# Computer Architecture

**Dr. Haroon Mahmood**

**Assistant Professor**

**NUCES Lahore**

## Memory

- Memory unit that communicates directly with CPU is called Main memory
- Stores large number of bits
  - *m x n: m words of n bits each*
  - k = Log2(*m) address input signals*
  - or *m = 2^k words*
  - e.g., 4,096 x 8 memory:
- 32,768 bits
- 12 address input signals
- 8 input/output data signals

## Write ability/ storage permanence

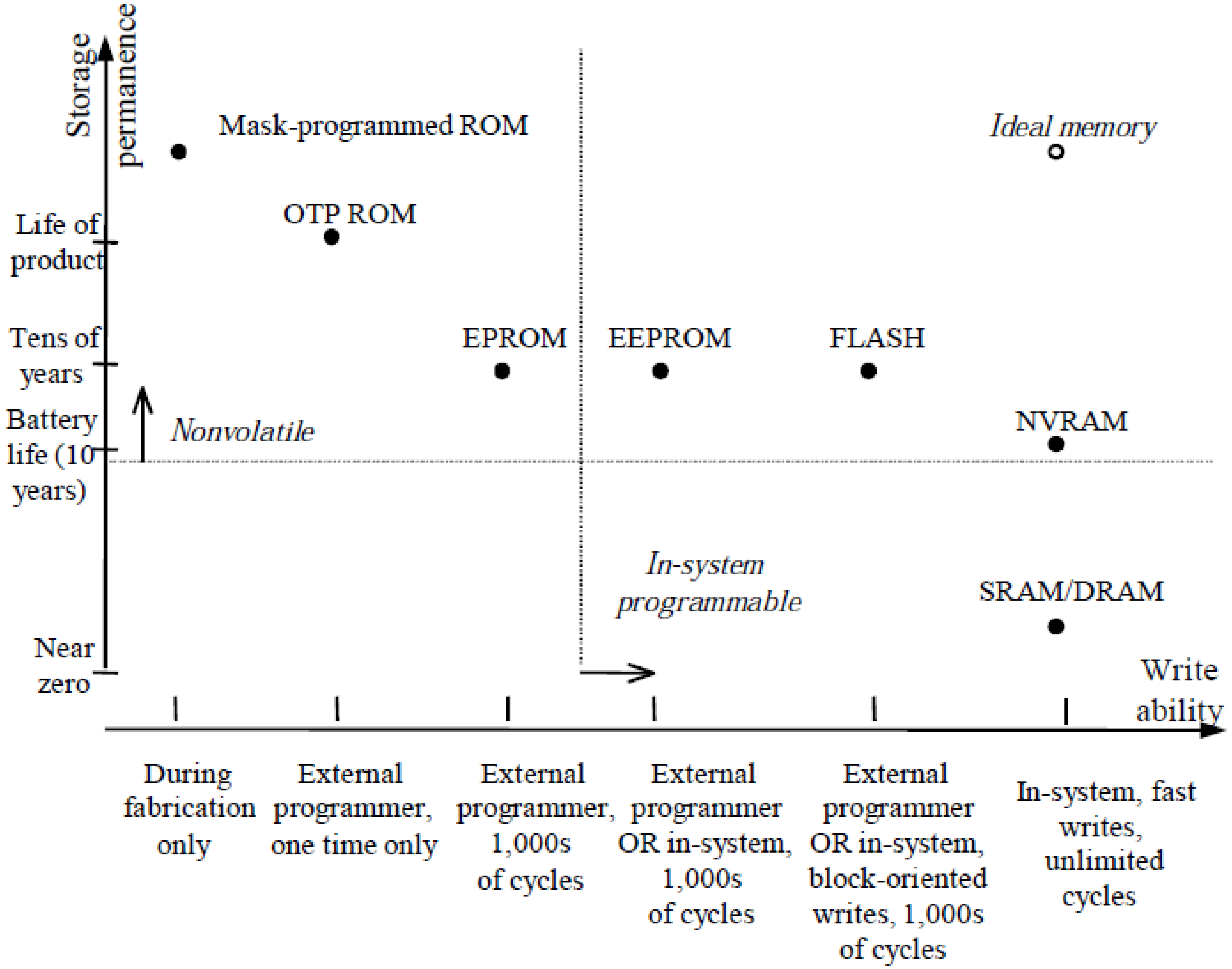
- ROM: read only, bits stored without power
- RAM: read and write, lose stored bits without power

- Traditional distinctions blurred
- Advanced ROMs can be written to e.g., EEPROM
- Advanced RAMs can hold bits without power e.g., NVRAM

- Write ability
  - Manner and speed a memory can be written

- Storage permanence
  - ability of memory to hold stored bits after they are written

# Write ability/ storage permanence



## Ranges of write ability

- High end
    - processor writes to memory simply and quickly e.g., RAM
- Middle range
    - processor writes to memory, but slower  e.g., FLASH, EEPROM
- Lower range
    - special equipment, “programmer”, must be used to write to memory  e.g., EPROM, OTP ROM
- Low end
    - bits stored only during fabrication e.g., Mask-programmed ROM

## Range of storage permanence

- High end
  - essentially never loses bits e.g., mask-programmed ROM
- Middle range
  - holds bits days, months, or years after memory’s power source turned off e.g., NVRAM
- Lower range
  - holds bits as long as power supplied to memory e.g., SRAM
- Low end
  - begins to lose bits almost immediately after written e.g., DRAM
- Nonvolatile memory
  - Holds bits after power is no longer supplied
  - High end and middle range of storage permanence

## ROM: “Read-Only” Memory

- Nonvolatile memory
  - Can be read from but not written to, by a processor
  - Traditionally written to, “programmed”, before inserting to the system
- Uses
  - Store software program for general-purpose processor
- Program instructions can be one or more ROM words
  - Store constant data needed by system
  - Implement combinational circuit

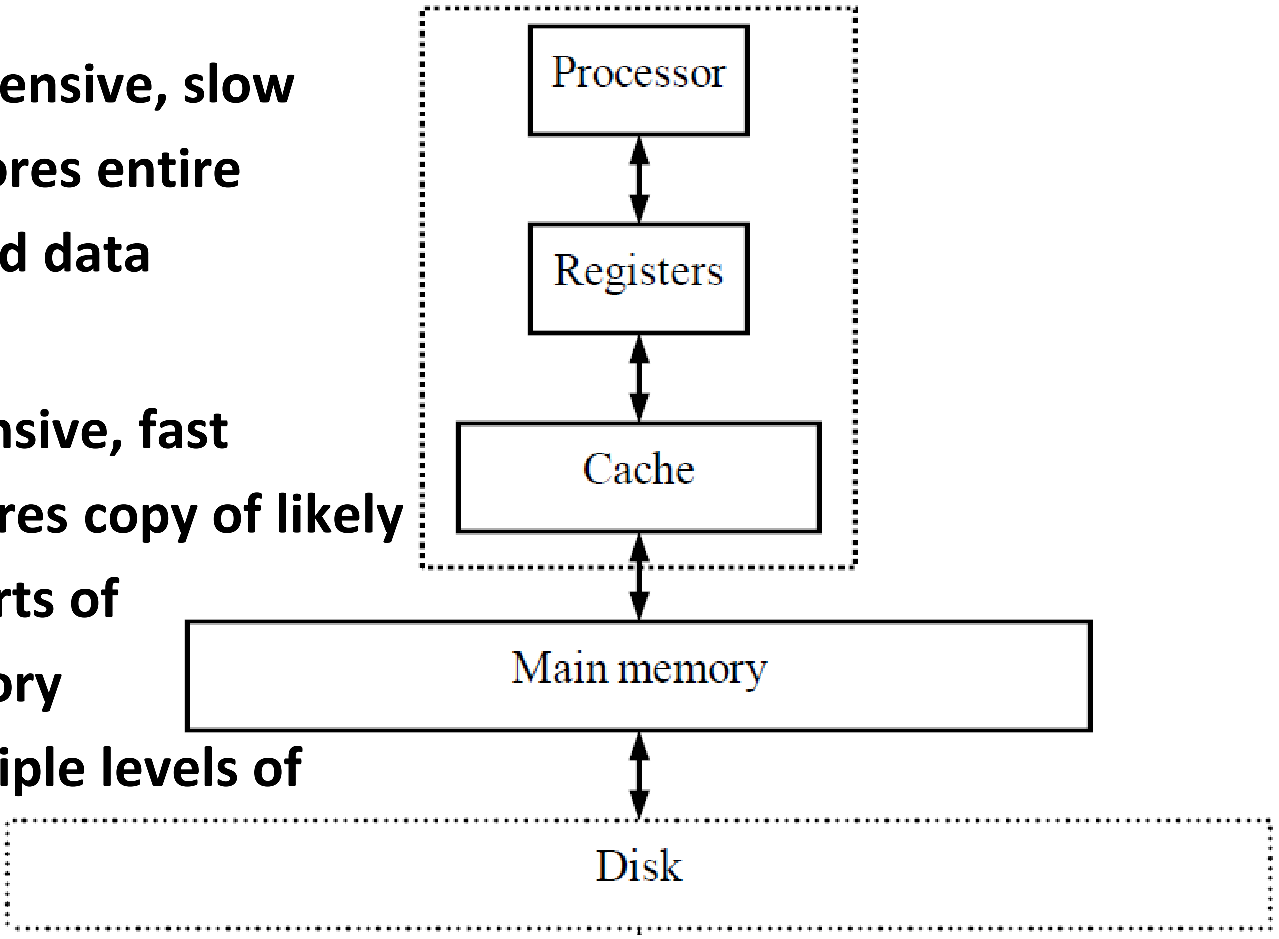
## Basic types of RAM

- **SRAM: Static RAM**
  - Memory cell uses flip-flop to store bit
  - Requires 6 transistors
  - Holds data as long as power supplied

- **DRAM: Dynamic RAM**
  - Memory cell uses MOS transistor and capacitor to store bit
  - More compact than SRAM
  - “Refresh” required due to capacitor leak
  - word’s cells refreshed when read
  - Typical refresh rate 15.625 microsec.
  - Slower to access than SRAM

## Memory Hierarchy

- Want inexpensive, fast memory
- Main memory
  - Large, inexpensive, slow memory stores entire program and data
- Cache
  - Small, expensive, fast memory stores copy of likely accessed parts of larger memory
  - Can be multiple levels of cache



## Cache

- Usually designed with SRAM
  - faster but more expensive than DRAM
- Usually on same chip as processor
  - space limited, so much smaller than off-chip main memory
  - faster access ( 1 cycle vs. several cycles for main memory)
- Cache operation:
  - Request for main memory access (read or write)
  - First, check cache for copy

## Problem 1

- **Assume a program that has:**
- **cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory)**
- **and Misses Per Kilo Instructions (MPKIs) of 20 (L1), 10 (L2), and 5 (L3).**
- **Find the total memory access time.**
- **Find memory access time without L3.**

## Problem 1

- Assume a program that has:
- cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory)
- and Misses Per Kilo Instructions (MPKIs) of 20 (L1), 10 (L2), and 5 (L3).
- Find the total memory access time.
- Find memory access time without L3.

With L3: 1000 + 10x20 + 30x10 + 300x5 = 3000

Without L3: 1000 + 10x20 + 10x300 = 4200

## Cache

- Cache hit
  - copy is in cache, quick access

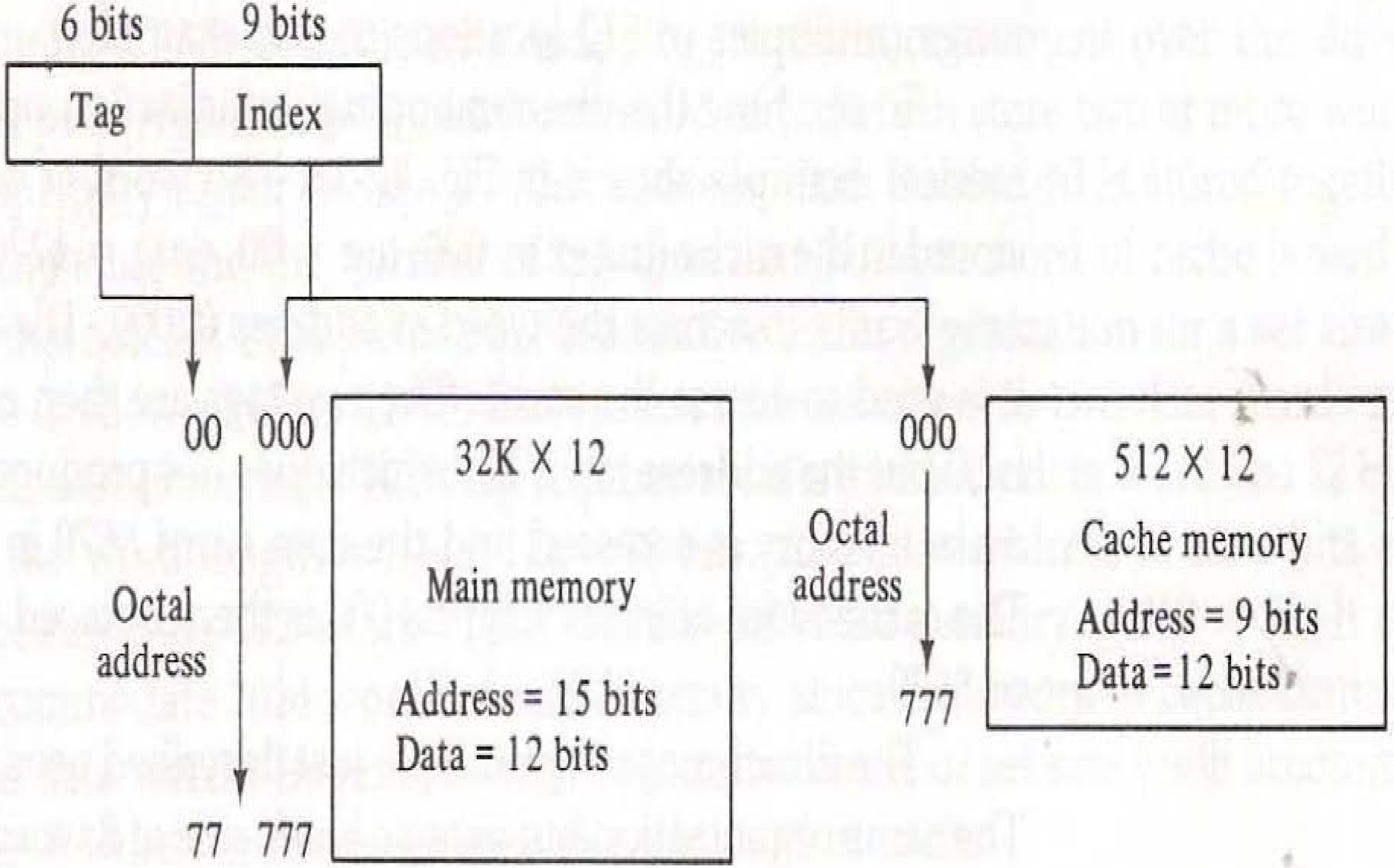
- Cache miss
  - copy not in cache, read address and possibly its neighbors into cache

- Several cache design choices
  - cache mapping
  - replacement policies
  - write techniques

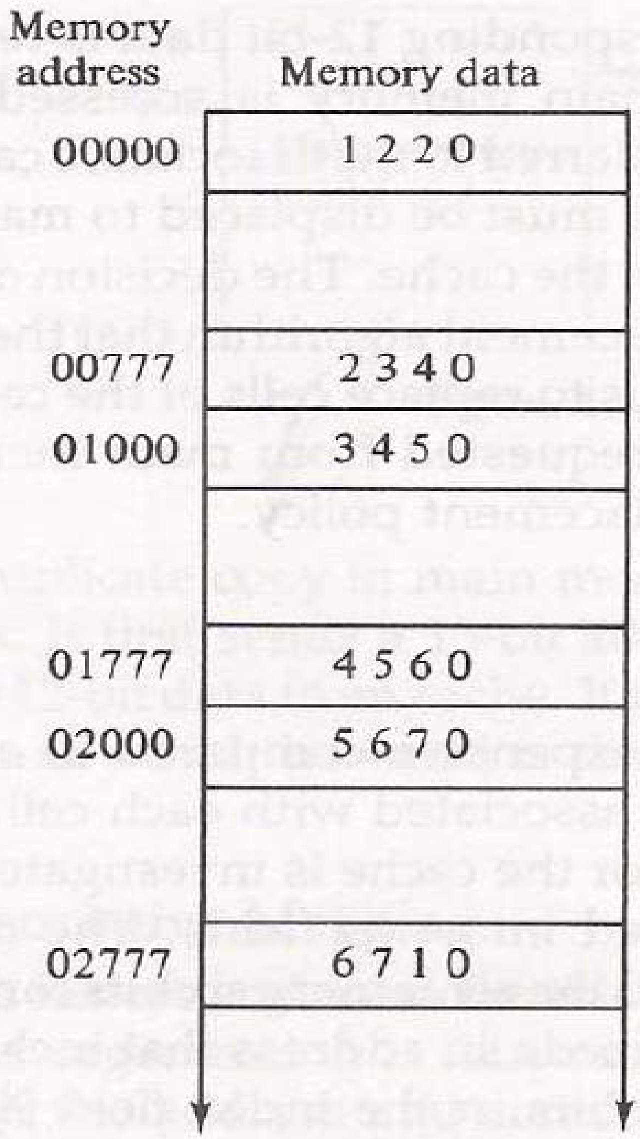
## Direct Mapped Memory

- Main memory address divided into 2 fields
  - Index
    - ✓ cache address
    - ✓ number of bits determined by cache size
  - Tag
    - ✓ compared with tag stored in cache at address indicated by index
    - ✓ if tags match, check valid bit
- Valid bit
  - indicates whether data in slot has been loaded from memory
- Offset
  - used to find particular word in cache line

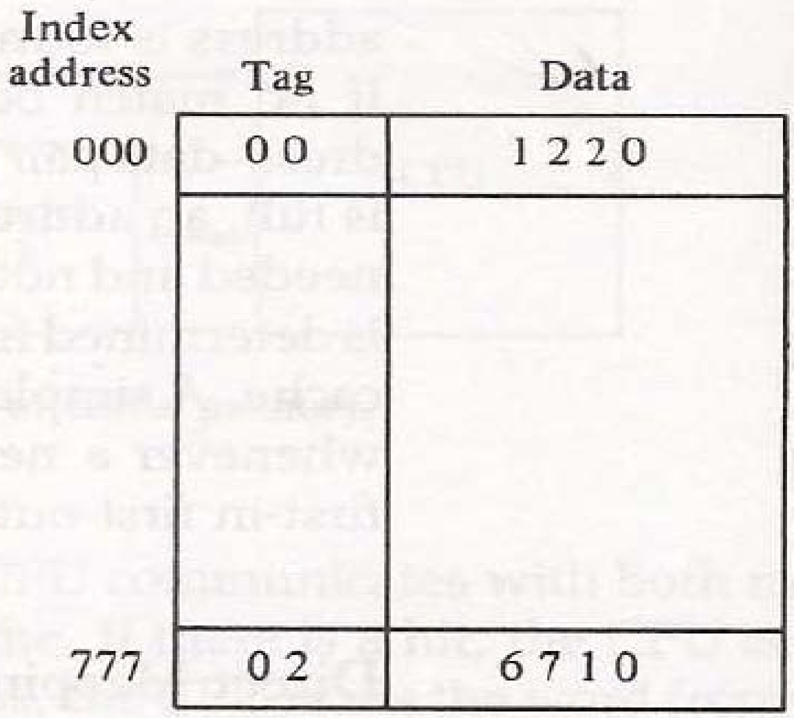
Figure Addressing relationships between main and cache memories.



# Direct Mapping Cache Organization



(a) Main memory



(b) Cache memory

## Associative Mapping

- Fastest and most flexible cache organization uses associative memory.
- It stores both address and content of memory word.
- Address is placed in argument register and memory is searched for matching address.
- All addresses stored in cache simultaneously compared with desired address
- If address is found corresponding data is read.
- If address is not found, it is read from main memory and transferred to cache.

## Set-Associative Mapping

- In direct mapping two words with same index in their address but different tag values can’t reside simultaneously in memory.

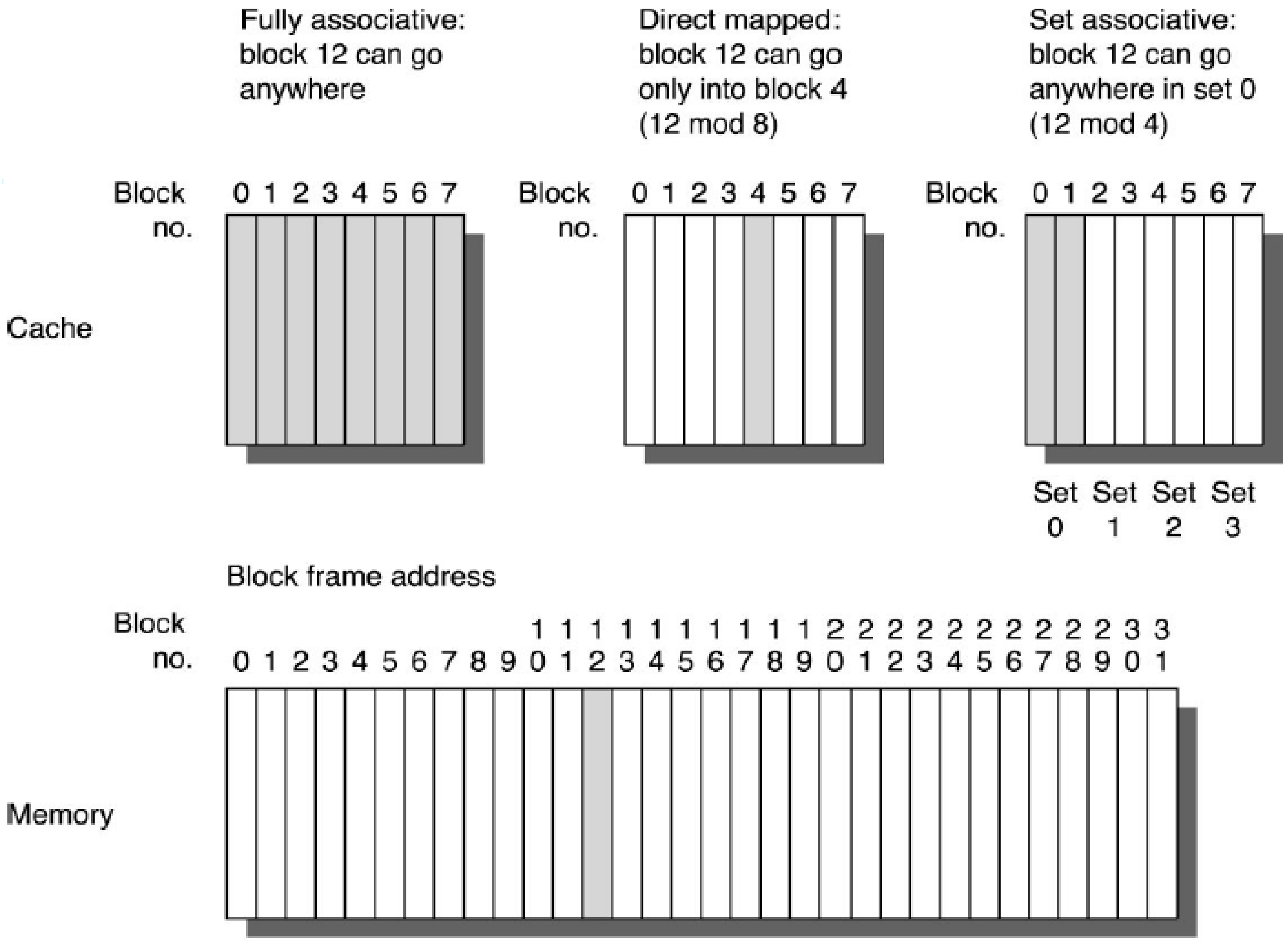
- In this mapping, each data word is stored together with its tag and number of tag-data items in one word of the cache is said to form set.

- In general, a set associative cache of set size k will accommodate k words of main memory in each word of cache.

| Index | Tag | Data | Tag | Data |
|---|---|---|---|---|
| 000 | 0 1 | 3 4 5 0 | 0 2 | 5 6 7 0 |
| | | | | |
| 777 | 0 2 | 6 7 1 0 | 0 0 | 2 3 4 0 |

**Figure** Two-way set-associative mapping cache.

# Block placement in Cache



## Memory Locality

- Memory hierarchies take advantage of *memory locality.*
- *Memory locality is the principle that future memory*
- accesses are *near past accesses.*

- Memories take advantage of two types of locality
  - *Temporal locality -- near in time*

    we will often access the same data again very soon

  - *Spatial locality -- near in space/distance*

    our next access is often very close to our last access (or recentaccesses).

## Types of Cache Misses

- **Compulsory misses:** happens the first time a memory word is accessed – the misses for an infinite cache
- **Capacity misses:** happens because the program touched many other words before re-touching the same word – the misses for a fully-associative cache
- **Conflict misses:** happens because two words map to the same location in the cache – the misses generated while moving from a fully-associative to a direct-mapped cache

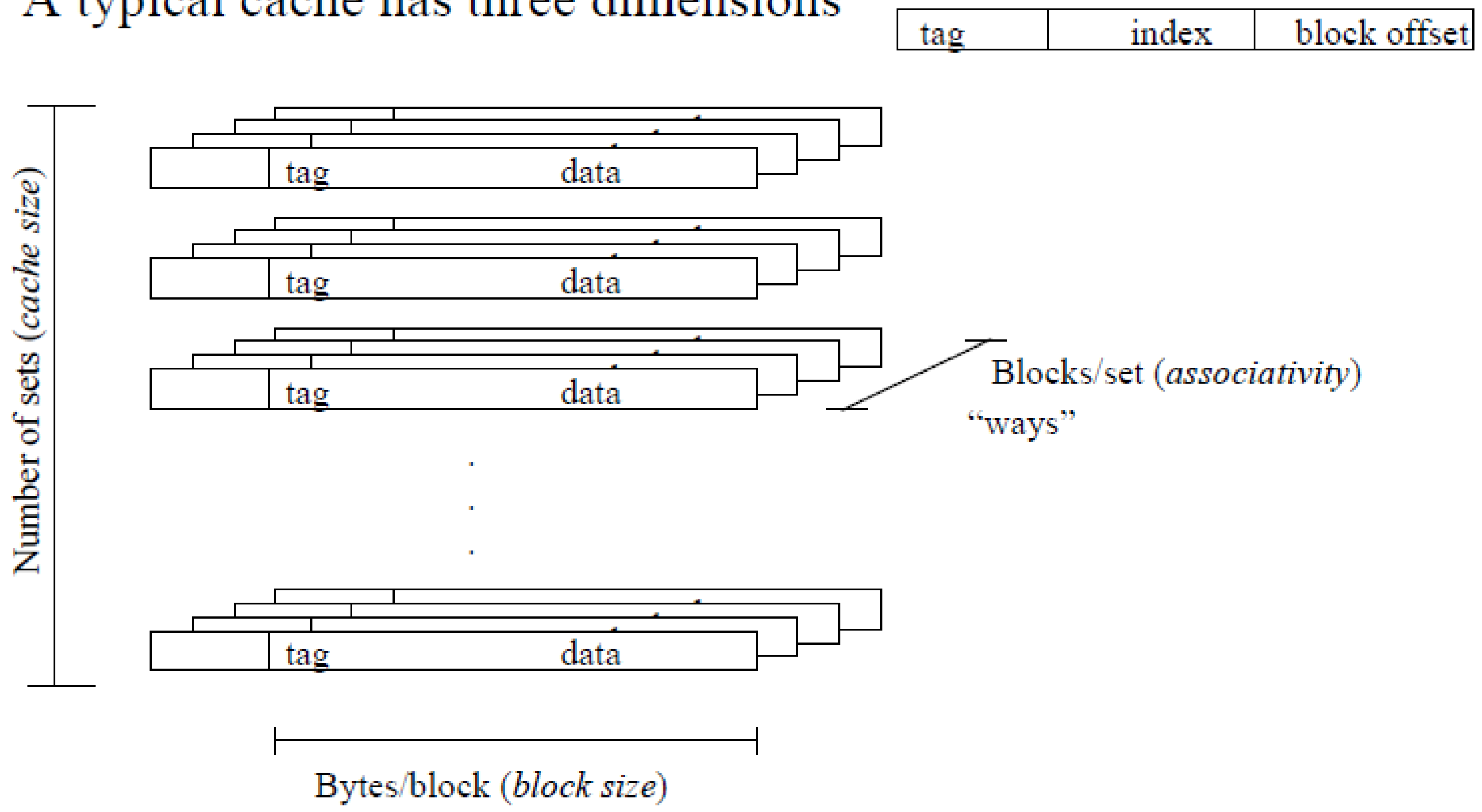
## Cache Replace policies

- Technique for choosing which block to replace
  - when fully associative cache is full
  - when set-associative cache's line is full
- Direct mapped cache has no choice

- Random
  - replace block chosen at random
- LRU: least-recently used
  - replace block not accessed for longest time
- FIFO: first-in-first-out
  - push block onto queue when accessed
  - choose block to replace by popping queue

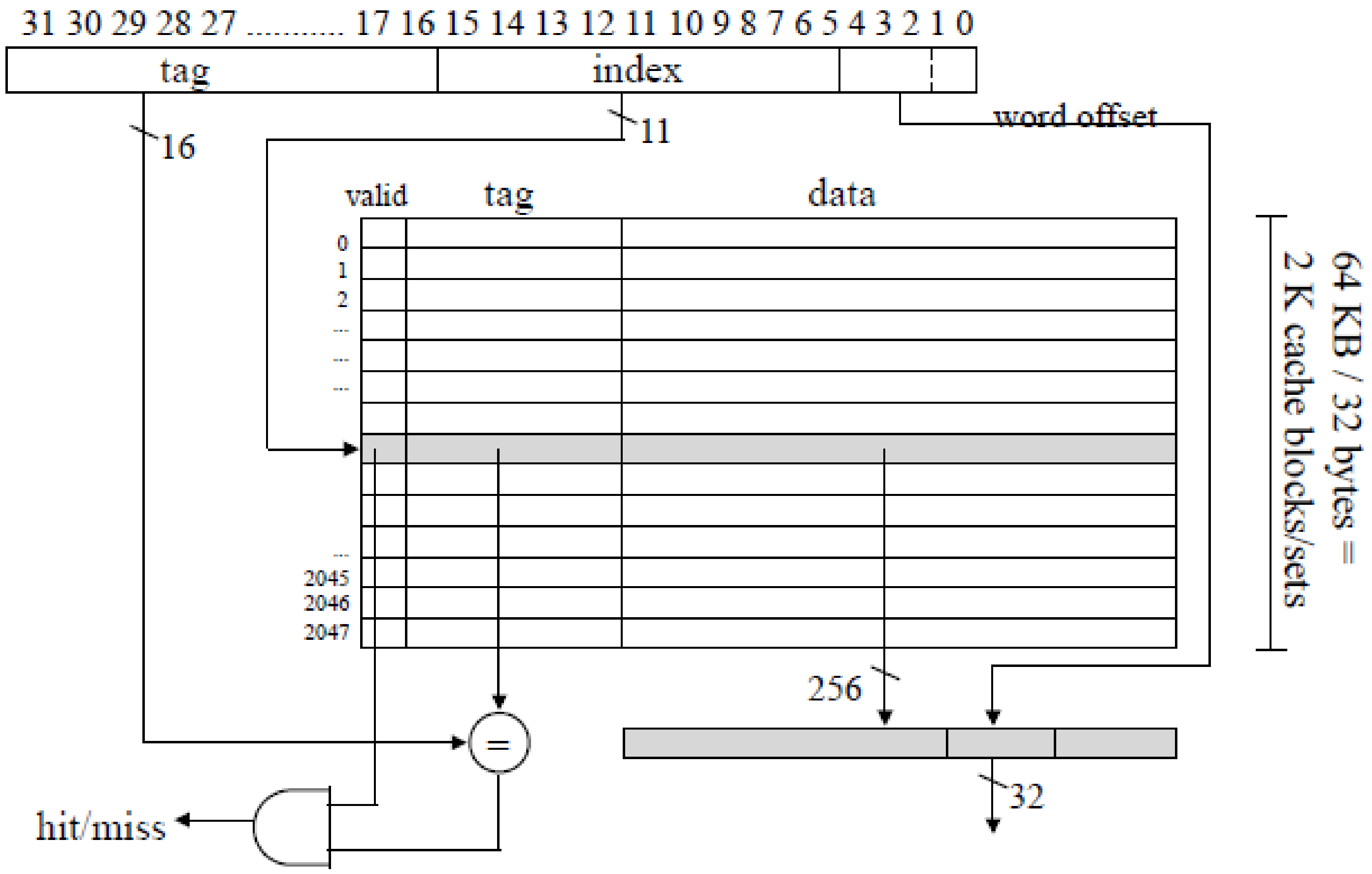
# Cache Organization

A typical cache has three dimensions


tag
index
block offset
tag
data
Number of sets (*cache size*)
Blocks/set (*associativity*)
"ways"
Bytes/block (*block size*)

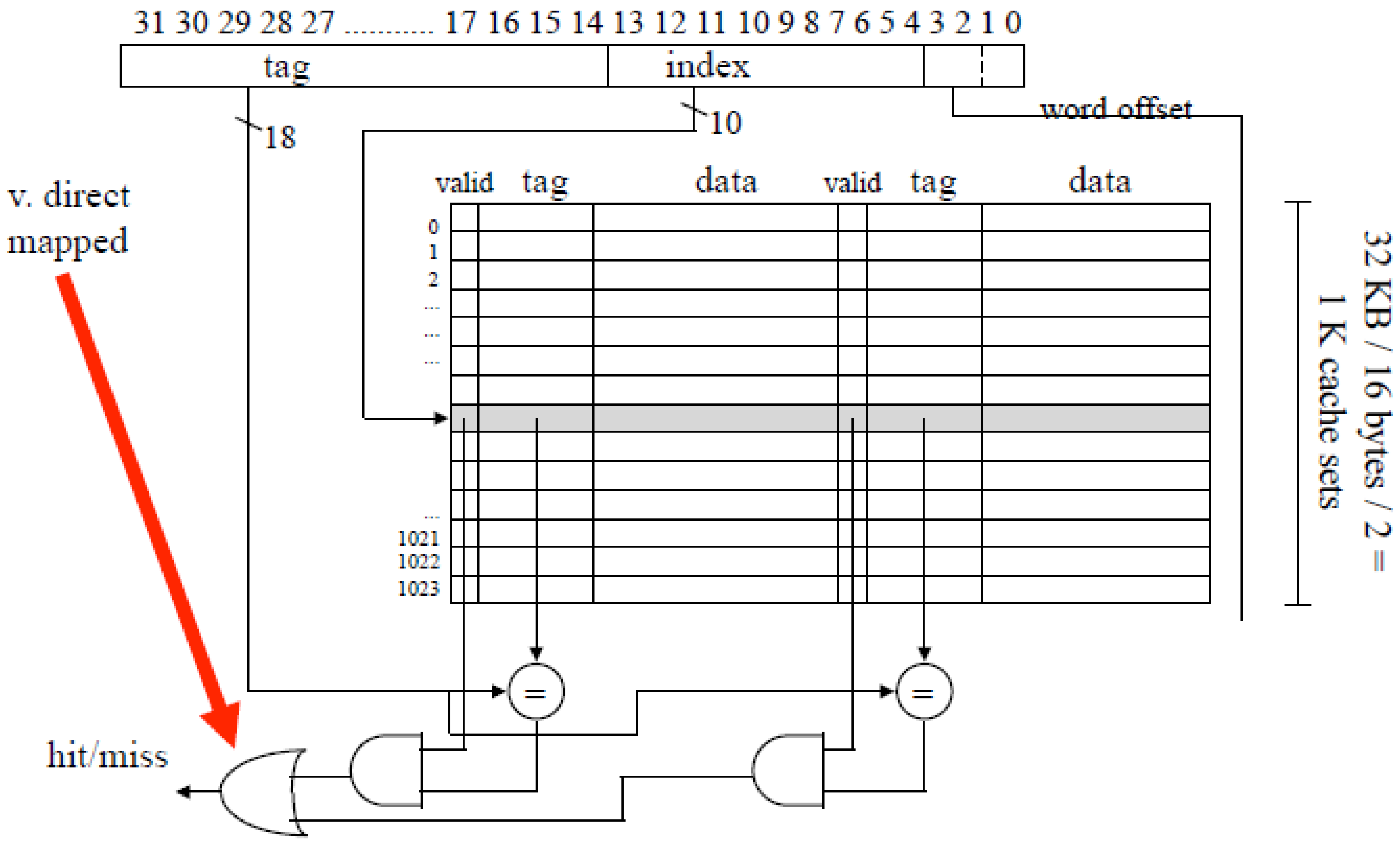
# Accessing direct mapped cache

64 KB cache, direct-mapped, 32-byte cache block size



# 2-way set-associative

32 KB cache, 2-way set-associative, 16-byte block size



## Problem 1

**8 KB fully-associative data cache array with 64 byte line sizes, assume a 40-bit address**

**How many sets? How many ways?**

**How many index bits, offset bits, tag bits?**

## Problem 1

8 KB fully-associative data cache array with 64 byte line sizes, assume a 40-bit address

How many sets (1) ? How many ways (128) ?

How many index bits (0), offset bits (6), tag bits (34) ?

## Problem 2

64 KB 16-way set-associative data cache array with 64 byte line sizes, assume a 40-bit address

How many sets?

How many index bits, offset bits, tag bits?

## Problem 2

64 KB 16-way set-associative data cache array with 64 byte line sizes, assume a 40-bit address

How many sets? 64

How many index bits (6), offset bits (6), tag bits (28)?

## Problem 3

- Assume
  - Cache of 4K blocks
  - 16 bytes block size
  - 32 bit address

- Direct mapped (associativity=1) :
  - 16 bytes per block = 2^4
  - 32 bit address : 32-4=28 bits for index and tag
  - #sets=#blocks/ associativity : log2 of 4K=12 : 12 for index
  - Total number of tag bits : (28-12)*4K=64 Kbits

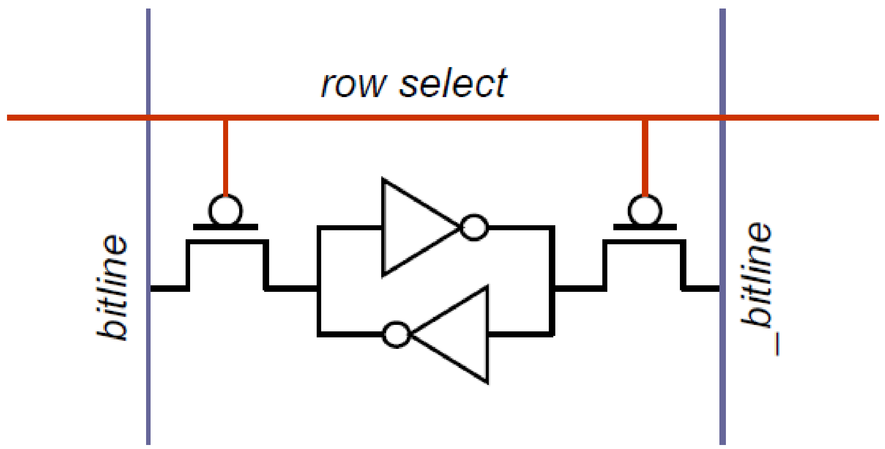
# Problem 3

- 2-way associative
  - #sets=#blocks/associativity : 2K sets
  - 1 bit less for indexing, 1 bit more for tag
  - Tag bits : (28-11) * 2 * 2K=68 Kbits

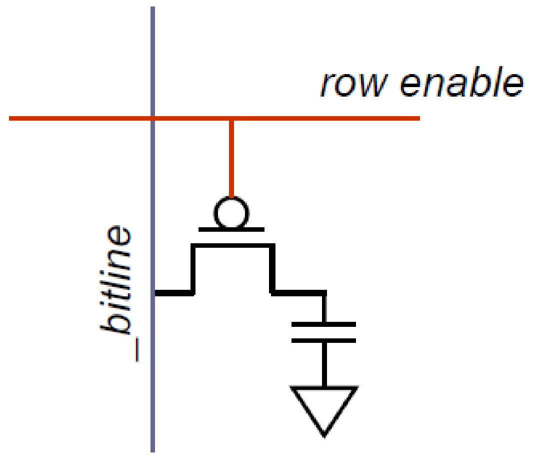
- 4-way associative
  - #sets=#blocks/associativity : 1K sets
  - 1 bit less for indexing, 1 bit more for tag
  - Tag bits : (28-10) * 4 * 1K=72 Kbits

# SRAM vs DRAM

Static Random Access Memory



Dynamic Random Access Memory

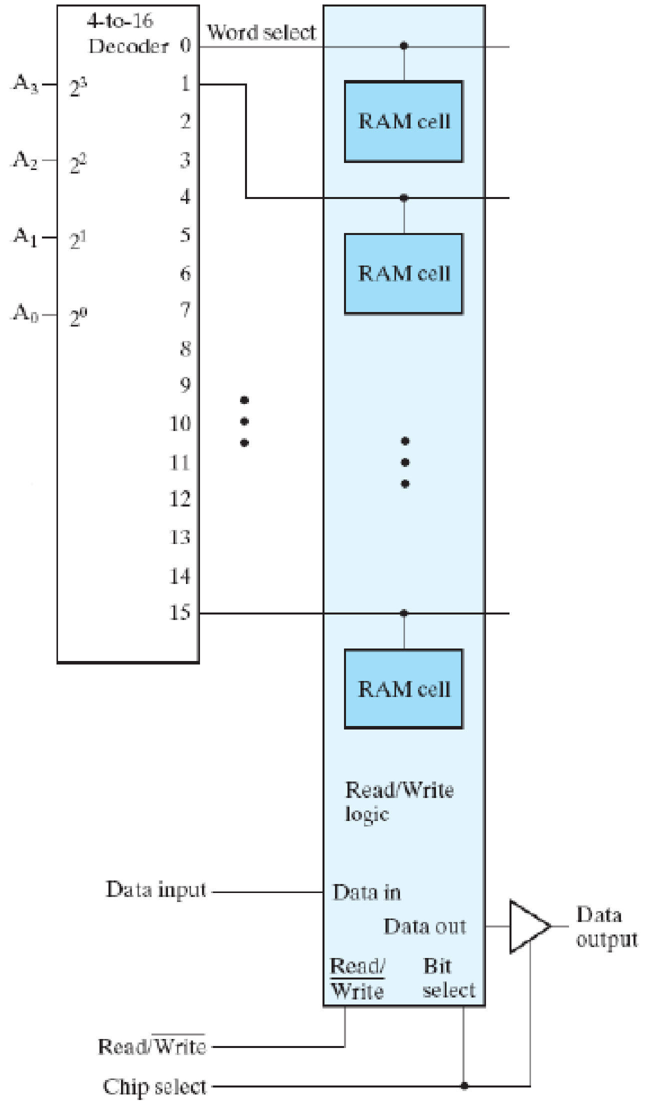


- **Bitlines driven by transistors**
  - **- Fast (10x)**
- **1 transistor and 1 capacitor vs. 6 transistors**
  - **– Large (~6-10x)**

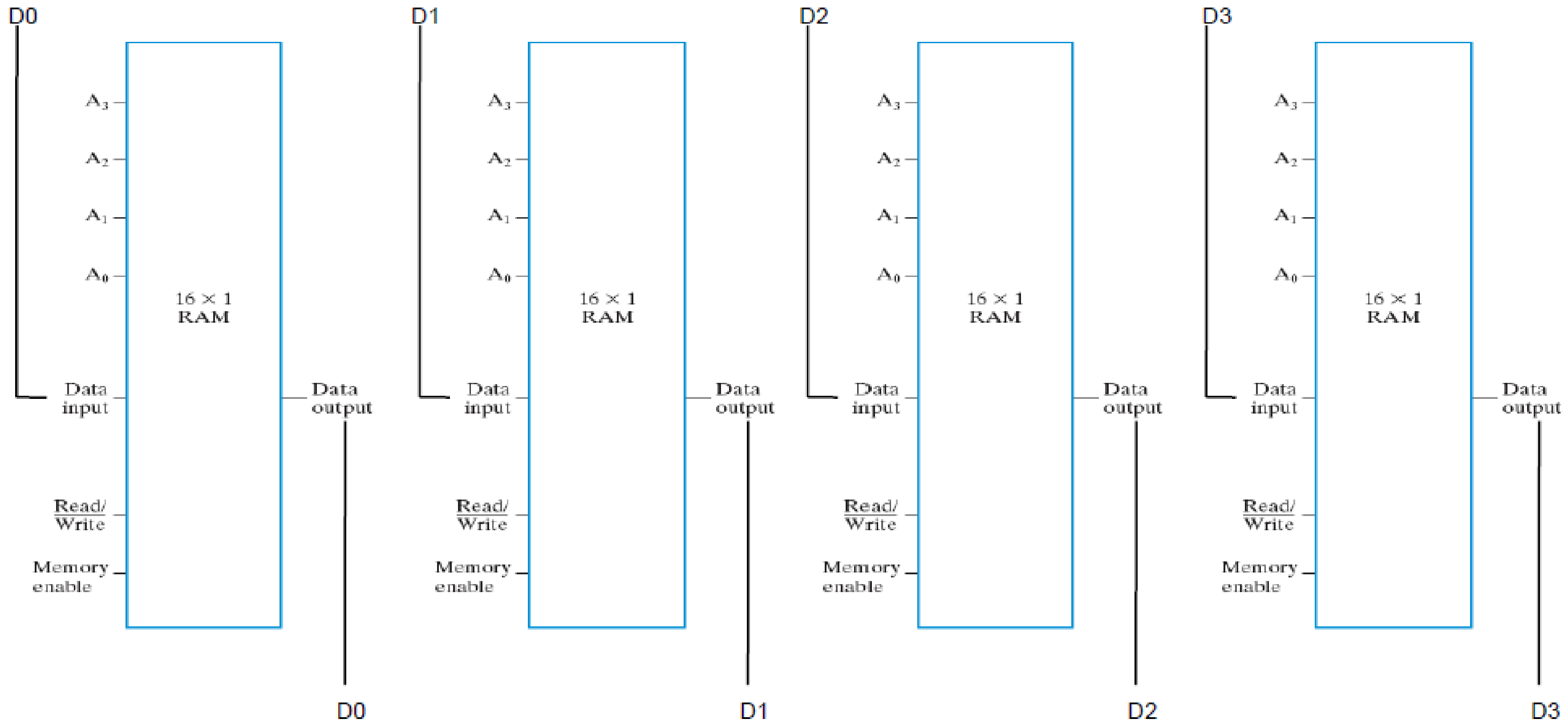
- **A bit is stored as charge on the capacitor**
- **Bit cell loses charge over time (read operation and circuit leakage)**
  - **Must periodically refresh**
  - **Hence the name *Dynamic* RAM**

*Credits: J.Leverich, Stanford*

# 16 x 1 bit RAM

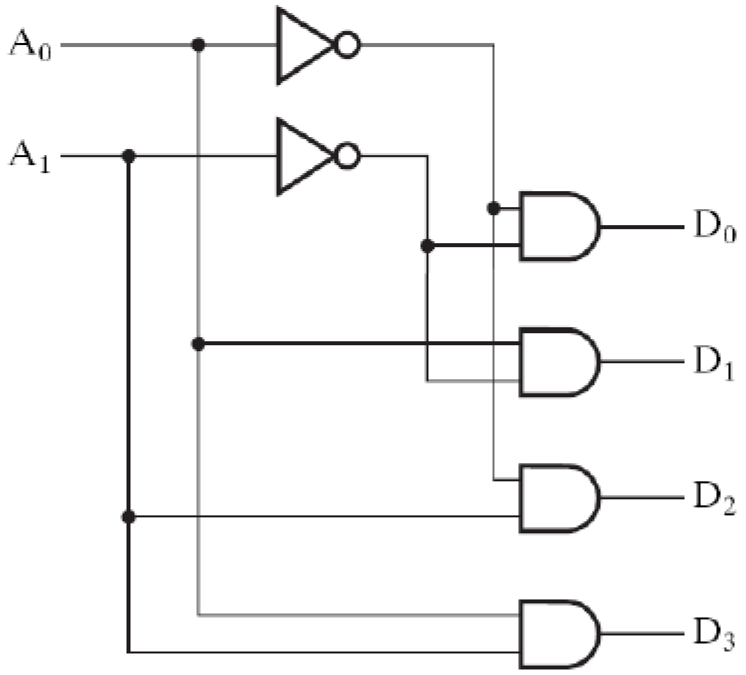

4-to-16
Decoder
Word select
$A_3$
$A_2$
$A_1$
$A_0$
$2^3$
$2^2$
$2^1$
$2^0$
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
RAM cell
RAM cell
RAM cell
Read/Write
logic
Data input
Data in
Data out
Data
output
$\text{Read}/\overline{\text{Write}}$
Bit
select
$\text{Read}/\overline{\text{Write}}$
Chip select

# 16 x 4 bit RAM


D0
D1
D2
D3
$A_3$
$A_2$
$A_1$
$A_0$
16 × 1 RAM
Data input
Data output
Read/$\overline{\text{Write}}$
Memory enable
D0
D1
D2
D3

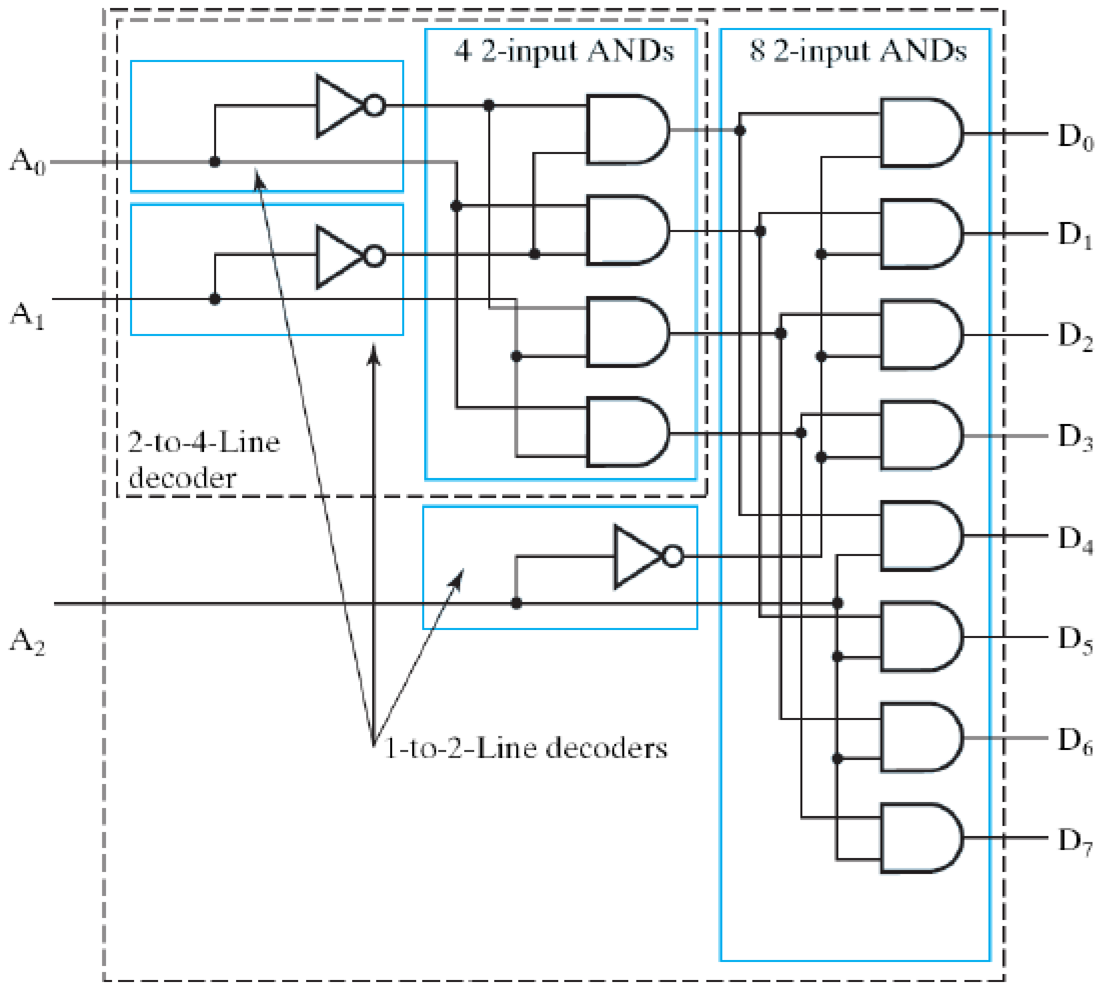
# Decoder

## Decoder 2 to 4



# Decoder

## Decoder 3 to 8

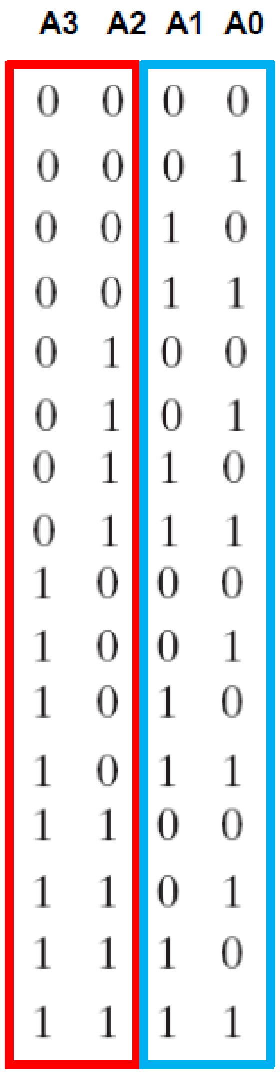


# Increasing number of bits

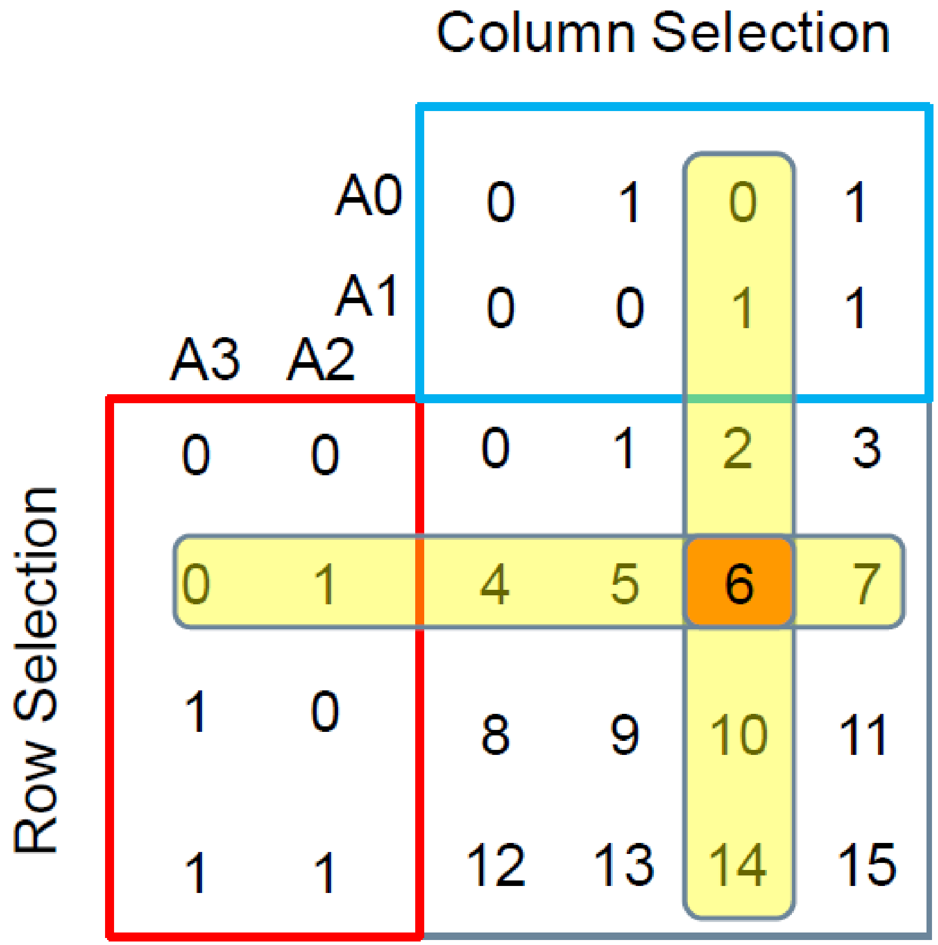
**What to do to reduce the size of decoders?**

- **Solution:** Use a *coincident selection scheme or matrix selection scheme*
- Each address location has two components: a Row Address and a Column Address
- This scheme allows the use of two decoders (Row and Column) instead of one, reducing in this way the total number of gates needed to select the memory locations. Each decoder handles half of the address signals

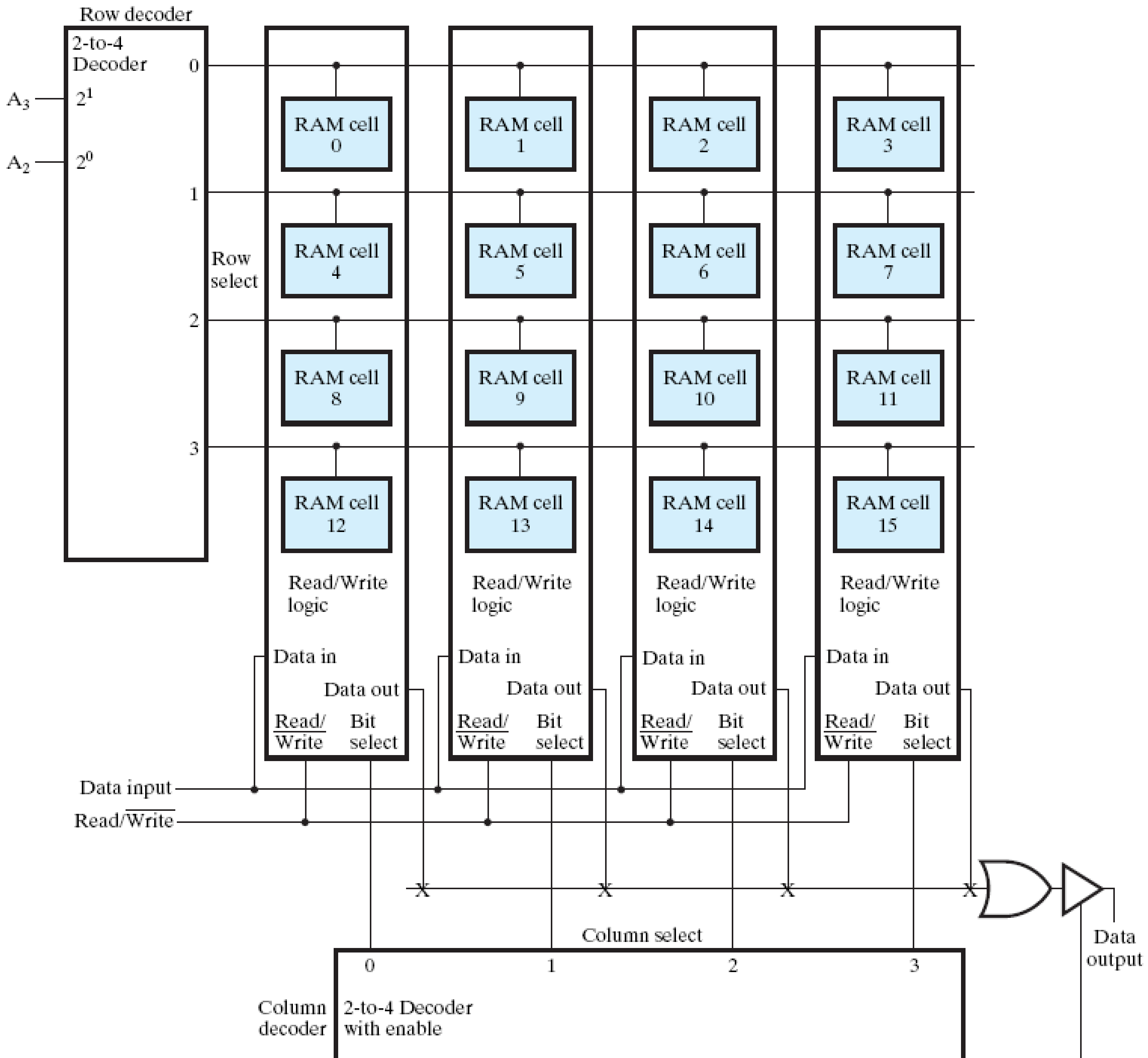
# Managing increasing number of bits



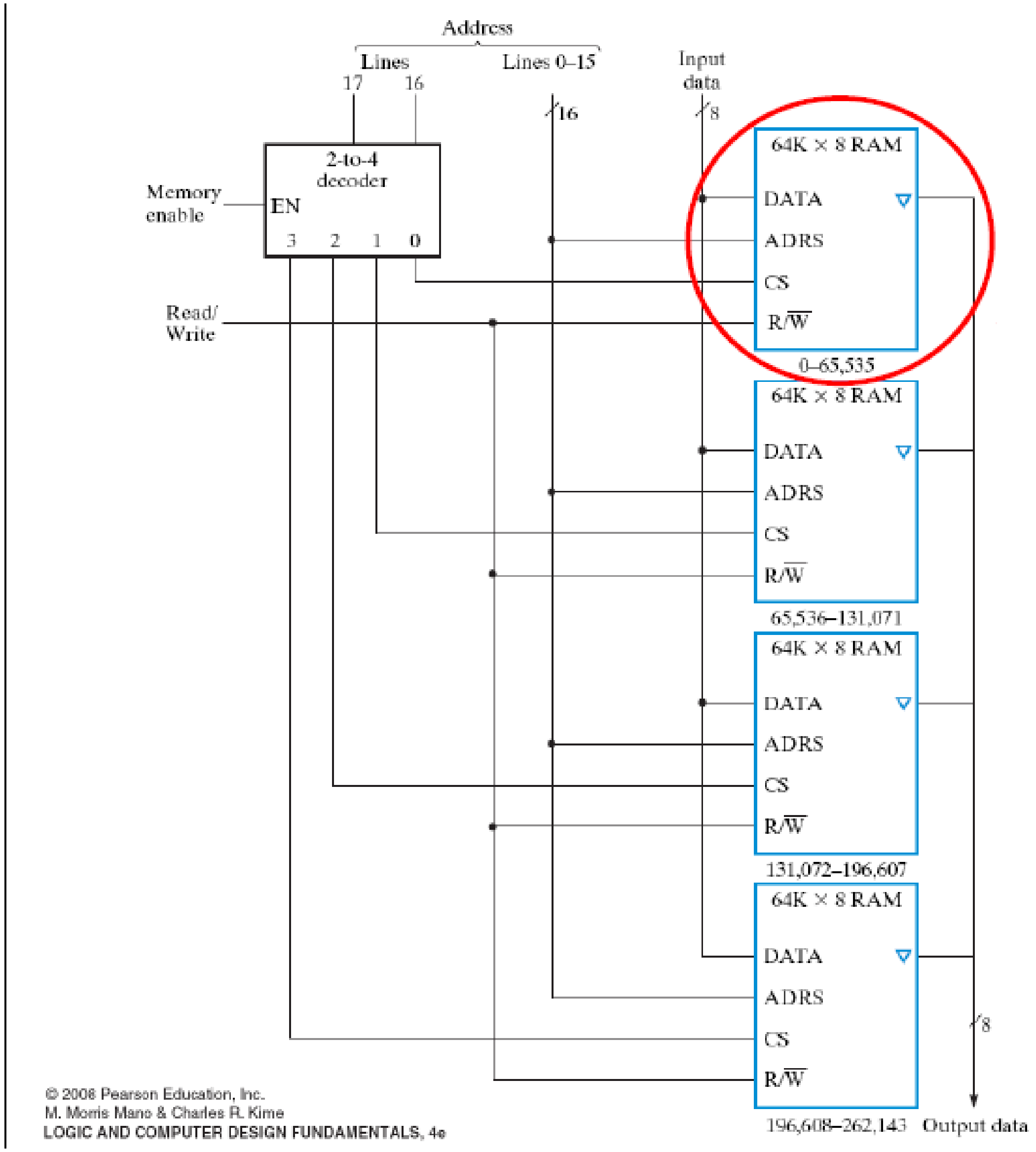
| A3 | A2 | A1 | A0 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |



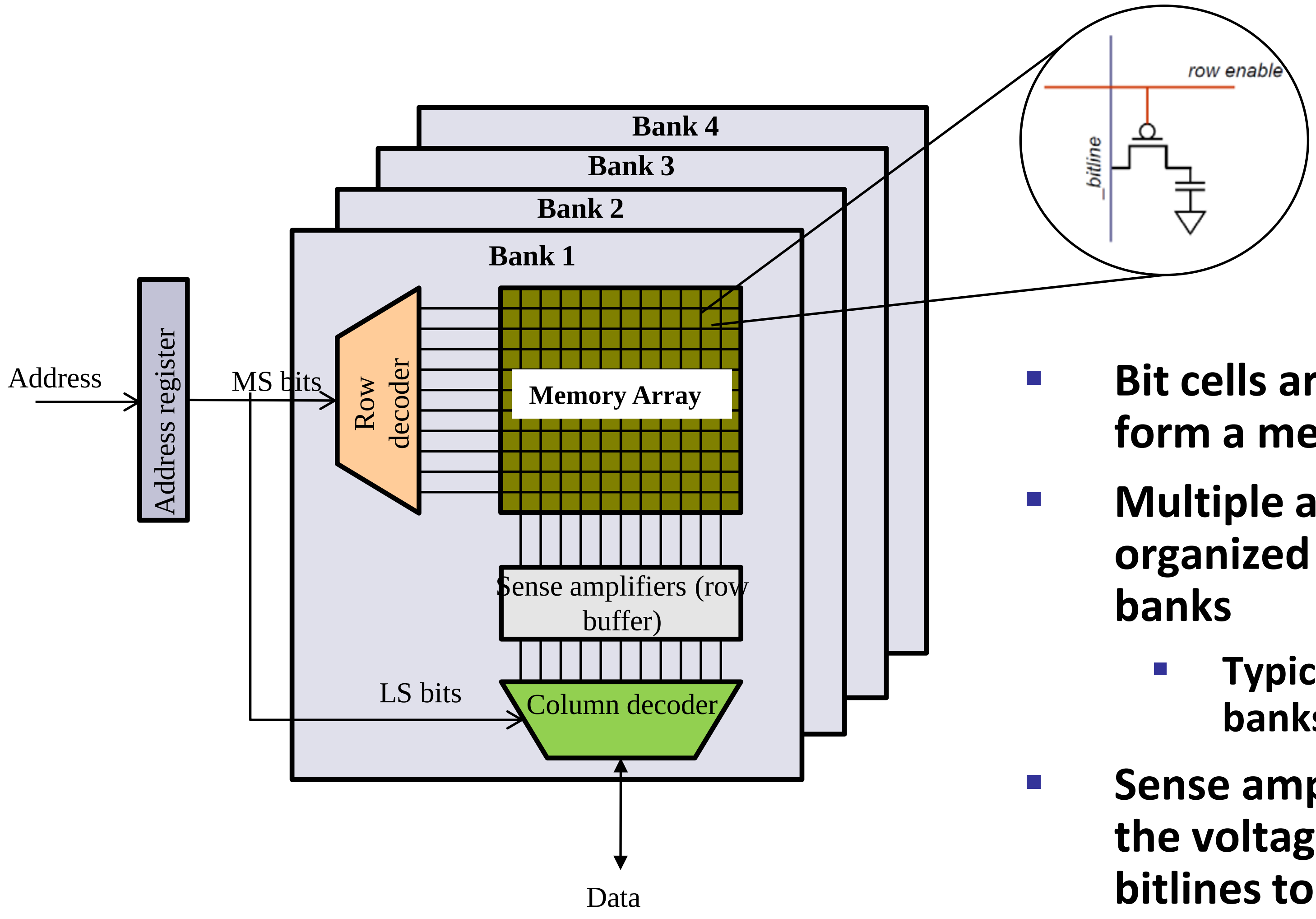
8-7


Row decoder
2-to-4 Decoder
A3
A2
2^1
2^0
0
1
2
3
Row select
RAM cell 0
RAM cell 1
RAM cell 2
RAM cell 3
RAM cell 4
RAM cell 5
RAM cell 6
RAM cell 7
RAM cell 8
RAM cell 9
RAM cell 10
RAM cell 11
RAM cell 12
RAM cell 13
RAM cell 14
RAM cell 15
Read/Write logic
Data in
Data out
Read/Write
Bit select
Data input
Read/Write
Column select
0
1
2
3
Column decoder
2-to-4 Decoder with enable
Data output

## 256K x 8 bits RAM using 64K x 8 bits RAMs



# DRAM: Internal architecture



- Bit cells are arranged to form a memory array
- Multiple arrays are organized as different banks
  - Typical number of banks are 4, 8 and 16
- Sense amplifiers raise the voltage level on the bitlines to read the data out

*Credits: J.Leverich, Stanford*

## Design of memory chips

- **Exercises**
  - Design a 1 G x 8 bits memory using 128 M x 8 bits memory chips
  - Design a 1 G x 8 bits memory using 256 M x 8 bits memory chips
  - Design a 4 G x 32 bits memory using 512 M x 16 bits memory chips

## Problem Statement

- Consider a cache with 64 blocks and a block size of 16 bytes. To what block number does byte address 1200 map?

**Solution**

- With 16 bytes per block, byte address 1200 is block address

$$\left\lfloor \frac{1200}{16} \right\rfloor = 75$$

- Cache block number (75 modulo 64) = 11
- This block maps all addresses between 1200 and 1215

## Problem Statement

- How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?

- Total Words: 16 KB is 4K (2^12) words
- Total Blocks = 1024 (2^10) blocks
- Each block has Data bits + Tag bits + valid bit
  - Data bits = 4 × 4 x 8 =128 bits
  - Tag is 32 – 10 – 2 – 2 bits
  - valid bit
- The total cache size

$$2^{10} \times (4 \times 32 + (32 - 10 - 2 - 2) + 1) = 2^{10} \times 147 = 147 \text{ Kbits}$$

## Measuring and Improving Cache Performance

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory-stall clock cycles}) \times \text{Clock cycle time}$$

$$\text{Memory-stall clock cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

## CPU time

- CPU time = IC x (CPI execution + Memory stalls per instruction) x Clock cycle time

- CPU time = IC x (CPI execution + Mem accesses per instruction x Miss rate x Miss penalty) x Clock cycle time (includes hit time as part of CPI)

- Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

## Multilevel Cache Example

**Given**

- CPU base CPI = 1, clock rate = 4GHz
- Miss rate/instruction = 2%
- Main memory access time = 100ns

- **Miss penalty** = 100ns/0.25ns = 400 cycles

- **Effective CPI** = 1 + 0.02 × 400 = 9

# Exercise

**Let us consider a computer with a L1 cache and L2**

**Processor Clock Frequency = 1 GHz ;**
**Hit Time L1 = 1 clock cycle ; Hit Rate L1 = 95%;**
**Hit Time L2 = 5 clock cycles; Hit Rate L2 = 90% ; Miss Penalty L2 = 15 cc;**
**Memory Accesses Per Instruction = 78% ; CPI exec = 3**

1. **How much is the Global Miss Rate for Last Level Cache?**

   **Miss Rate L1 L2 = Miss RateL1 x Miss RateL2= 0.05 x 0.1 = 0.005 = 0.5%**

2. **How much is the AMAT?**

   **AMAT = Hit TimeL1 + Miss RateL1 x (Hit TimeL2 + Miss RateL2 x Miss PenaltyL2)**

   **= 1 clock cycle + 0.05 x (5 + 0.1 x 15) clock cycles = 1.325 clock cycles**

3. **How much is the CPU time**

   **CPU-time = IC x (CPI exec + MAPI x MRL1 x HTL2 + MAPI x MRL1 L2 x MPL2 ) x TCLK**

   **= IC x (3 + 0.78 x 0.05 x 5 + 0.78 x 0.005 x 15) x TCLK**

   **= IC x 3.2535 x TCLK**

## Problem Statement

**Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls and the miss penalty is 100 cycles for all misses, determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.**

## Solution

Instruction miss cycles = I × 2% × 100 = 2.00 × I

Data miss cycles = I × 36% × 4% × 100 = 1.44 × I

Total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I

Total CPI including memory stalls is 2 + 3.44 = 5.44

$$\frac{\text{CPU time with stalls}}{\text{CPU time with perfect cache}} = \frac{I \times CPI_{stall} \times \text{Clock cycle}}{I \times CPI_{perfect} \times \text{Clock cycle}}$$

$$= \frac{CPI_{stall}}{CPI_{perfect}} = \frac{5.44}{2}$$

i.e. the stalls are reducing the performance of the computer by 2.72 times

## Observations

- What if we increase the CPU performance without improving memory?
    - If we improve the performance in the previous problem such that CPI becomes 1 instead of 2 then

  Total CPI including memory stalls is 1 + 3.44 = 4.44

    Ratio of performance will become 4.44/1 = 4.44

i.e. performance with memory stall will be decreased by 4.44 times

- What if we improve clock rate without improving memory?
    - Same effect as described above

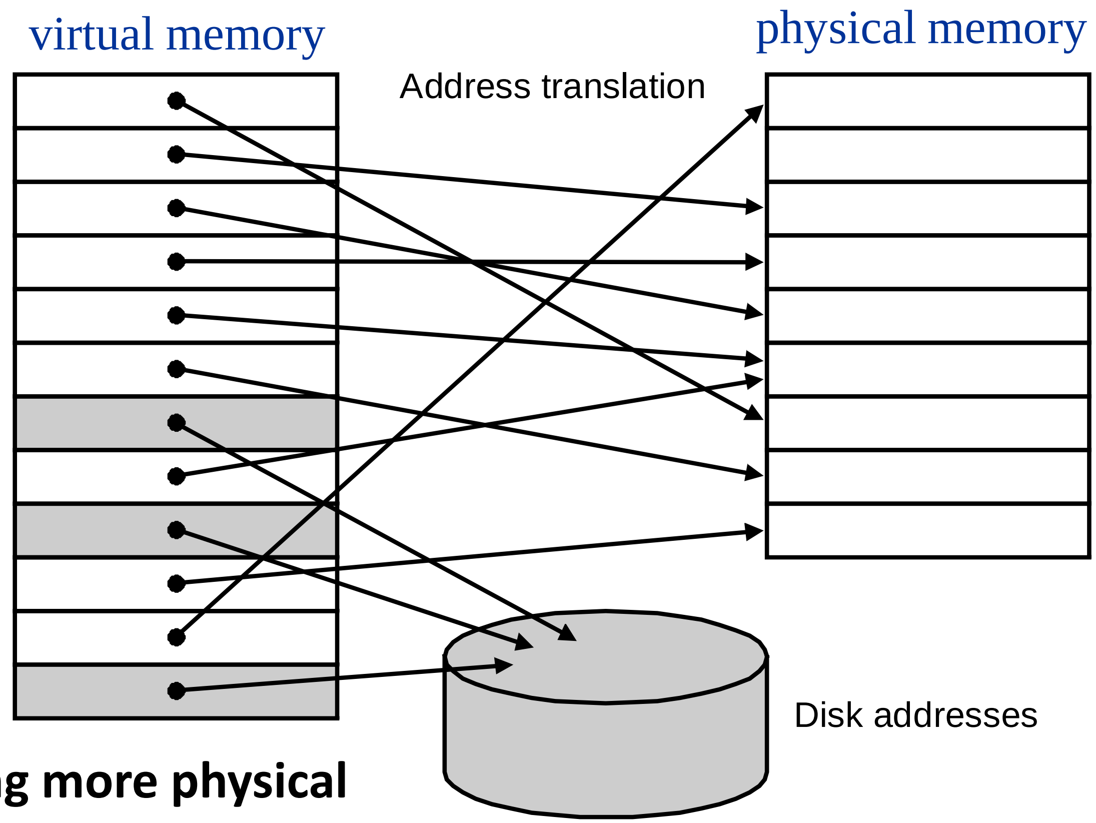
## Handling Writes

- Inconsistent States of cache and memory

- Solutions?
  - Write-through
    - Issue: Performance
  - Write buffer
    - Issue: Occurrence of stalls
  - Write-back
    - Issue: Harder to implement

# Virtual Memory

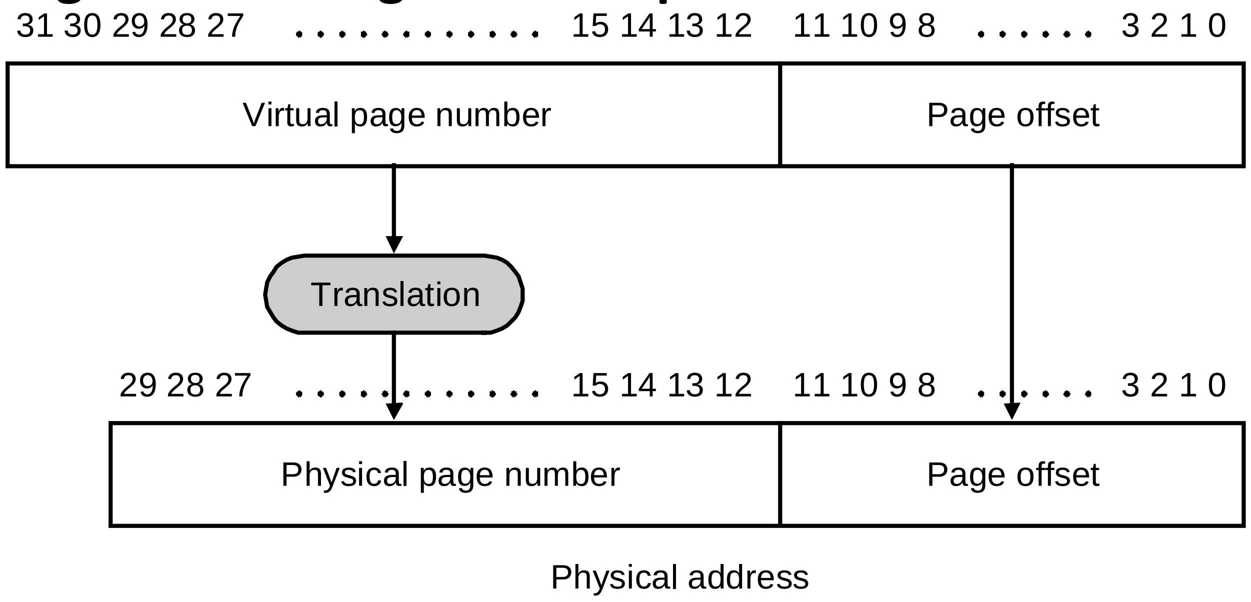
- Main memory can act as a cache for the secondary storage (disk)



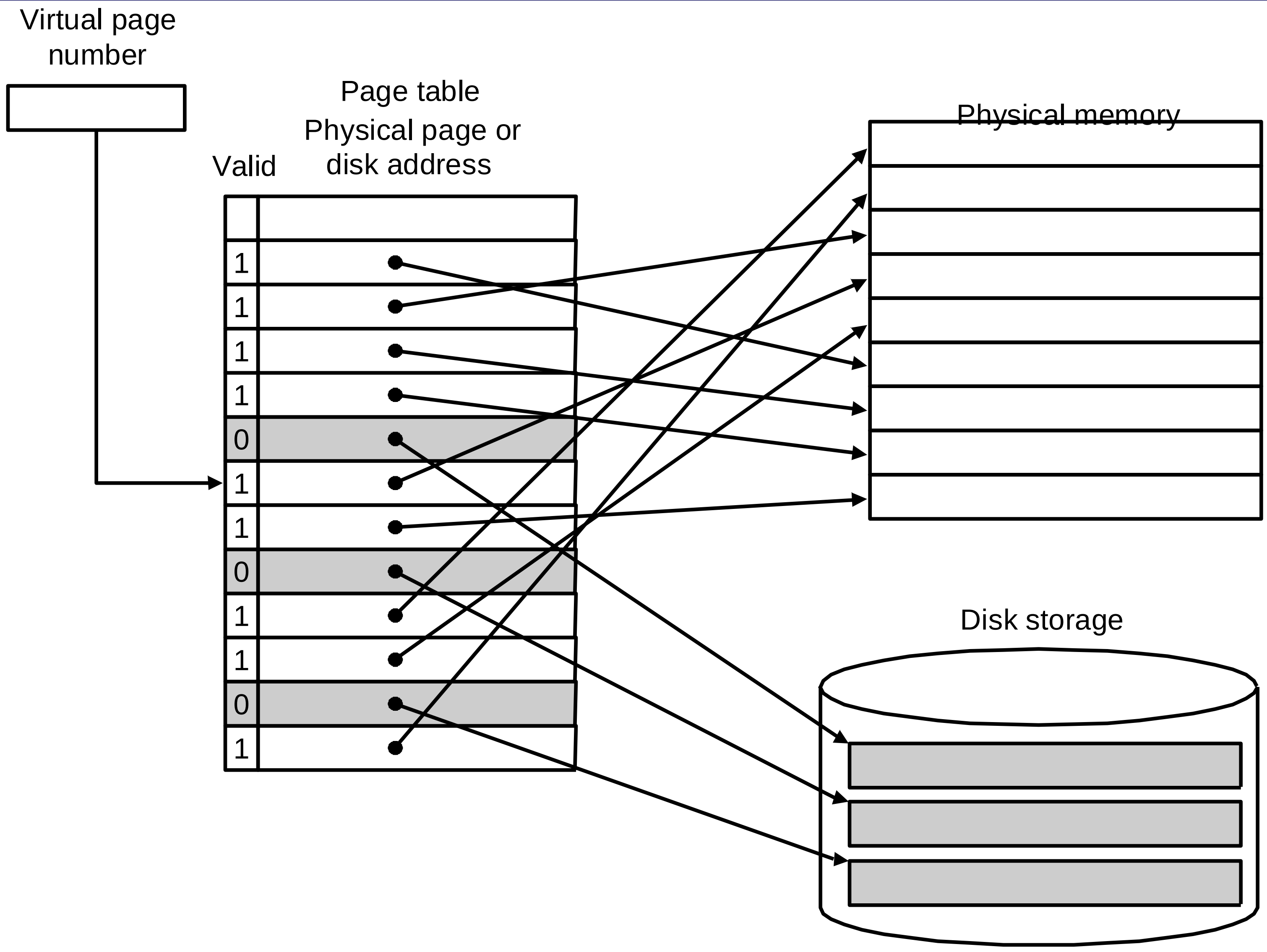
- Advantages:
  - illusion of having more physical memory
  - program relocation
  - protection

## Pages: virtual memory blocks

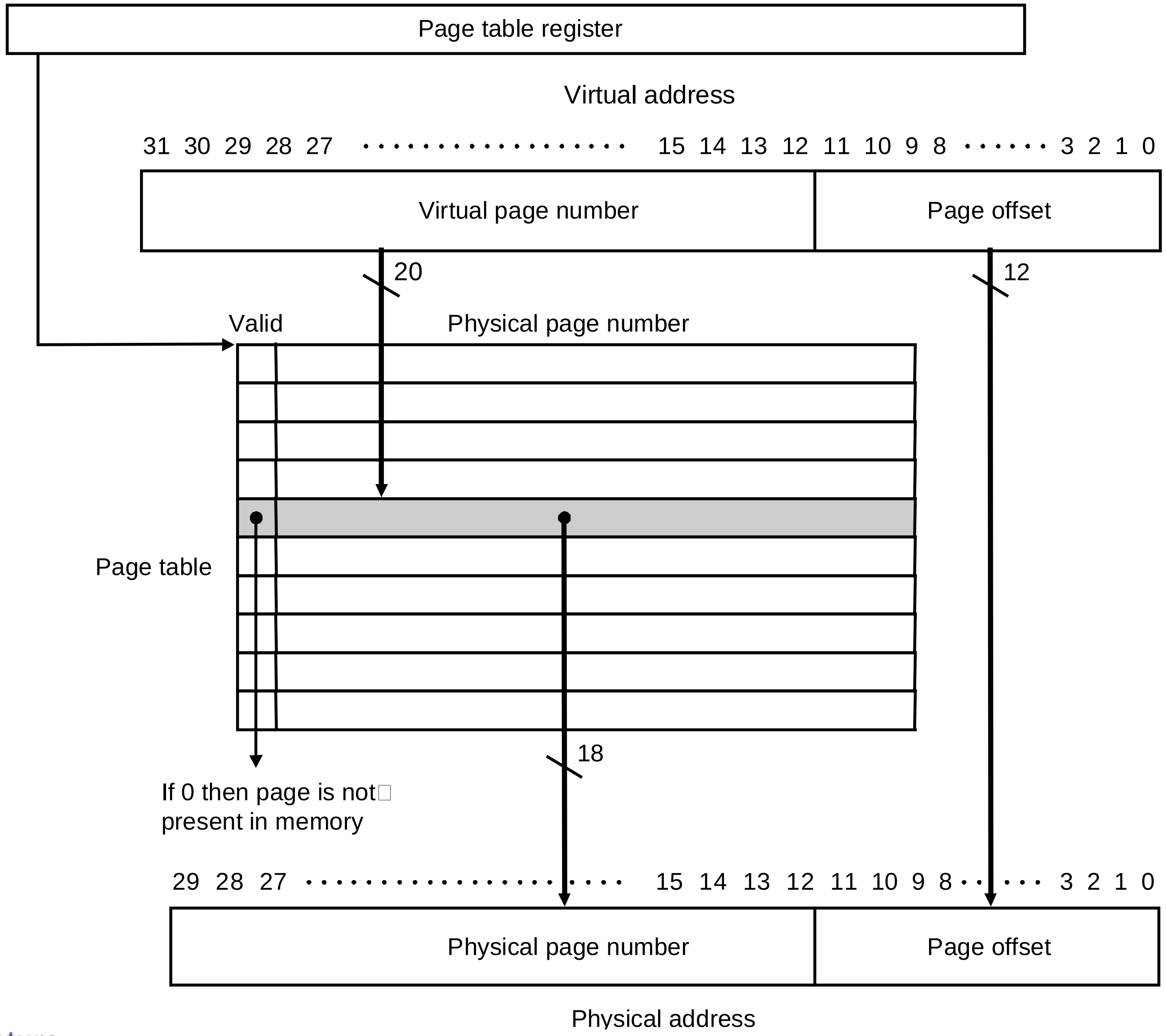
- Page faults: the data is not in memory, retrieve it from disk
  - huge miss penalty, thus pages should be fairly large (e.g., 4KB)
  - reducing page faults is important (LRU is worth the price)
  - can handle the faults in software instead of hardware
  - using write-through is too expensive so we use writeback



# Page Tables


Virtual page number
Page table
Valid
Physical page or disk address
1
1
1
1
0
1
1
0
1
1
0
1
Physical memory
Disk storage

# Page Tables


Page table register
Virtual address
31 30 29 28 27 ・・・・・・・・・・・・・・・・・・ 15 14 13 12 11 10 9 8 ・・・・・・ 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
If 0 then page is not present in memory
18
29 28 27 ・・・・・・・・・・・・・・・・・・・・・・ 15 14 13 12 11 10 9 8 ・・・・・ 3 2 1 0
Physical page number
Page offset
Physical address

# Size of page table

- **Assume**
  - **40-bit virtual address; 32-bit physical**
  - **4 Kbyte pages; 4 bytes per page table entry (PTE)**

- Solution
  - Size = $N_{entries}$ * Size-of-entry = $2^{40} / 2^{12}$ * 4 bytes = 1 Gbyte

- Reduce size:
  - Dynamic allocation of page table entries
  - Hashing: inverted page table
    - 1 entry per physical available instead of virtual page
  - Page the page table itself (i.e. part of it can be on disk)
  - Use larger page size (multiple page sizes)

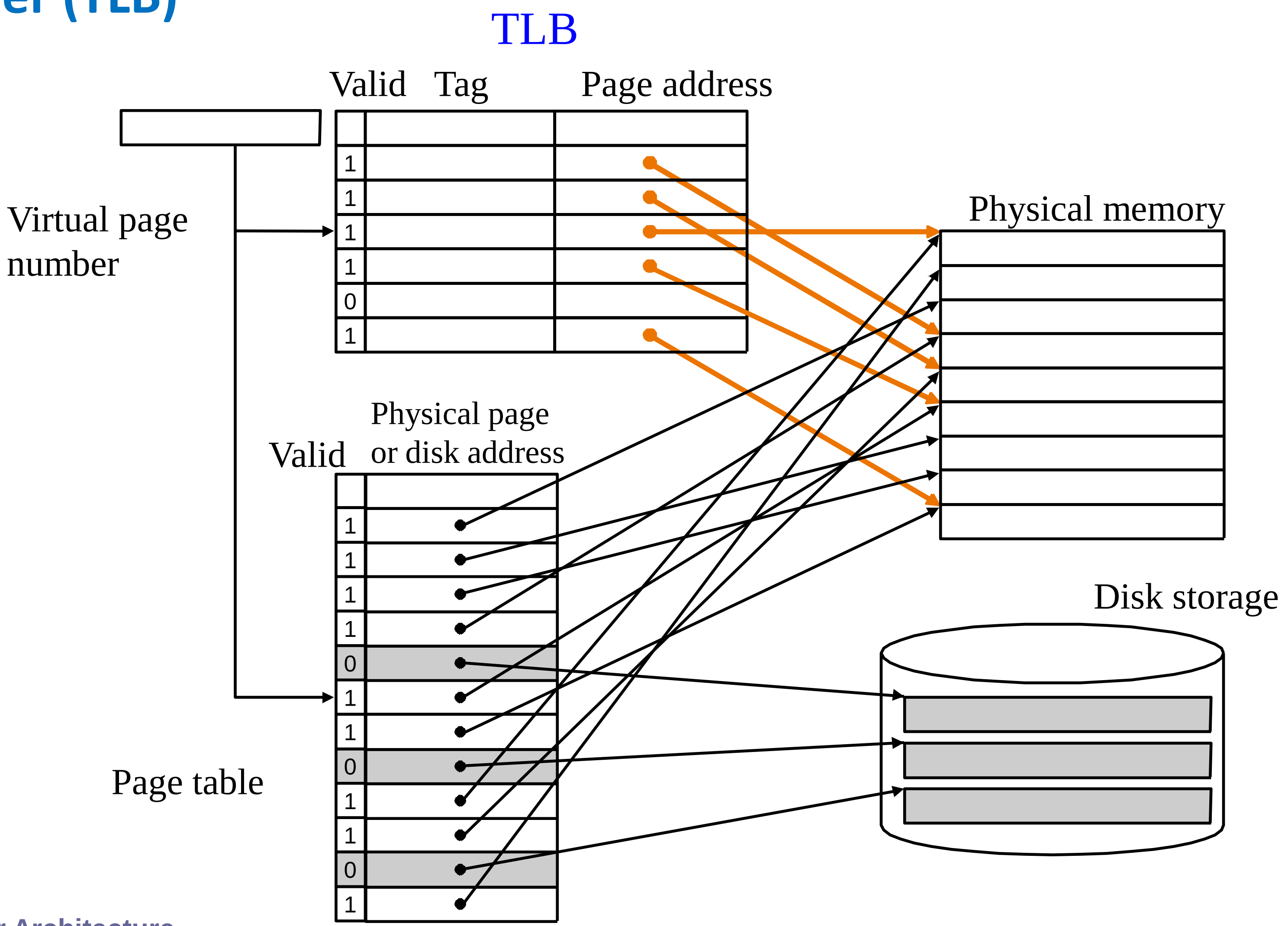
## Fast Translation Using a TLB

- Address translation would appear to require extra memory references
  - One to access the PTE (page table entry)
  - Then the actual memory access

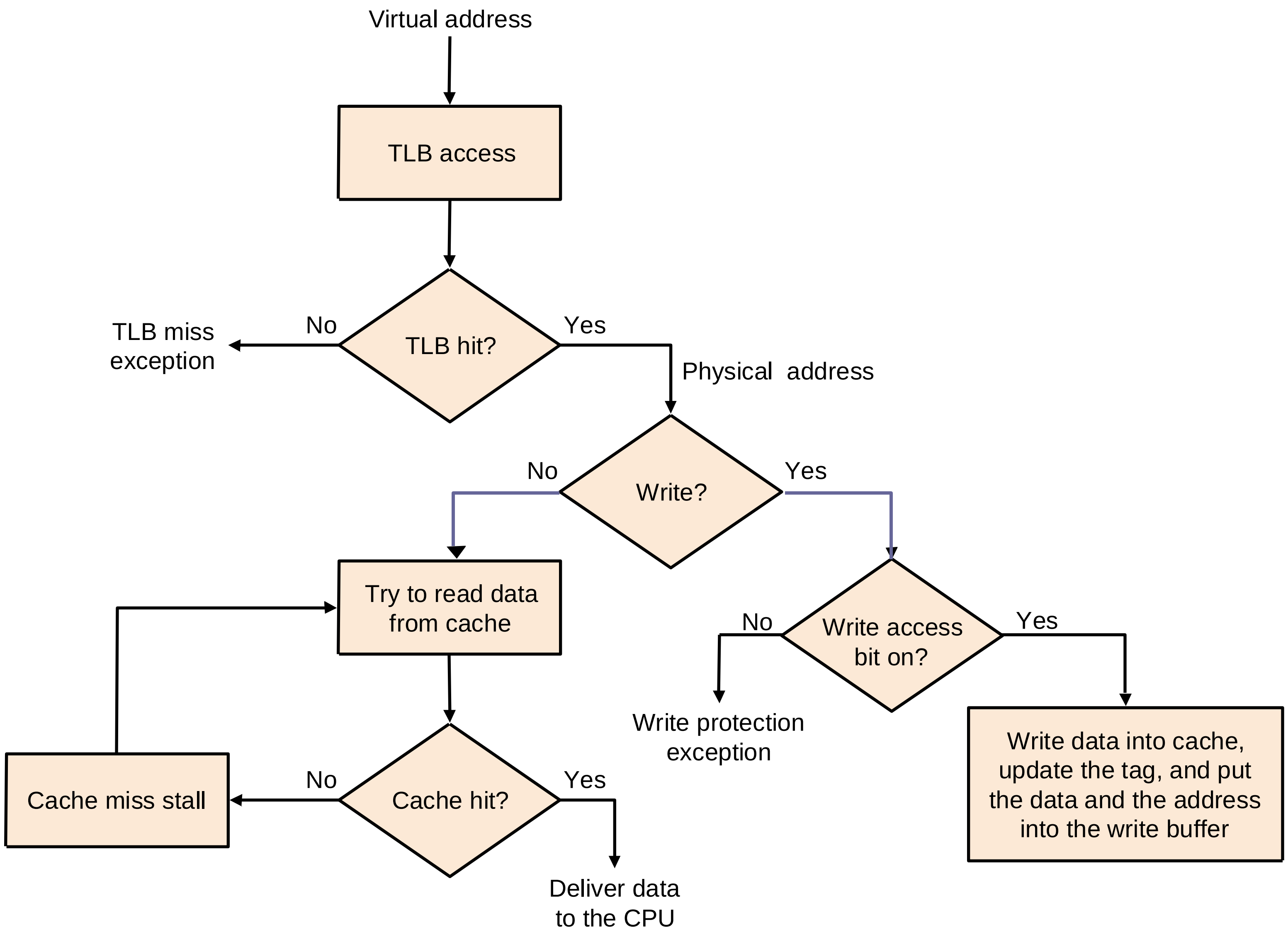
- However access to page tables has good locality
  - So use a fast cache of PTEs within the CPU
  - Called a Translation Look-aside Buffer (TLB)
  - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  - Misses could be handled by hardware or software

# Making Address Translation Fast

- **A cache for address translations: translation lookaside buffer (TLB)**



# TLBs and caches

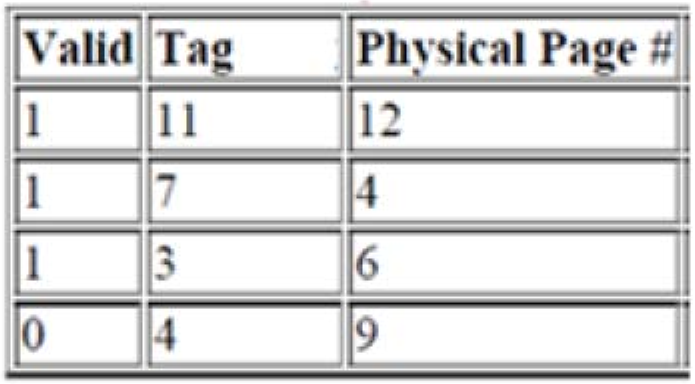


# Overall operation of memory hierarchy

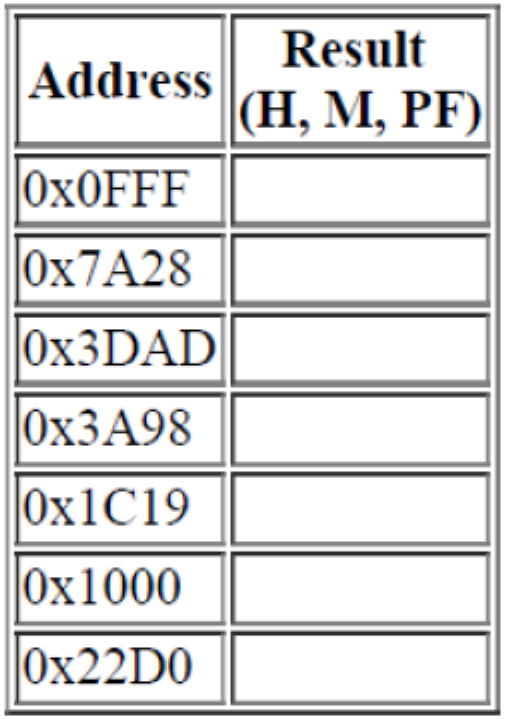
- **Each instruction or data access can result in three types of hits/misses: TLB, Page table, Cache**
- **Q: which combinations are possible? Check them all!**

| TLB | Page table | Cache | Possible? |
|---|---|---|---|
| hit | hit | hit | Yes, that’s what we want |
| hit | hit | miss | Yes, but page table not checked if TLP hit |
| hit | miss | hit | no |
| hit | miss | miss | no |
| miss | hit | hit | |
| miss | hit | miss | |
| miss | miss | hit | no |
| miss | miss | miss | |

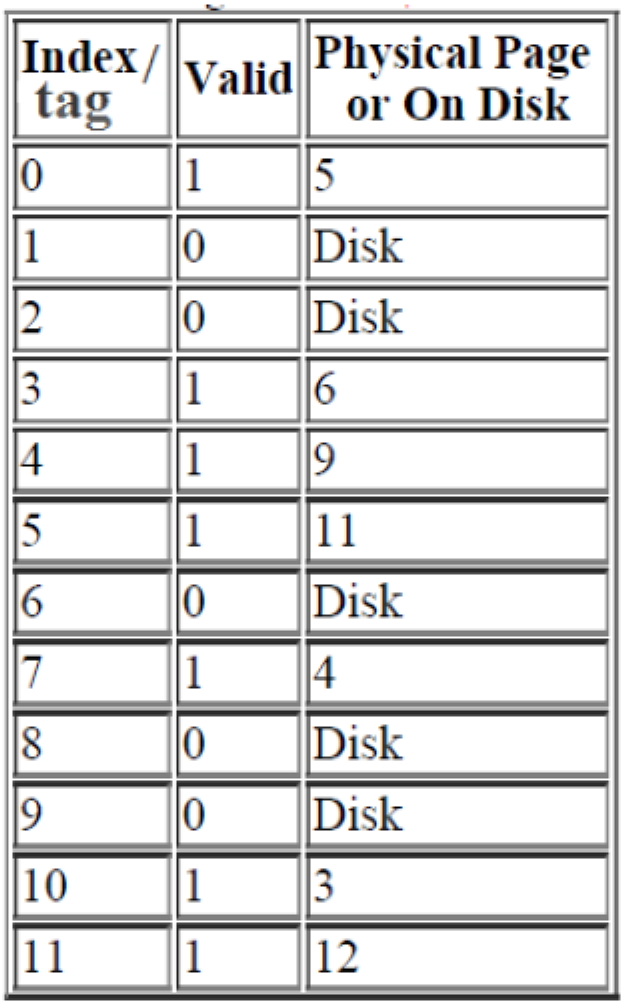
## Page size is 4KB and TLB cache size is 4 lines



| Valid | Tag | Physical Page # |
|---|---|---|
| 1 | 11 | 12 |
| 1 | 7 | 4 |
| 1 | 3 | 6 |
| 0 | 4 | 9 |



| Address | Result (H, M, PF) |
|---|---|
| 0x0FFF | |
| 0x7A28 | |
| 0x3DAD | |
| 0x3A98 | |
| 0x1C19 | |
| 0x1000 | |
| 0x22D0 | |



| Index / tag | Valid | Physical Page or On Disk |
|---|---|---|
| 0 | 1 | 5 |
| 1 | 0 | Disk |
| 2 | 0 | Disk |
| 3 | 1 | 6 |
| 4 | 1 | 9 |
| 5 | 1 | 11 |
| 6 | 0 | Disk |
| 7 | 1 | 4 |
| 8 | 0 | Disk |
| 9 | 0 | Disk |
| 10 | 1 | 3 |
| 11 | 1 | 12 |