,000
DRAM	50-70ns	\$20-\$75
Magnetic disk	5,000,000 – 20,000,000 ns	\$0.20-\$2

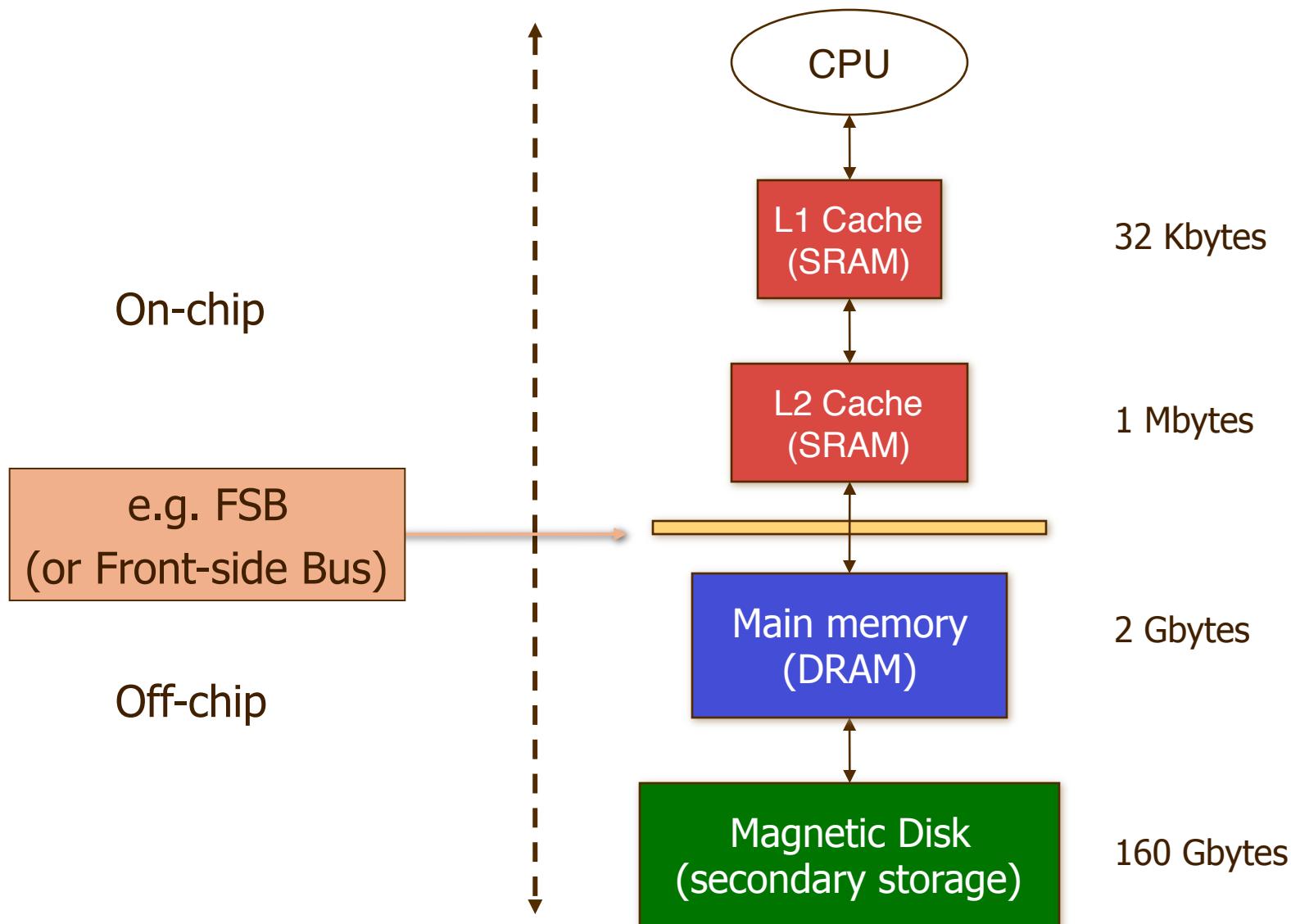
In memory hierarchy,

- Different levels of memory use different technologies
- Different technologies give different performance/cost trade-off
(example will be given later)

Cache was the name chosen to represent the level of the memory hierarchy between the CPU and the main memory in the first commercial machine

Example with Two Levels of Caches

15



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,

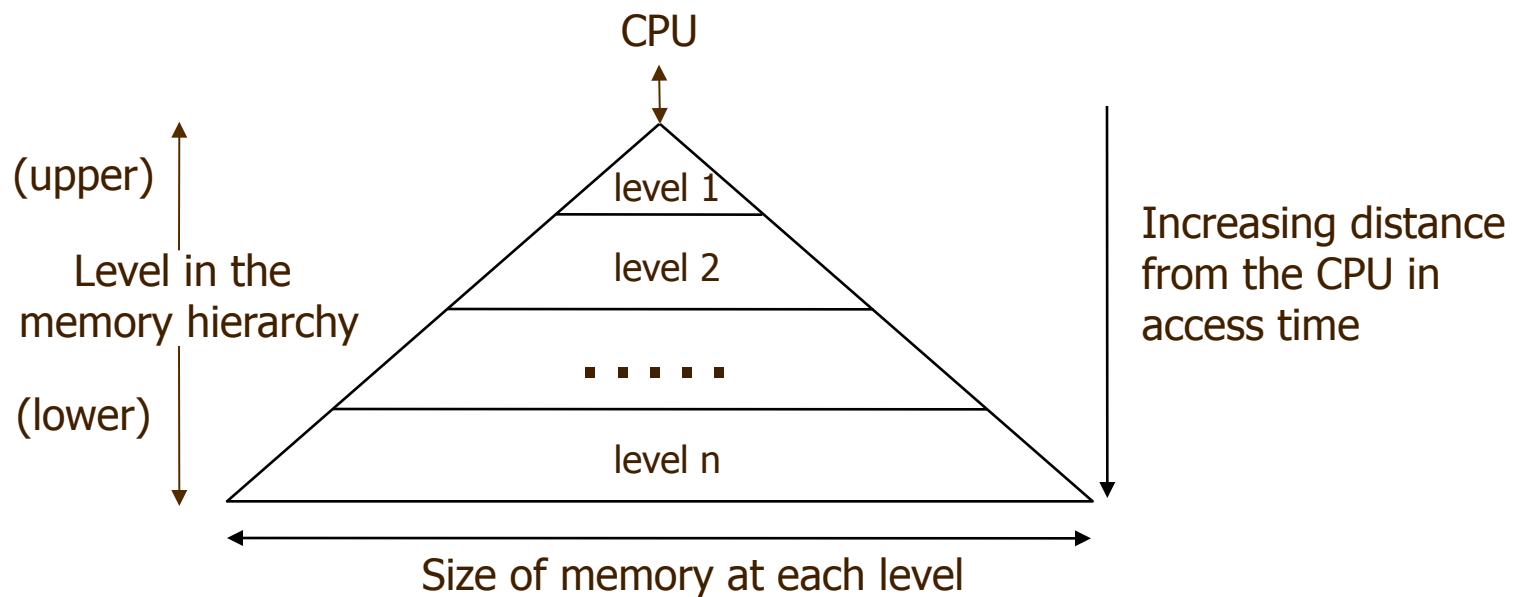
- ❑ First, look for the requested item in the cache
 - If the item is in the cache, return the item to the CPU
 - If NOT in the cache, look it up in the memory
- ❑ We can say
 - If not found in this level, look it up at next level until found
 - **Keep (or cache) a copy of the found item at this level after use**

Why Memory Hierarchy Improves Performance?

17

Memory hierarchy takes advantage of:

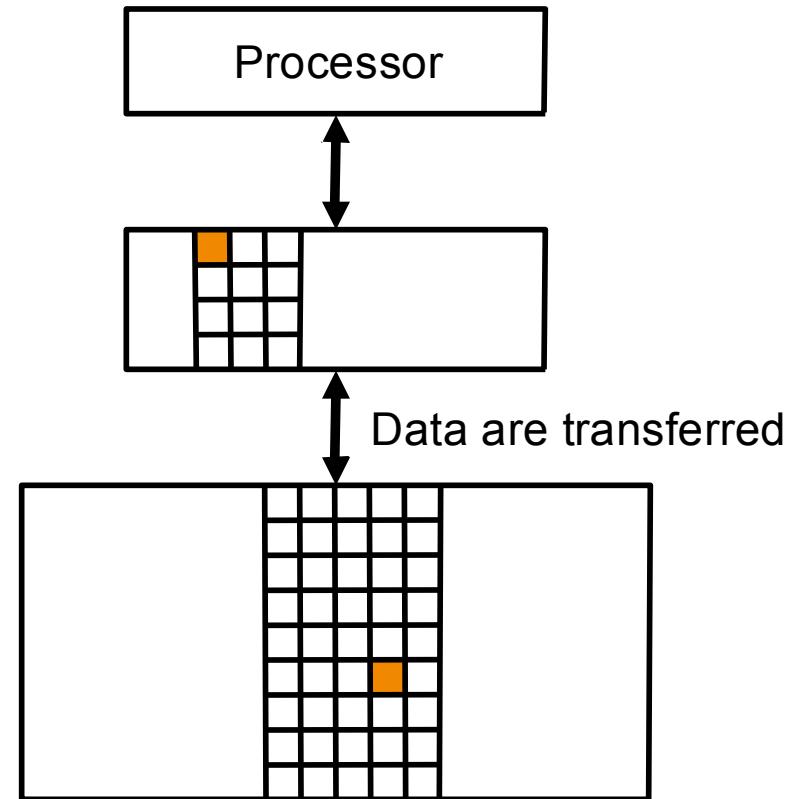
- ① **Temporal locality** by keeping more recently accessed data items closer to the processor
 - Increase the chance that the item is found in shorter time
- ② **Spatial locality** by moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy
 - Increase the chances that the close-by item is found in shorter time



Data Transfer Between Levels

18

- A level closer to the processor is a subset of any level further away
- Data are copied in **blocks** (block size is usually 32 bytes or 64 bytes) between only two adjacent levels at a time
- **Hit**: when the data item requested by the processor appears in some block in the upper level
- **Miss**: when the data item requested by the processor is not in the upper level; data transfer occurs from the lower level to the upper level



- **Hit rate** (or **hit ratio**):

- Fraction of memory accesses found in the upper level

- **Miss rate:** $1 - \text{hit rate}$

- **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**

Example with One Level of Cache

20

Example (assume everything else is ideal)

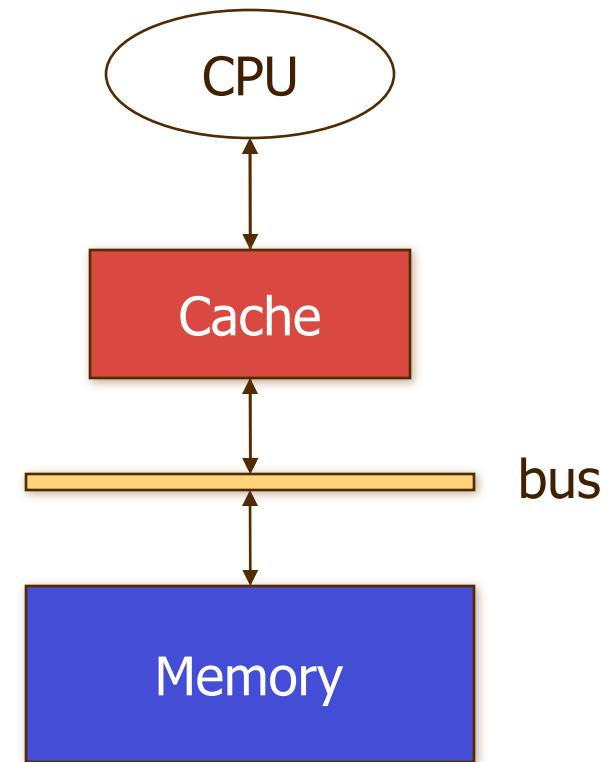
- ❑ 3-cycle cache access latency
- ❑ 100-cycle memory access latency
- ❑ Assuming 0-cycle for bus and cache lookup
- ❑ Data: 70% of the times in cache

Average memory latency

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

$$= 3 + 0.3 * 100$$

$$= \underline{33 \text{ cycles}}$$



Without cache, it is 100 cycles !!!

With caching, performance is better ☺ we are closer to the ideal case!

Example with Two Levels of Caches

21

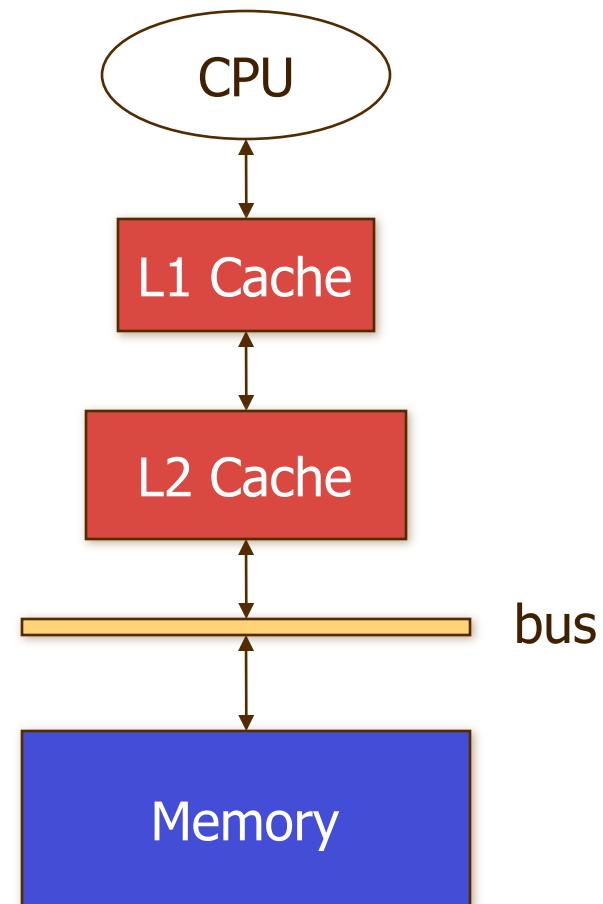
Example

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

Average memory latency

$$\begin{aligned} &= 3 + 0.3 * 12 + 0.3 * 0.4 * 100 \\ &= 3 + 3.6 + 12.0 \\ &= \underline{18.6 \text{ cycles}} \end{aligned}$$

(better than one level)



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
= hit time₁ + miss ratio₁ * miss penalty₁
= hit time₁ + miss ratio₁ * (hit time₂ + miss ratio₂ * miss penalty₂)
= hit time₁ +
miss ratio₁ * hit time₂ +
miss ratio₁ * miss ratio₂ * miss penalty₂

The same idea can be extended to multiple levels of caches

Suppose 1000 memory references

40 misses in the 1st-level cache and 20 misses in the 2nd-level cache

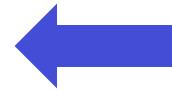
Hit time₁ = 3 cycles;

Hit time₂ = 12 cycles;

Memory latency = 100 cycles

- ❑ Miss rate₁ = $40 / 1000 = 0.04$ or 4%
- ❑ Miss rate₂ = $20 / 40 = 0.5$ or 50%
- ❑ Miss penalty₂ = memory latency
- ❑ Average memory access latency
 $= 3 + 0.04 * (12 + 0.5 * 100)$
 $= 3 + 0.04 * 12 + 0.04 * 0.5 * 100$
 $= 5.48$ cycles

2. Cache



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 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

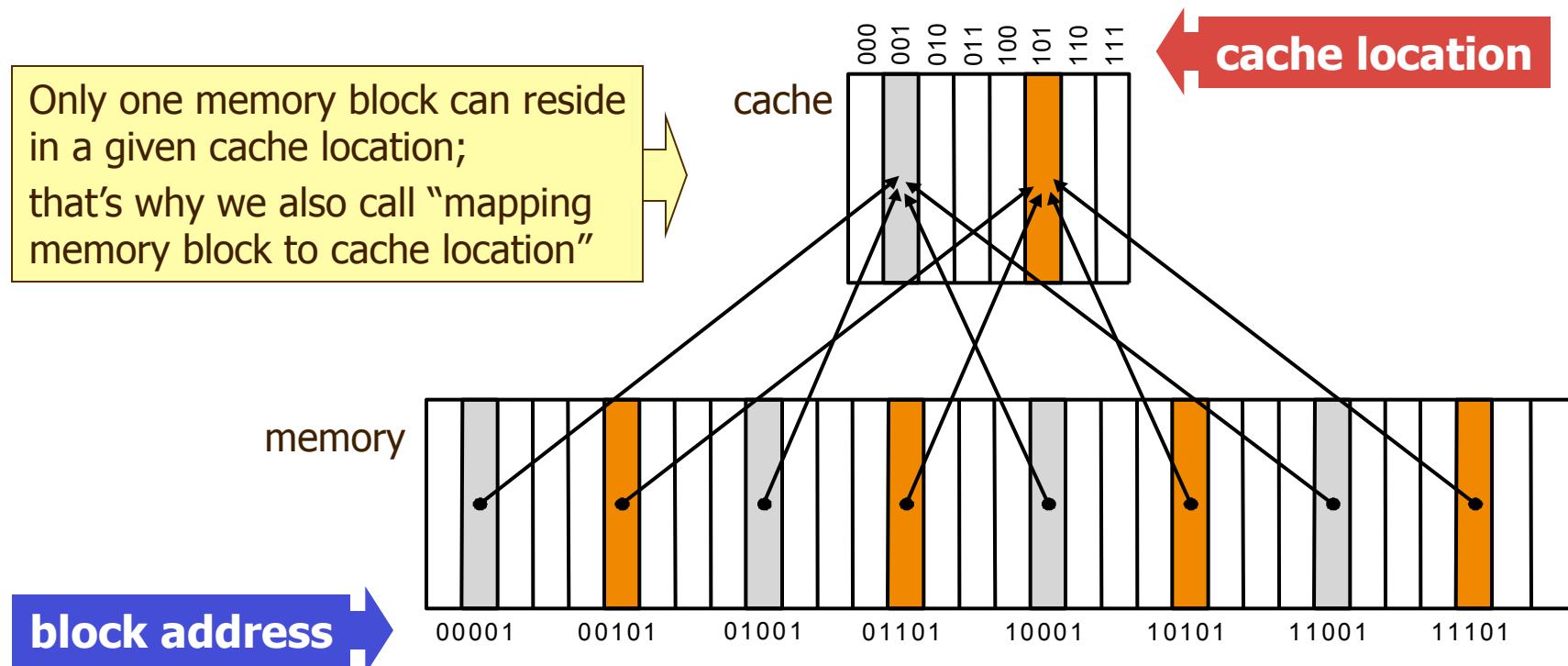
Three basic organizations

- ❑ **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)

A common mapping strategy:

`cache_location = block_address MOD number_of_blocks_in_cache`

- ❑ If the number of cache blocks (N) is a power of 2; $N = 2^m$
- ❑ Long-latency MOD operation can be avoided
- ❑ Instead, `cache_location` = the low-order m bits of block address



Example

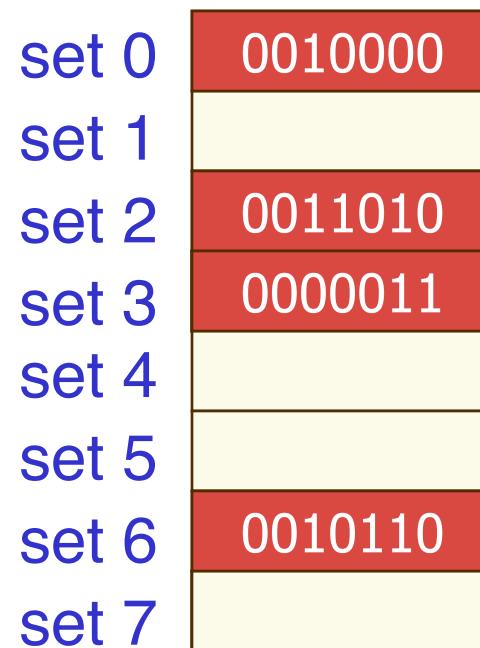
28

Consider block address stream below (from left to right):

0010~~110~~₂, 0011~~010~~₂, 0010~~110~~₂, 0010~~110~~₂, 0010~~000~~₂, 0000~~011~~₂, 0010~~000~~₂, ...

M **M** **H** **H** **M** **M** **H**

- ❑ Let's call cache location "set"
- ❑ Block address mapping to the frames:



M: cache miss

H: cache hit

8-entry cache

Mapping an Address to a Multiword Cache Block

29

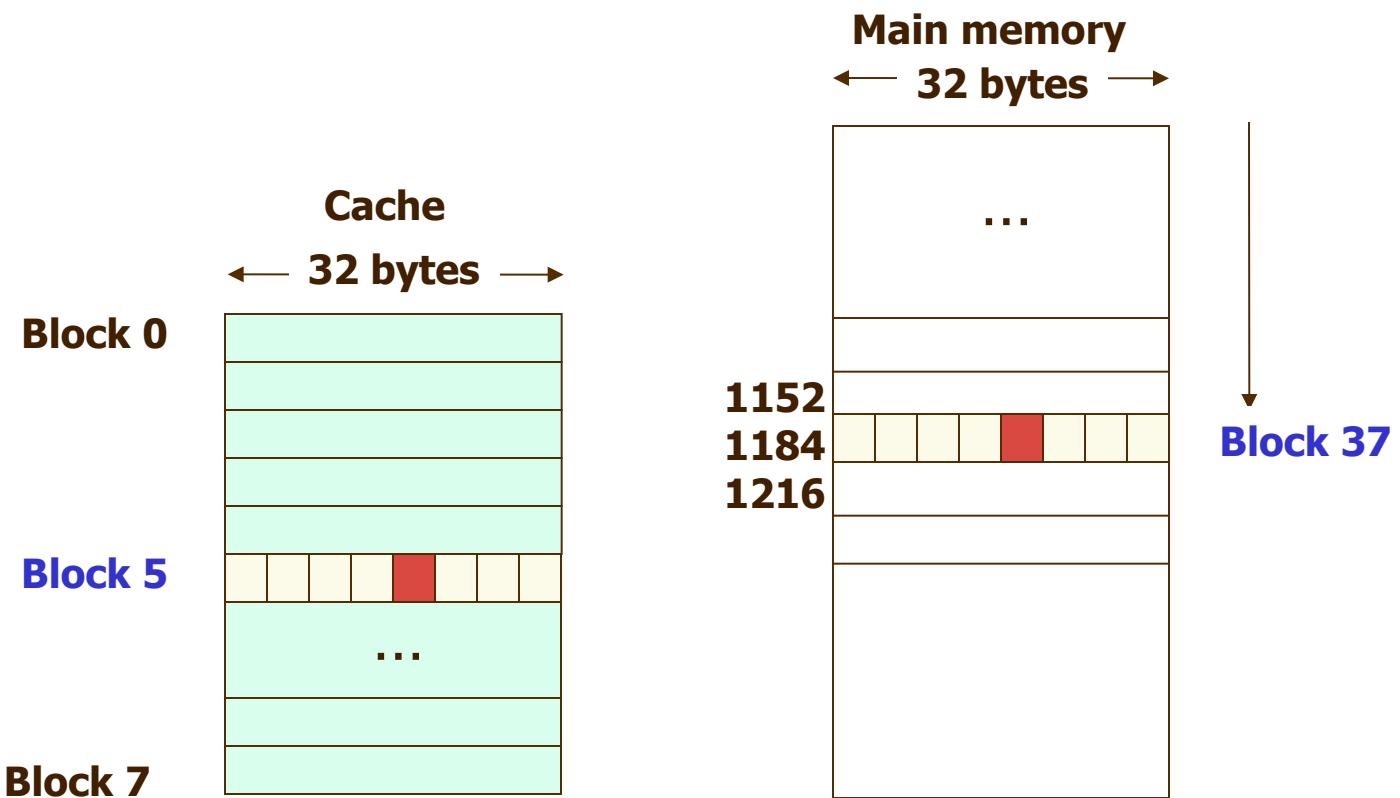
Consider a direct-mapped cache with

- ❑ 8 entries (or cache blocks) and a block size of 32 bytes
- ❑ What block number does **byte address** 1200 map to?

- First, find the **block address**, which equals to $\frac{\text{byte address}}{\text{bytes per block}}$
- i.e. block address for byte address 1200 = $\text{floor}(1200/32) = 37$
- ◆ The block address is the block containing all addresses between
$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \times \text{bytes per block} \quad \text{and}$$
$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \times \text{bytes per block} + (\text{bytes per block} - 1)$$
- Next, block location = **(block address) MOD (number of cache blocks)**
- i.e. map to cache block number $(37 \text{ MOD } 8) = 5$

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

- The memory address generated by CPU upon execution = **1200**
- **1200** will be first sent to the cache to look for the data
- By the mapping method discussed, the data block will be in entry **5**



Example: Accessing a Cache

31

Consider a direct-mapped cache with

- 8 cache frames and a block size of 32 bytes

Memory (byte) address generated by CPU	Block address	Hit or miss in cache	Assigned cache block (where found or placed)
0010 1100 0010 ₂	0010 110 ₂	Miss	(0010110 ₂ mod 8) = 110 ₂
0011 0100 0000 ₂	0011 010 ₂	Miss	(0011010 ₂ mod 8) = 010 ₂
0010 1100 0100 ₂	0010 110 ₂	Hit	(0010110 ₂ mod 8) = 110 ₂
0010 1100 0010 ₂	0010 110 ₂	Hit	(0010110 ₂ mod 8) = 110 ₂
0010 0000 1000 ₂	0010 000 ₂	Miss	(0010000 ₂ mod 8) = 000 ₂
0000 0110 0000 ₂	0000 011 ₂	Miss	(0000011 ₂ mod 8) = 011 ₂
0010 0001 0000 ₂	0010 000 ₂	Hit	(0010000 ₂ mod 8) = 000 ₂
0010 0100 0001 ₂	0010 010 ₂	Miss	(0010010 ₂ mod 8) = 010 ₂

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

Deciding Number of Sets in Direct-Mapped Cache

32

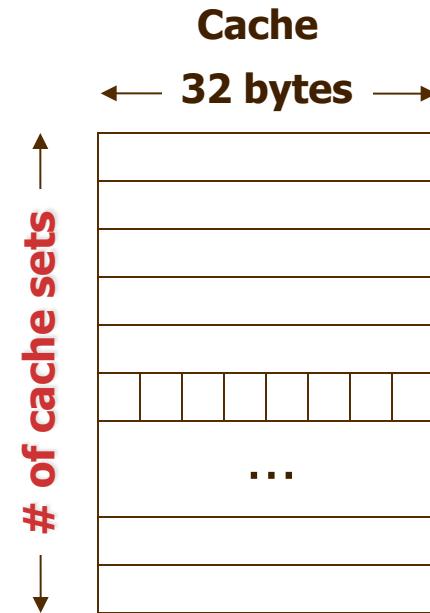
Example: Direct-mapped (DM) cache

- ❑ Cache size = 16 Kbytes
 - ❑ Cache block size = 32 bytes
 - ❑ N = # of blocks in cache
 - ❑ **# of cache sets = ? (in DM, N = Sets)**
- # of cache sets = $16K / 32$

$$= 2^{14} / 2^5$$

$$= 2^9$$

$$= 512$$

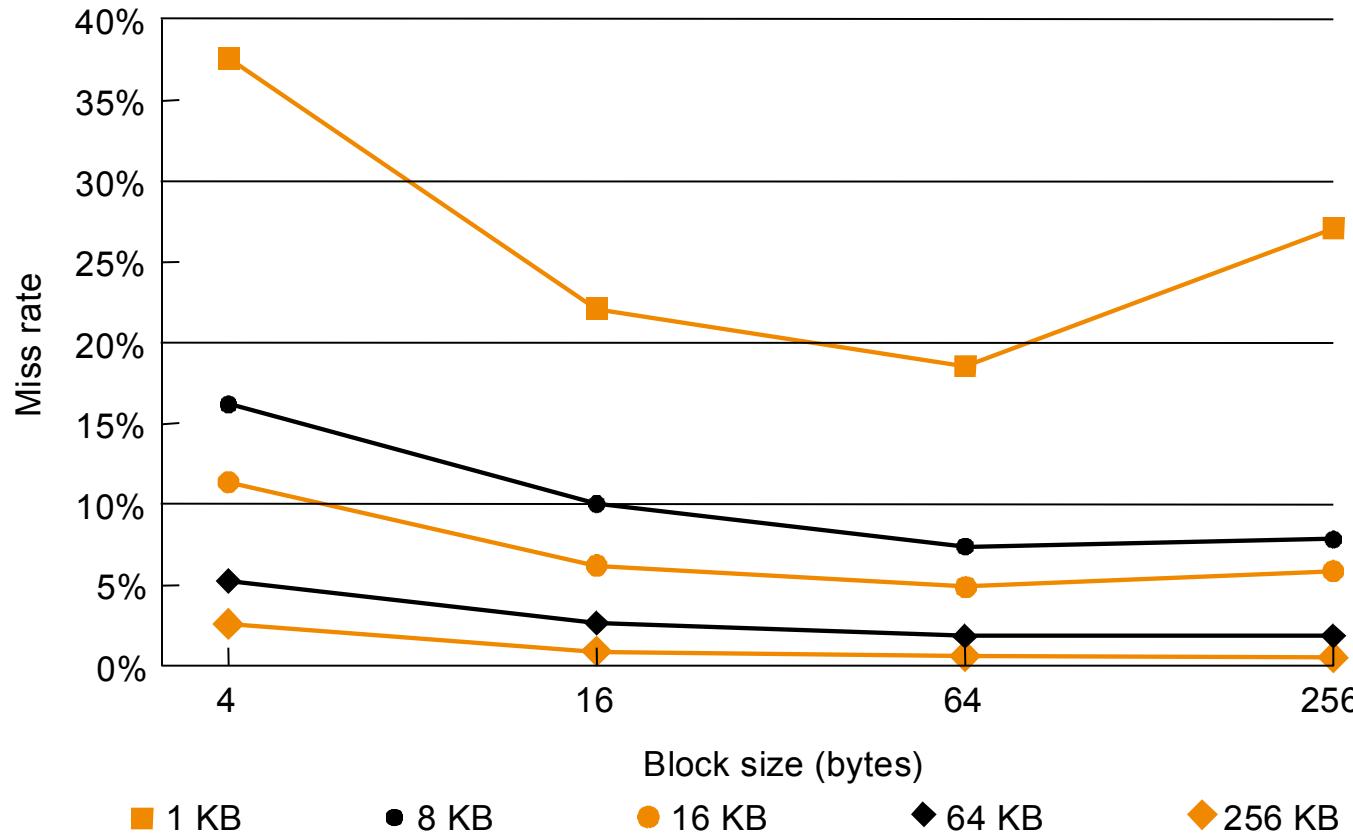


If the answer is 2^m , it implies that we need **rightmost m bits** of the **block address** to form the index (cache location) into the cache; in this example, we need **9** bits

Miss Rate vs. Block Size

33

- Bigger cache block size, less number of cache blocks; and vice versa
- Given cache size is fixed, varying cache block size changes miss rate



Blocks mapped to same cache frame can't be present simultaneously

Example:

- ❑ # of cache sets = 8
- ❑ Consider block access sequence: 100011_2 , 001011_2 , 100011_2
- ❑ Block 100011_2 & 001011_2 are mapped to the same cache set 011_2
- ❑ Arrival of block 001011_2 will kick (i.e. replace) 100011_2 out
 - i.e. both blocks can't be present simultaneously
 - Subsequent access to 100011_2 will **not hit** but **miss**
 - Less performance improvement with such **cache conflict**

Question:

- ❑ Can we alleviate this conflict problem to increase the cache hit rate?

Fully associative (FA)

- Each block can be placed **anywhere** in the cache
- **Advantage:**
 - No cache conflict \Rightarrow 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

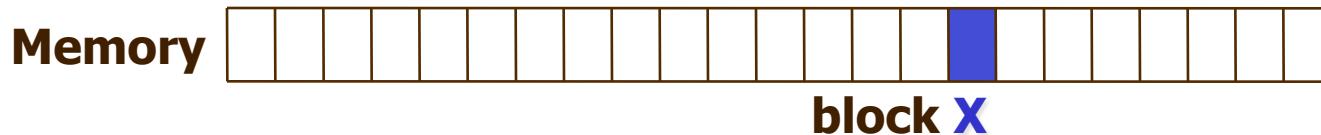
Set associative (SA)

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

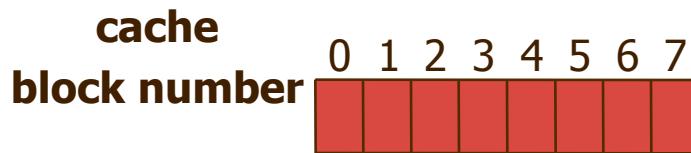
Fully Associative vs. Set Associative

36

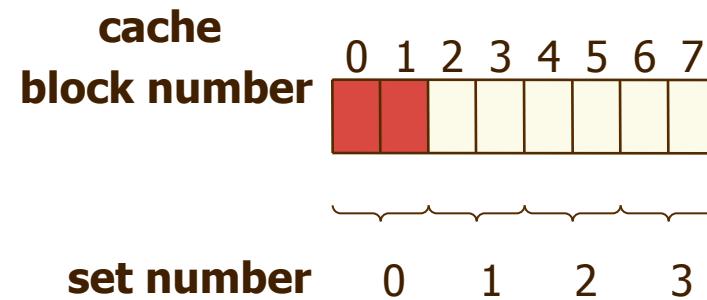
- Highlighted locations are placement candidates for memory block X



Fully associative cache



2-way set associative cache



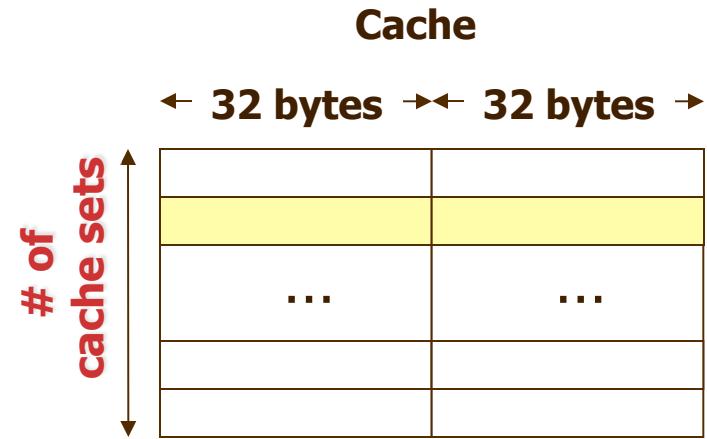
- Assume both FA and SA have same number of cache blocks

Deciding Number of Sets in Set-Associate Cache

37

Example: 2-way set-associative (SA) cache

- ❑ Associativity = 2
- ❑ Cache size = 16Kbytes
- ❑ Cache block size = 32bytes
- ❑ N = # of blocks in cache
- ❑ **# of cache sets** = ? (in SA, $N \neq \text{Sets}$)



- $N = 16K / 32 = 512$
- **# of cache sets** = $N / 2$
= 256

In general, for a m -way set-associative cache,

- ❑ $\# \text{ of cache sets} = \text{cache size} / \text{cache block size} / m$

Fully-associative (**FA**) cache is a special case of set-associative cache

- ❑ i.e. there is only **ONE set** in **FA** since a block can be placed anywhere

Example:

- ❑ Cache size = 16Kbytes
- ❑ Cache block size = 32bytes
- ❑ **N** = # of blocks in cache = $16K / 32 = 512$ (still the same calculation)
- ❑ **# of cache sets = 1**

An **N-way set associative cache**

- ❑ Consists of a number of sets, each of which consists of **N** blocks

Mapping strategy:

- ❑ `number_of_sets_in_cache` = cache size / cache block size / **N**
- ❑ `cache_location` = `block_address MOD number_of_sets_in_cache`

Placement of the memory block

- ❑ Can be in any “way” within the set
- ❑ e.g. if 4-way, there are 4 feasible locations to cache the block
(how to choose the location to place it will be answered later)

Special cases:

- ❑ A direct mapped cache  as a **1-way** set associative cache
- ❑ A fully associative cache with M blocks  an **M-way** set associative

Possible Associativity Structures

40

Example:

- Cache Size = 256 bytes
= 64 words
- Block Size = 32 bytes
= 8 words
- Number of blocks = 8

One-way set associative
(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data												

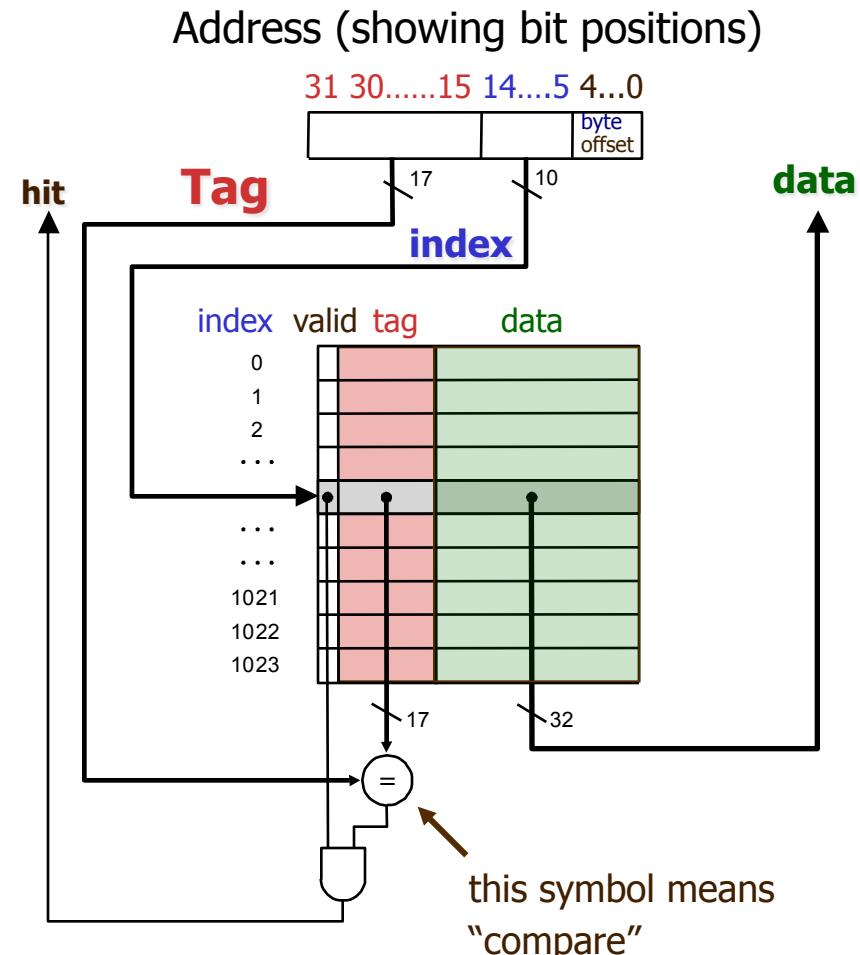
- **Increase** in degree of associativity ⇒
 - **Advantage: Decrease** in miss rate
 - **Disadvantage: Increase** in hit time (due to longer search time)

Block Identification in DM

41

- Assume a DM with 1024 cache sets
- Assume the block size is 32 bytes

- Each cache location can contain a block from a number of different memory locations
- Question: how do we know which block is actually in the location, i.e. how to tell **hit** or **miss**?
 - A **tag** is used to store the address information
 - The tag needs only to contain those high-order bits that are not used as an index into the cache
 - A **valid bit** is needed to indicate whether a cache block contains valid information



Example: Bits in a Cache

42

- A DM cache with 16KB of data and 8-word (2^5 bytes) blocks
- Assume a 32-bit address
- How many total bits are required?

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

Block size of 8 words $\rightarrow 2^9$ blocks

Each block has $8 \times 32 = 256$ bits of data

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

And, 1 valid bit

$$\text{Total bits per entry} = (256 + 18 + 1)$$

Total cache size

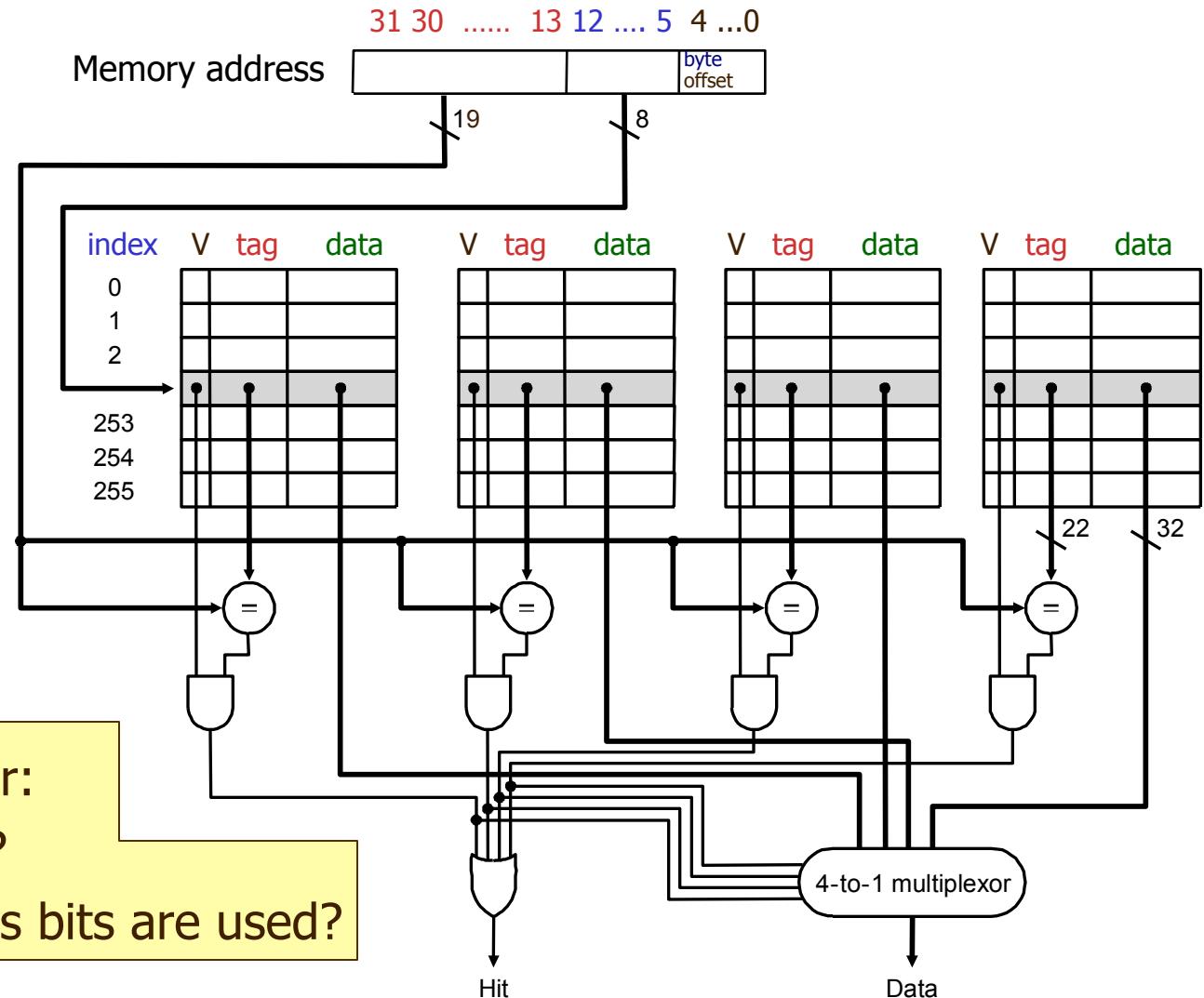
$$2^9 \times (256 + 18 + 1) = 2^9 \times 275 = 140800 = 137.5 \text{ Kbits}$$

- How much overhead is needed?

Block Identification in N-way Set Associative Cache

43

- Parallel lookup for the requested address in the cache set



Questions to answer:

- Which cache set?
- i.e. which address bits are used?

Recall the N-way SA's **mapping strategy**:

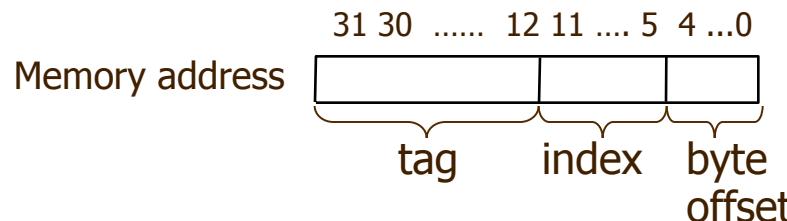
- ❑ `number_of_sets_in_cache` = cache size / cache block size / N
- ❑ `cache_location` = `block_address MOD number_of_sets_in_cache`

If `number_of_sets_in_cache` = 2^m ,

- ❑ Use next m address bits after the `byte offset` to serve as the index

Example:

- ❑ 16KB cache, 4 ways, 32-byte cache block
 - `number_of_sets_in_cache` = $16K / 32 / 4 = 128 = 2^7$
 - Byte offset = 5 (because $32 = 2^5$)
 - So, bit 0~4 are for byte offset, bit 5~11 are for index, the rest is tag



Upon a memory access by CPU, a request is sent to the first level cache

If the cache reports a **hit**

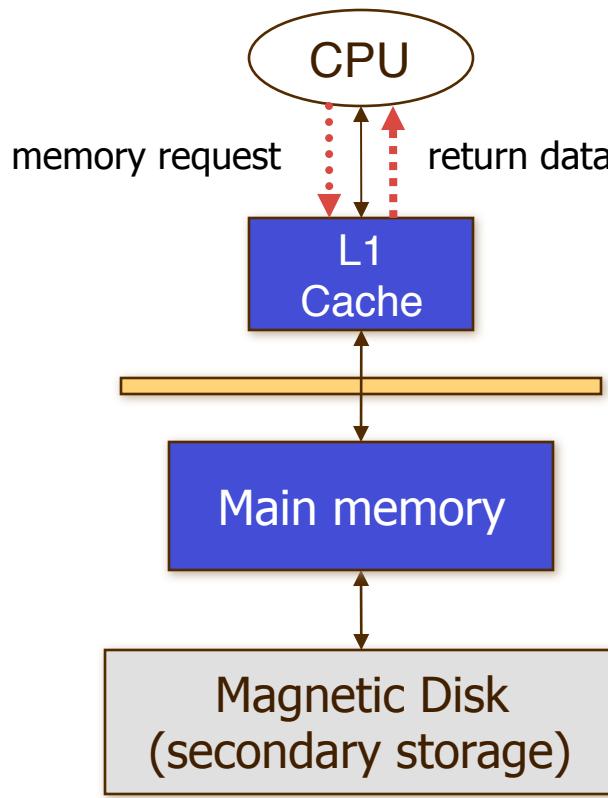
- ❑ CPU continues using the data from cache as if nothing had happened

If the cache reports a **miss**, some extra work is needed

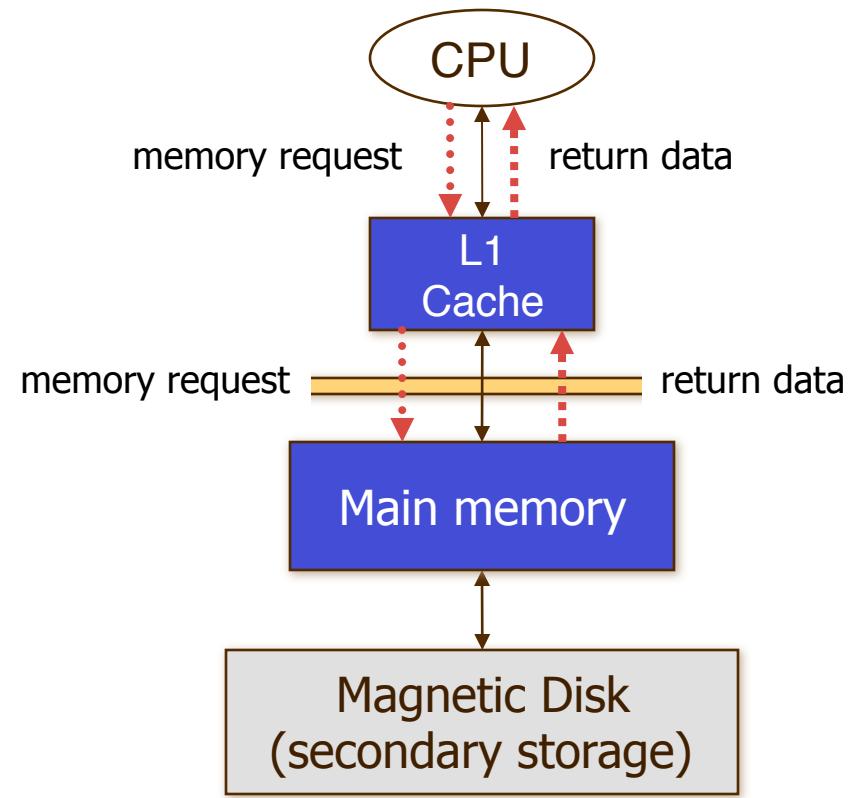
- ❑ A separate controller initiates a request to refill the cache
 - This request is sent to the next level cache or memory
(Keep going to next level until data are found)
- ❑ During the process of refilling, CPU is stalled
 - Stall ⇒ entire CPU stops operating until the 1st-level cache is filled
 - Unlike interrupts, stall does not need saving the state of all registers

Flow of Handling Cache Accesses

46



Cache hit



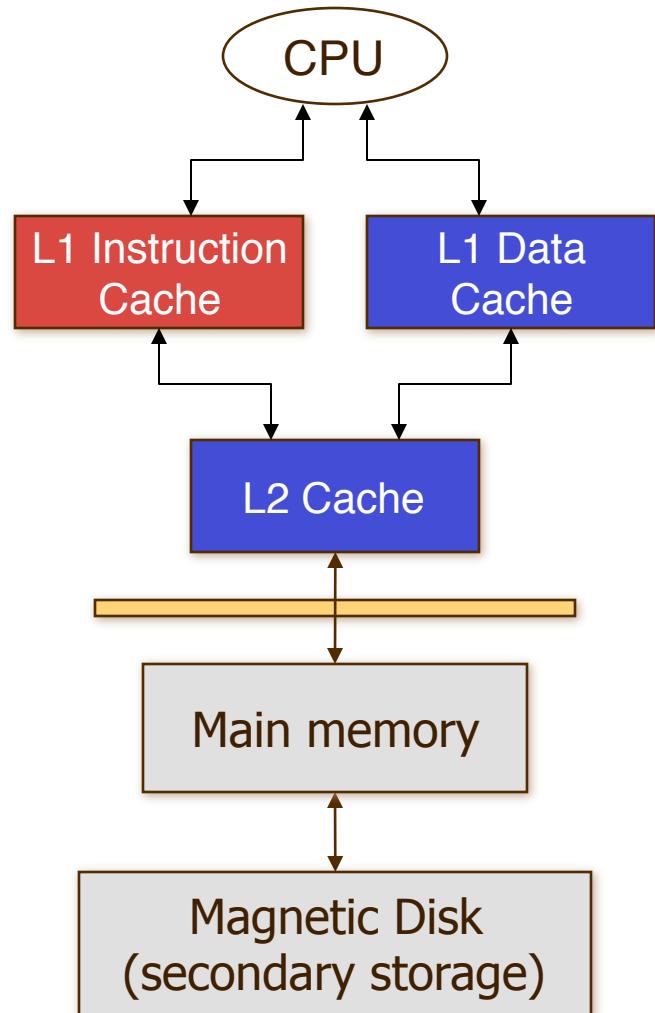
Cache Miss

Caching the Instruction

47

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

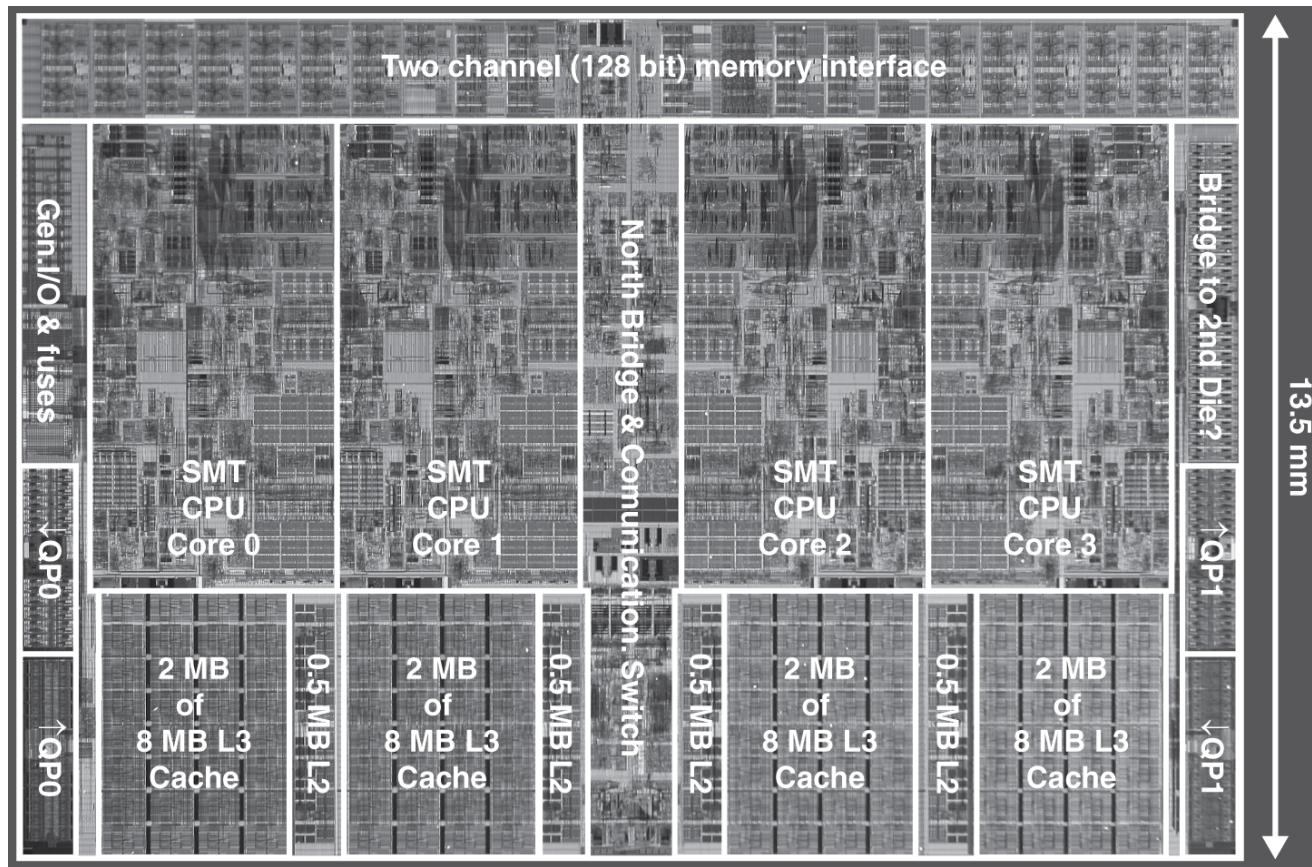


Example: Steps on an Instruction Cache Miss

48

1. Send the PC value to the memory
2. Instruct main memory to perform a read and wait for completion
3. 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
4. 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.

Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

- ❑ 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**)
- ❑ Fully associative: all blocks are candidates
- ❑ Set associative: only blocks within a particular set are candidates

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

Example of LRU Replacement Policy

51

Assume the cache is 4-way set-associative

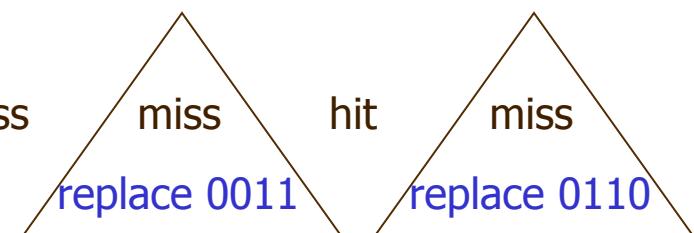
Consider block address stream below (from left to right):

$0011010_2, 0010010_2, 0110010_2, 1000010_2, 0001010_2, 0010010_2, 0100010_2$

- ❑ All these block addresses mapped to same set “010”
- ❑ Question: which blocks remain in the set at the end?

- ❑ Answer: (just look at the tag)

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



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

Definition

- ❑ If the content of a cache entry is modified, it is considered as dirty

If a line is dirty, what should we do upon block replacement?

- ❑ It is a question about how we update the memory

Two options:

1. Write-back
2. Write-through

Upon write access by CPU,

- ❑ 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:

- ❑ CPU 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

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

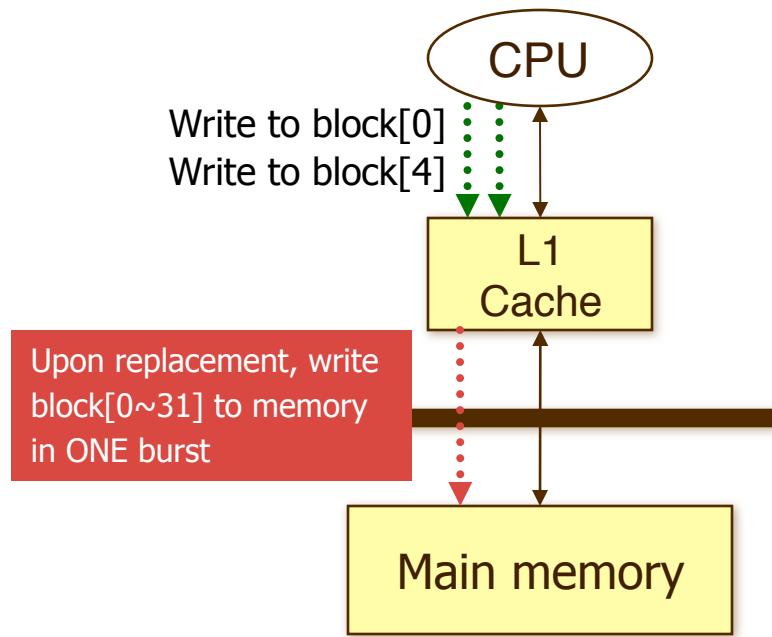
Disadvantages:

- ❑ 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

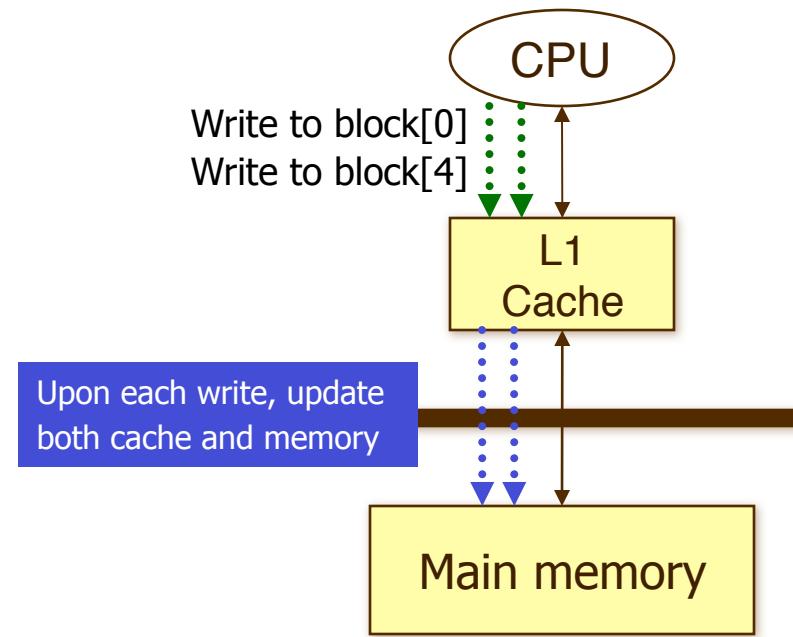
Notes: A **write buffer** is needed for a high-speed system if write-through strategy is used
A **write buffer** is a queue to hold data while data are waiting to be written to memory

Write-Back vs Write-Through

56



Write-Back



Write-Through

2. Virtual Memory

Motivations:

- ❑ 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 \Rightarrow automatically map in hardware)

The processor generates **virtual addresses**

- ❑ While the memory is accessed using **physical addresses**

That means, programs see the **virtual address space**

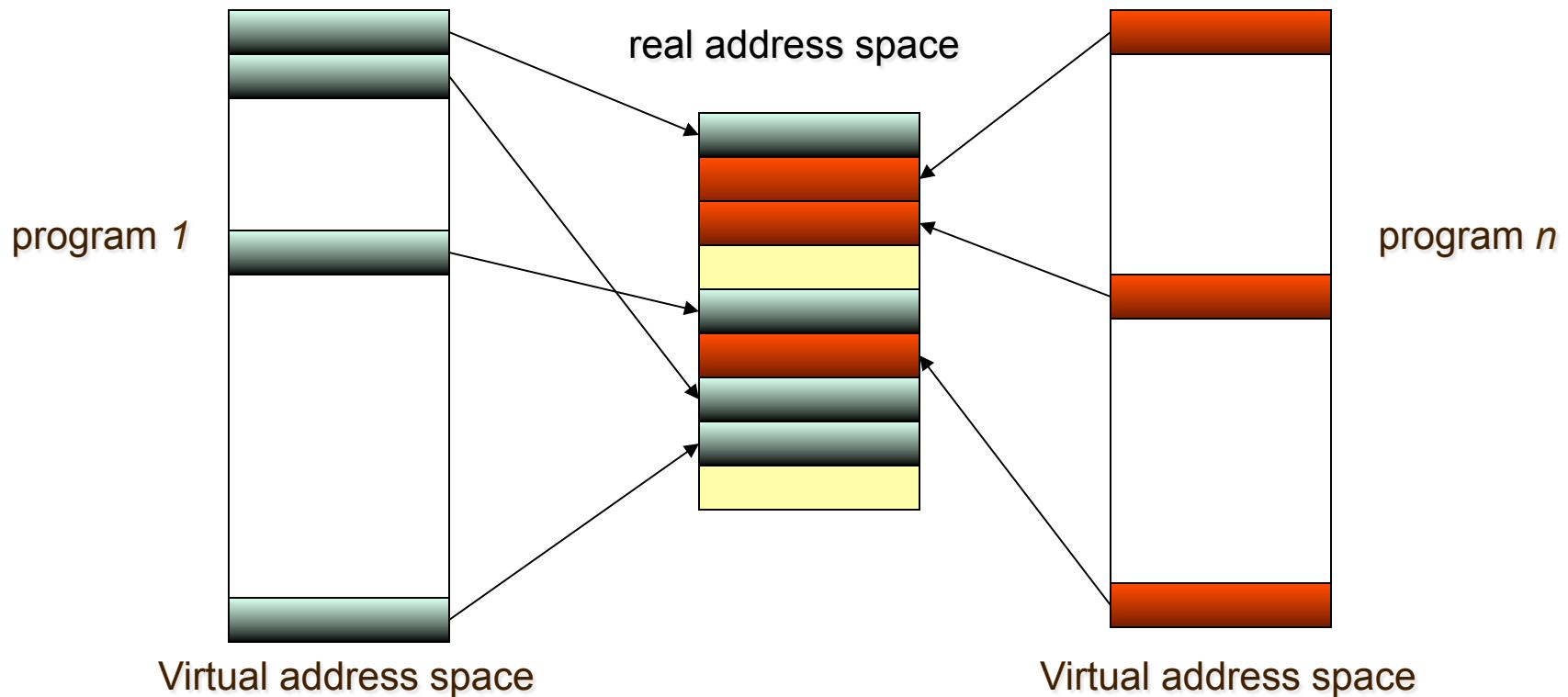
- ❑ While the system sees the **physical address space**
- ❑ Virtual address space is as big as a register can address

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

10.2 Virtual Memory: Basic Concepts

60



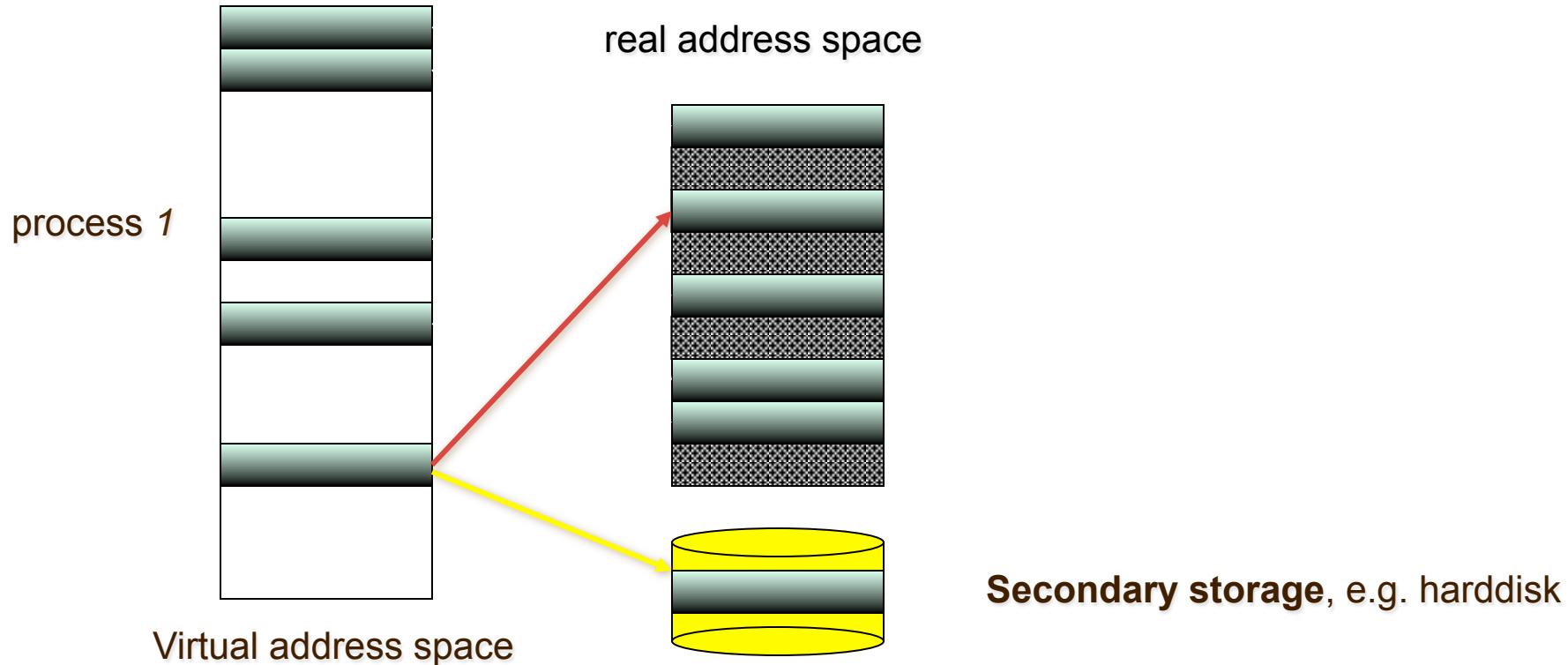
- The “*mapping function*” is implemented as a *table*

10.2 Virtual Memory: Basic Concepts

61

More precisely

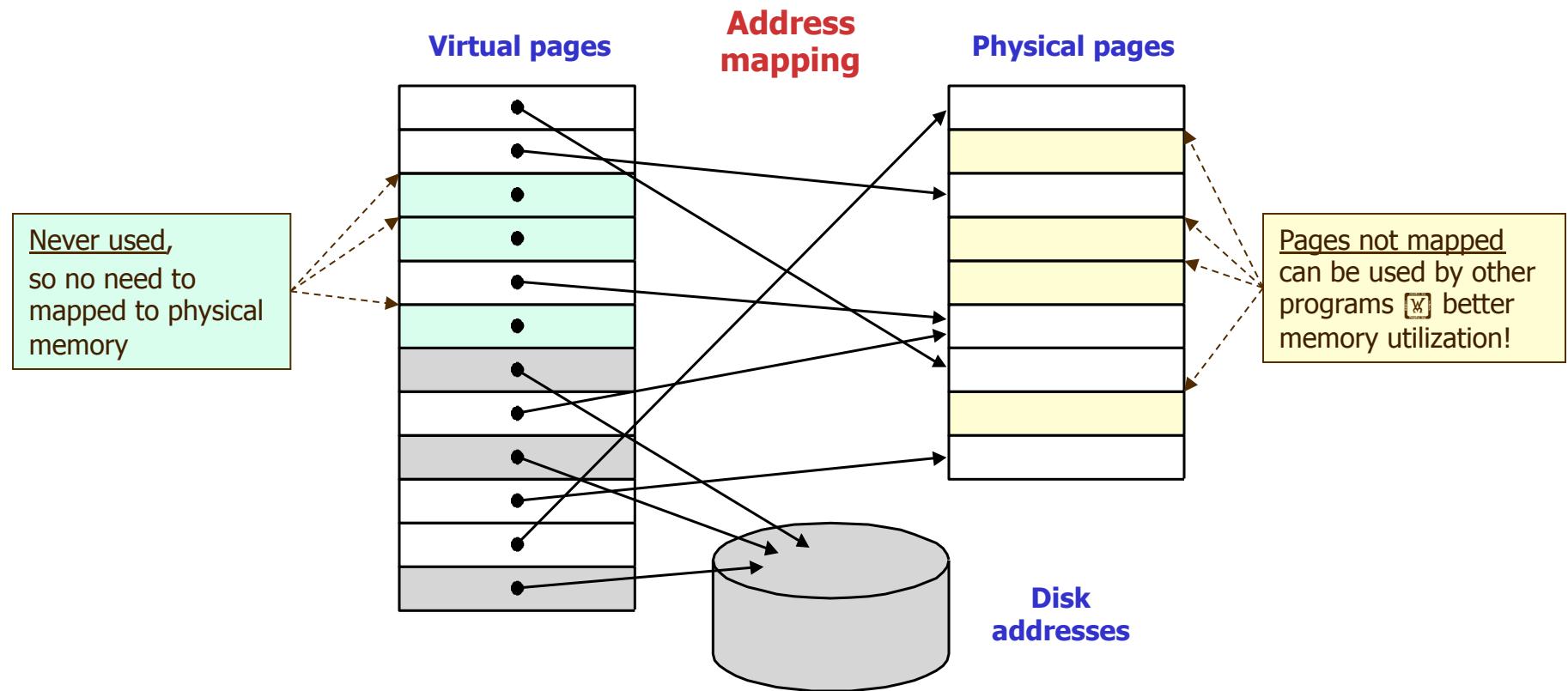
- ❑ Portion of V, when not currently in use, is stored in **secondary storage**
- ❑ When it is requested later, OS shuttles it into the memory, **replacing** other portions not currently in use (replacement policy answers this part)



How Does Virtual Memory Work?

62

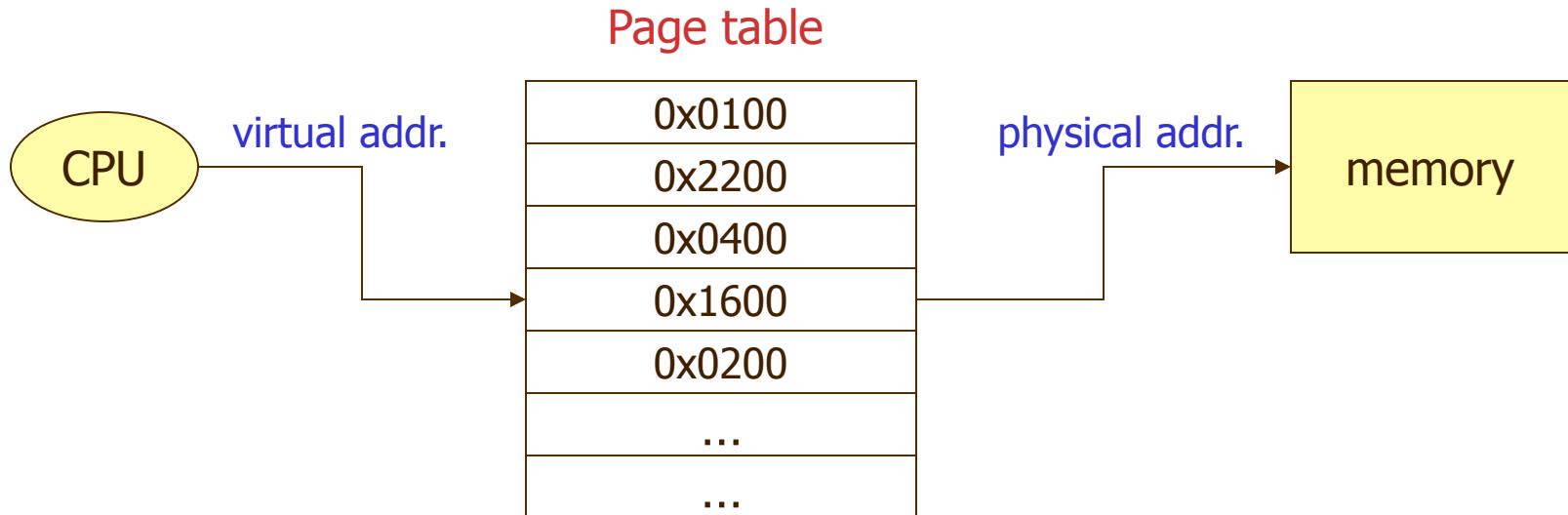
- Virtual page size = physical page size



Implementation of mapping function: **page table**

- Page table maps virtual pages to physical pages

Example

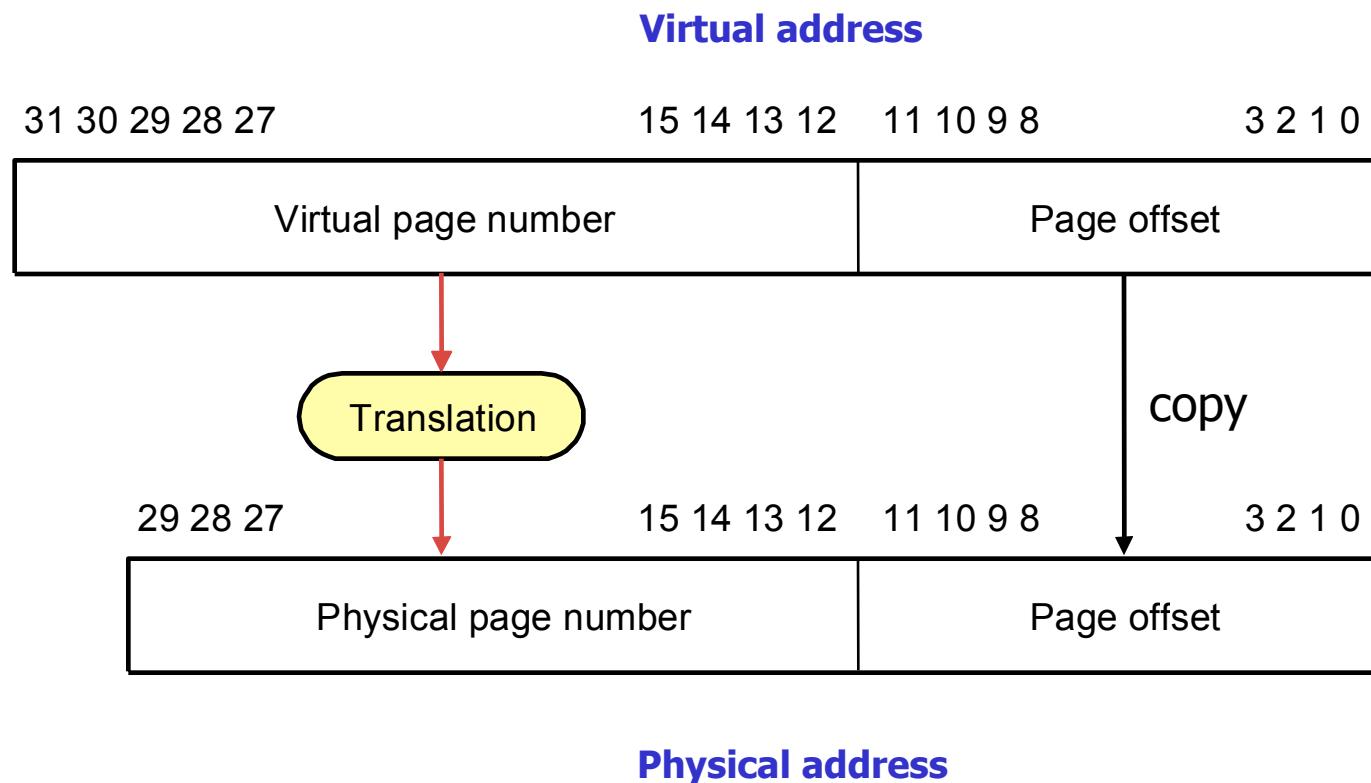


Question: How to convert a virtual address into a virtual page number?

Address Translation Example

64

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



- ❑ 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**

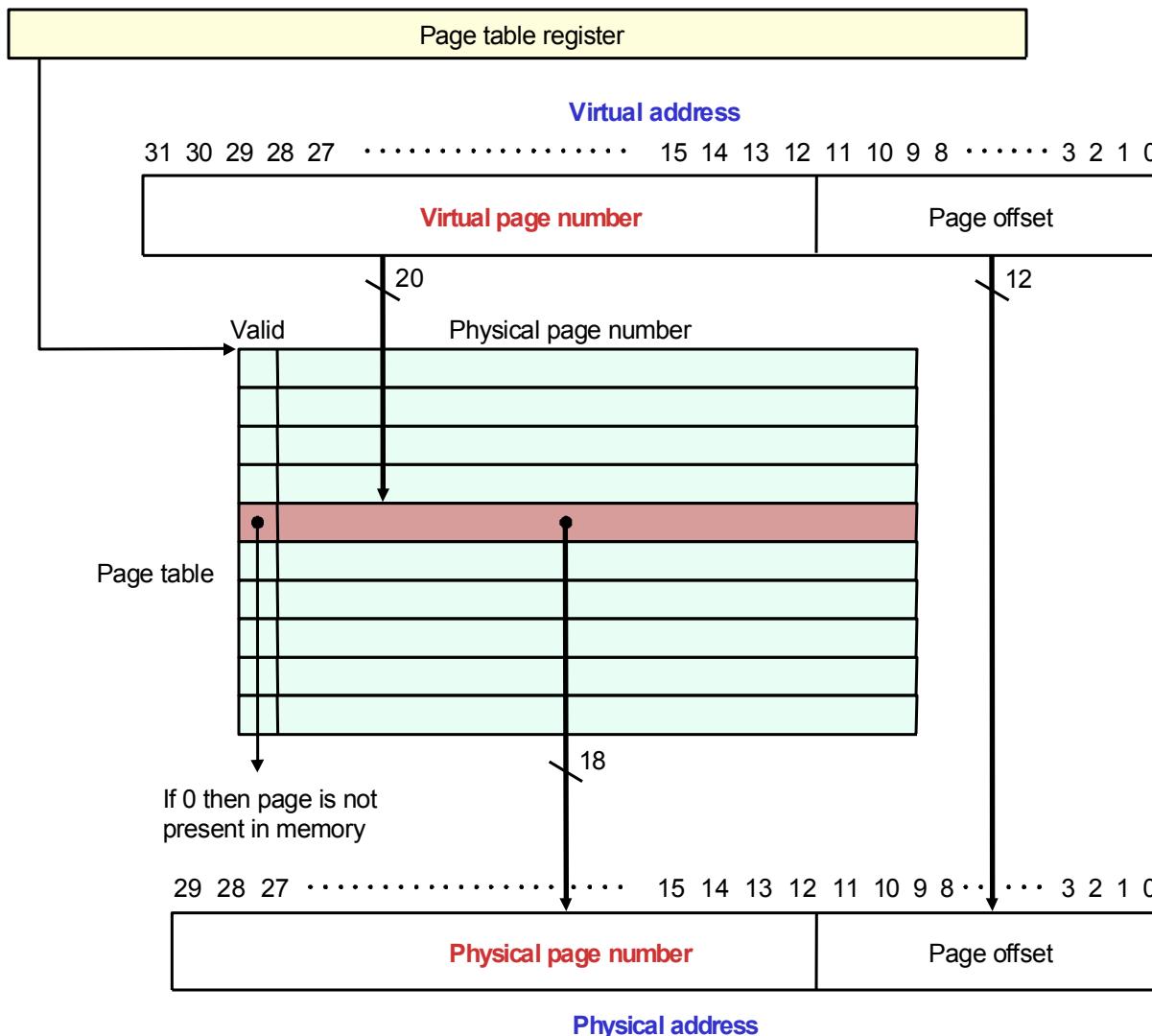
- ❑ Too small the page size, too often we see page faults
- ❑ Too large the page size, higher chances of page fragmentation

Mapping virtual addresses to physical addresses through a **page table**

- ❑ Page table 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

Page Table: Put All Together (cont'd)

67



Example of Looking Up Page Table

68

- ❑ Page size = 4Kbyte (2^{12})
- ❑ Virtual address space = 2^{32}
- ❑ Physical address space = 2^{28}
- ❑ What are the sizes of the virtual and physical page number?
 - Size of virtual page number = $32 - 12 = 20$
 - Size of physical page number = $28 - 12 = 16$
- ❑ What is the physical address for 0xFFFF21340 using page table below?
 - $0xFFFF21340 = \underbrace{1111\ 1111\ 1111}_{\text{Virtual page number}}\ 0010\ 0001\ 0011\ 0100\ 0000_2$
 - Virtual page number = 0xFFFF21
 - Physical address = 0x0AC0340

...
0x0001
0x0AC0
0x0AB0
0x0200
...

Page table

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)

Upon each memory access

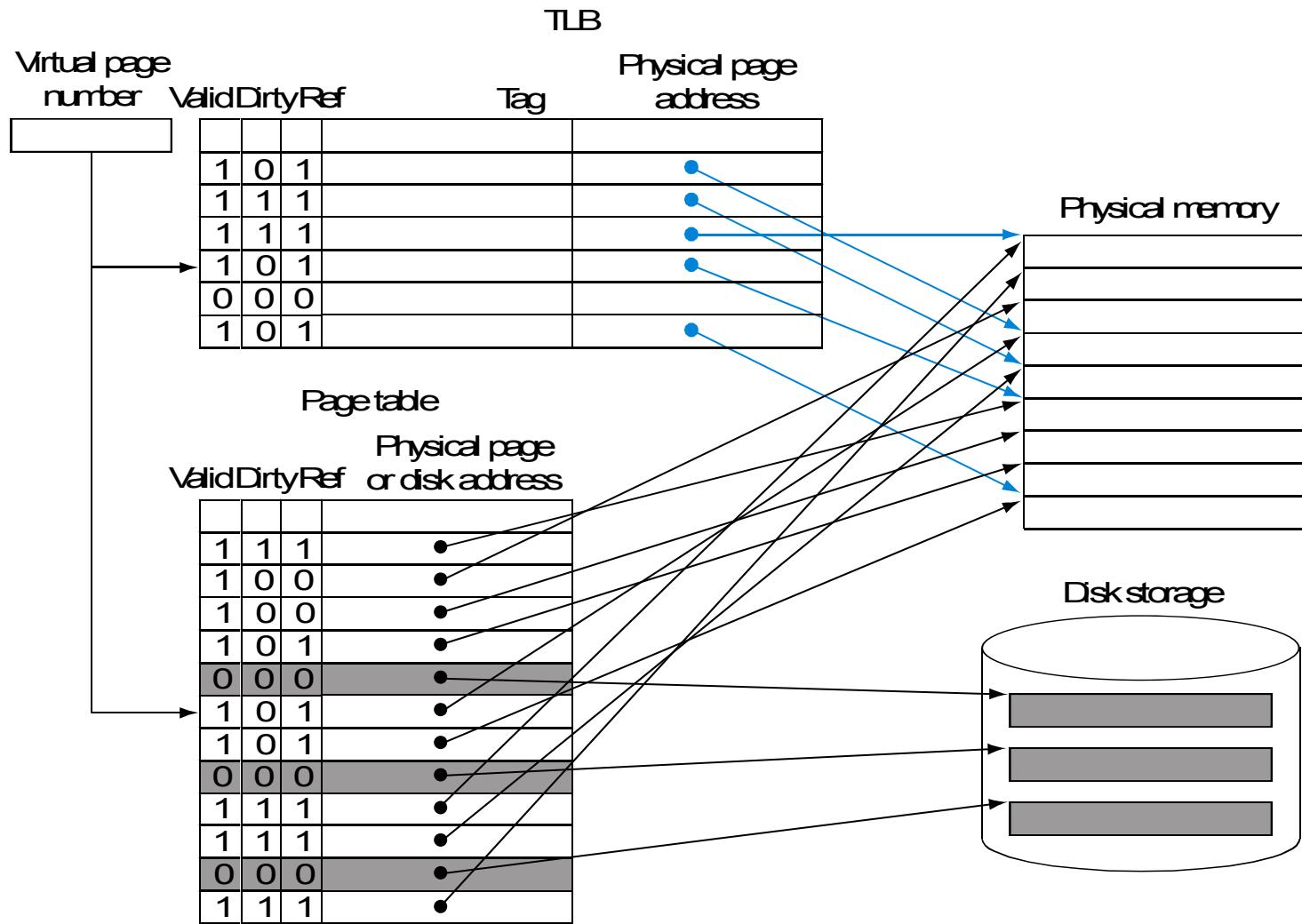
- ❑ Mapping for virtual address generated by CPU is first looked up in TLB
- ❑ If found, do the translation and done
- ❑ If not found, then looked up in the page table residing in memory

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

Translation-Lookaside Buffer (TLB)

71



- ❑ 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**
 - Direct-mapped, set-associative, or fully-associative
 - Data are transferred in **blocks** from main memory to cache upon **misses**
 - 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