# Data Structures and Algorithms

(CS210A)

## Lecture 13:

- **Majority element** : an efficient and practical algorithm
- **word RAM** model of computation: further refinements.

# Majority element

**Definition**: Given a **multiset** $S$ of $n$ elements,

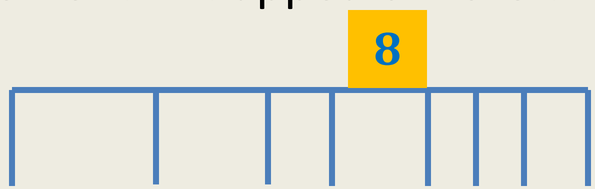
$x \in S$ is said to be majority element **if** it appears more than $n/2$ times in $S$.

$$S = \{a, b, d, z, b, w, b, b, z, b, b, b, b, c\}$$

# Majority element

**Definition**: Given a **multiset** $S$ of $n$ elements,

$x \in S$ is said to be majority element **if** it appears more than $n/2$ times in $S$.




$$S = \{a, b, d, z, b, w, b, b, z, b, b, b, b, c\}$$

**Problem**: Given a **multiset** $S$ of $n$ elements, find the majority element, if any, in $S$.

# Majority element

## **Trivial algorithms**:

**Algorithm 1**:

1. Count occurrence of each element
2. If there is any element with count > $\frac{n}{2}$ , report it.

**Running time**: O($n^2$) time

# Majority element

## Trivial algorithms:

**Algorithm 2**:

1. Sort the set $S$ to find its median
2. Let $x$ be the median
3. Count the occurrence of $x$, and
4. return $x$ if its count is more than $\frac{n}{2}$

This assumption is **not justified** in many real life applications.

**Running time**: O($n$ **log** $n$) time

**Critical assumption underlying Algorithm 2** :

elements of set $S$ can be compared under some <u>total order</u> $(=,<,>)$

# A real life application

Slots for inserting any two cards


Card-matching machine

**Problem**:
Given $n$ credit cards, determine if th
using minimum no. of operations on ca

This machine takes two cards and determines whether they are identical or not.

# Some observations

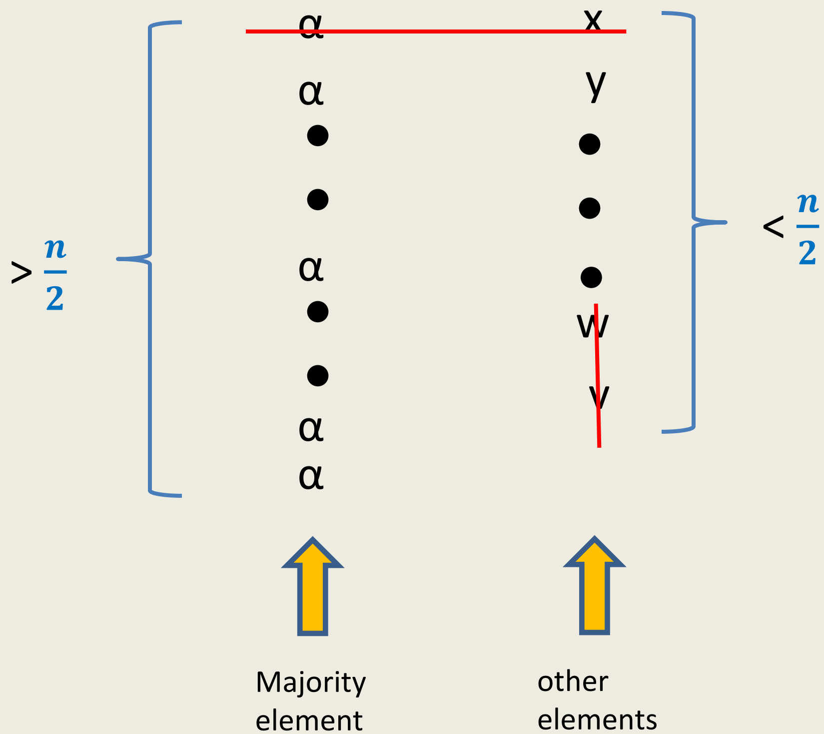
**Problem**: Given a **multiset** $\boldsymbol{S}$ of $\boldsymbol{n}$ elements,
where the only relation between any two elements is $\neq$ or $=$ ,
find the majority element, if any, in $\boldsymbol{S}$.

**Question:** How much time does it take to determine if an element $\boldsymbol{x} \in \boldsymbol{S}$ is majority ?

**Answer:** O($\boldsymbol{n}$) time

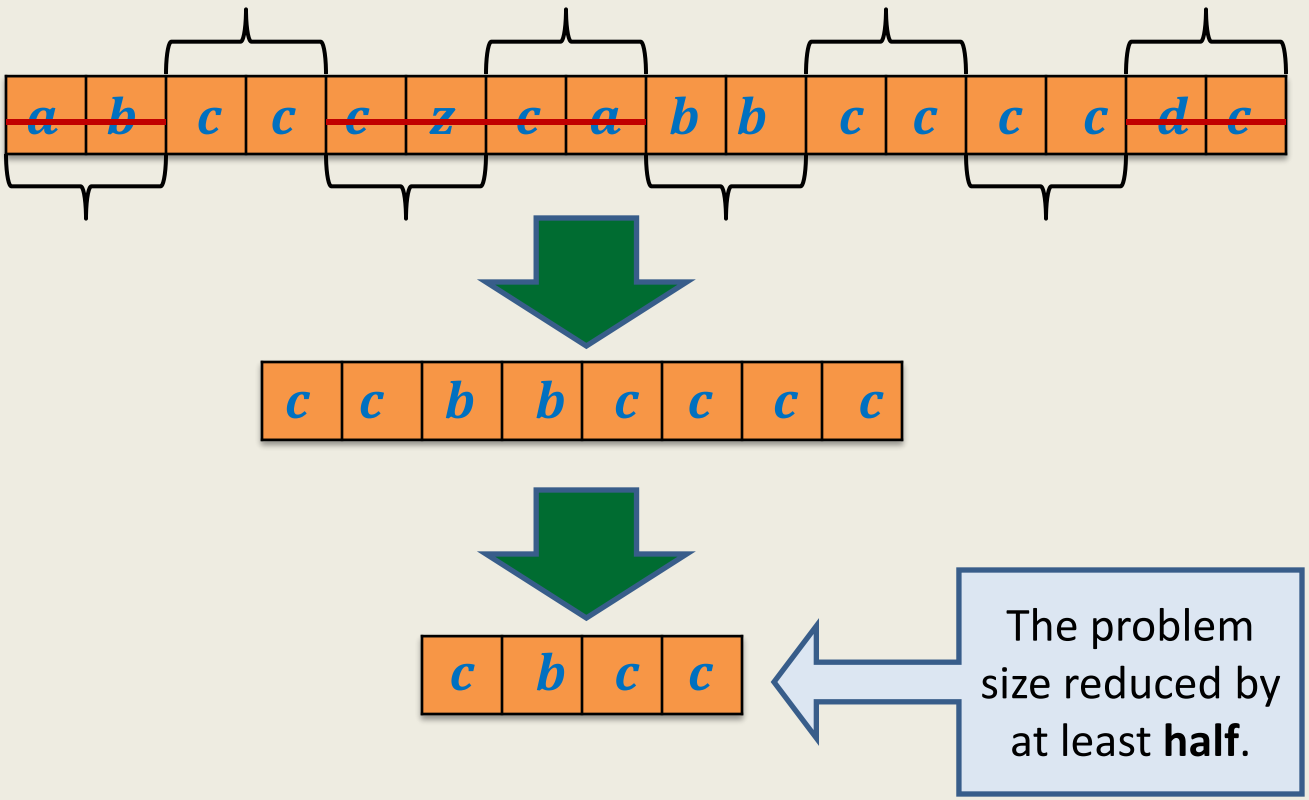
**Observation 1**: It is easy to verify whether an element is a majority

# Some observations




**Observation 2**: whenever we cancel a pair of distinct elements from the array, the majority element of the array remains preserved.

# Some observations




**Observation 3**: If there are $m$ pairs of **identical elements**, then majority element is preserved even if we keep **one element** per pair.

# Algorithm for 2-majority element

**Repeat**

1. Pair up the elements;
2. Eliminate all pairs of distinct elements;
3. Keep one element per pair of identical elements.

Take care if the no. of elements is odd

**Until** only one element is left.

Verify if the last element is a **majority** element.

**Time complexity:**

$\mathbf{T}(n) = c\, n + c\,\frac{n}{2} + c\,\frac{n}{4} + \ldots$

$O(n)$ time

**Extra/working space requirement** (assuming input is **"read only"**)

$O(n)$

# Further restrictions on the problem

**Restrictions**:

- We are allowed to make single scan.
- We have very limited extra space.

Our current algorithm doesn’t work for this real life example.

**Real life example**:

There are $10^{12}$ numbers stored on hard disk.

**RAM** can’t provide **O**($\boldsymbol{n}$) extra (working) space in this case.

# ALGORITHM FOR 2-MAJORITY ELEMENT

- **Single scan and**
- **O(1) extra space**

# Designing algorithm for 2-majority element
## single scan and using O(1) extra space

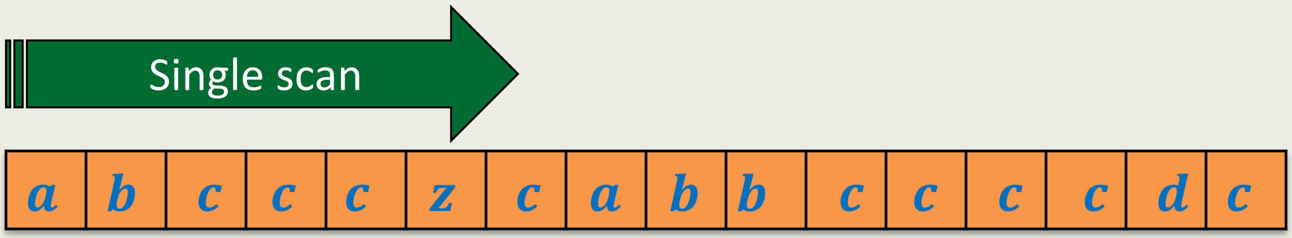
**Question**: Should we design algorithm from scratch to meet these constraints ?

**Answer**: No! We should try to adapt our current algorithm to meet these constraints.

**Question**: How crucial is pairing of elements in our current algorithm ?

Think over this question before going ahead.

# Designing algorithm for 2-majority element
## single scan and using O(1) extra space




**Insightful questions:**

- Do we really need to keep more than one element ?

  **No**. Just **cancel suitably** whenever encounter two *distinct* elements.

- Do we really need to keep multiple copies of an element **explicitly** ?

  **No**. Just keeping its **count** will suffice.

Ponder over these insights and make an attempt to design the algorithm before moving ahead ☺

# Algorithm for 2-majority element

## single scan and using O(1) extra space

**Algo-2-majority**(**A**)

{ count ← 0;

**for**($i = 0$ to $n - 1$)

{ **if** ( count=0 ){ **x** ← **A**[$i$];

count ← 1;

}

**else if**(**x**<> **A**[$i$]) count ← count - 1 ;

**else** count ← count + 1 ;

}

Count the occurrences of **x** in **A**, and **if** it is more than $n/2$, then

**print**(**x** is 2-majority element) **else print**(there is no majority element in A)

}

# Algorithm for 2-majority element
## single scan and using O(1) extra space

**Theorem**: There is an algorithm that makes just **a single scan** and uses **O**(1) **extra space** to compute majority element for a given multi-set.

**Homework**: Algorithm for 3-majority element

# Proving correctness of algorithm for 2-majority element

Optional Home work Exercise

Submit in the next class

“just the Assertion that holds at the end of each iteration”

# A problem in Practice sheet 1 (7 January)

**Merging two sorted arrays:**

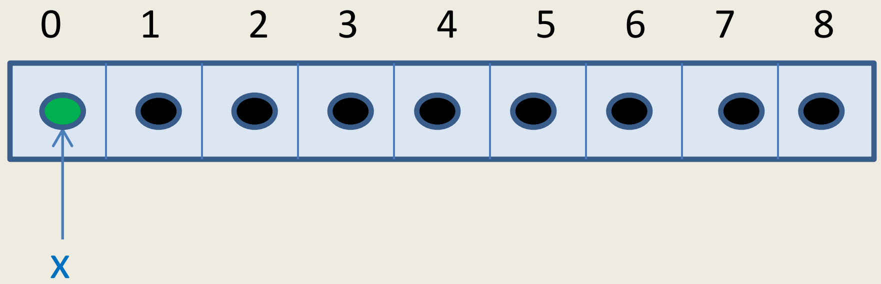
Given two sorted arrays **A** and **B** storing $n$ elements each,
Design an **O**($n$) time algorithm to output a sorted array **C** containing all elements of **A** and **B**.

**Example:**

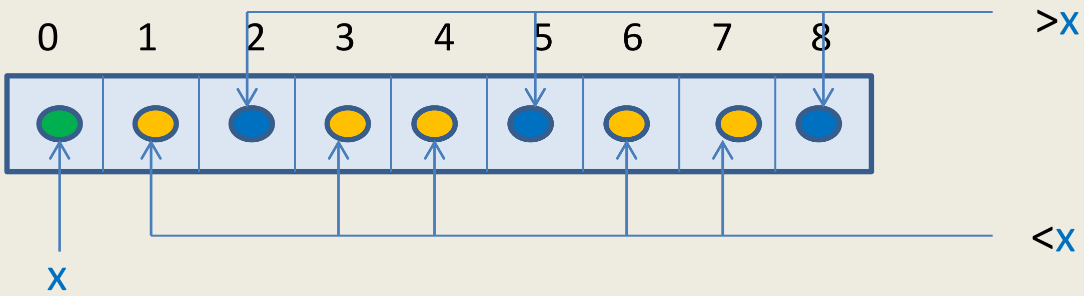
If **A**={1,5,17,19} **B=**{4,7,9,13}, then output is

**C**={1,4,5,7,9,13,17,19}.

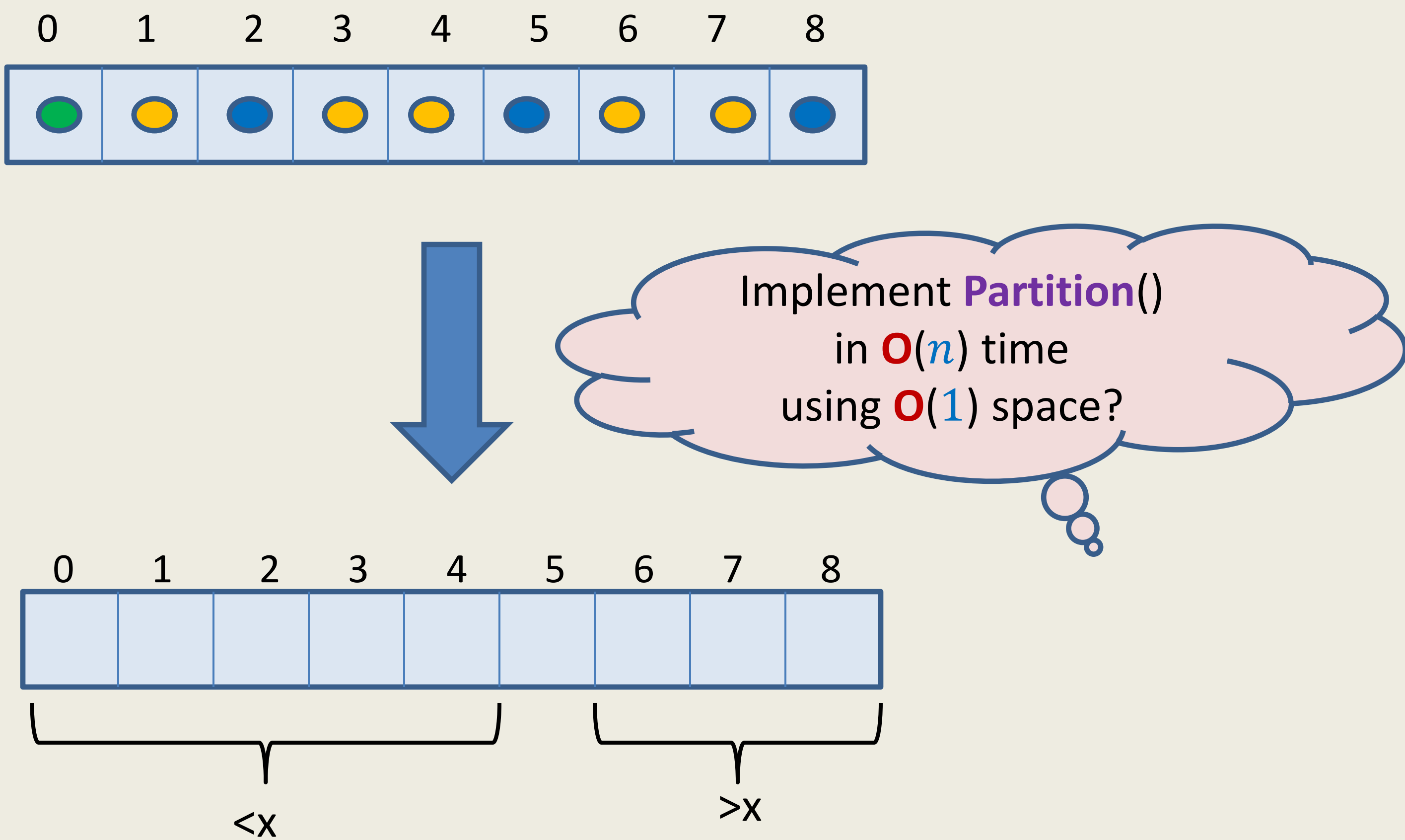
# A nice programming exercise ?



# A nice programming exercise ?



# A nice programming exercise ?



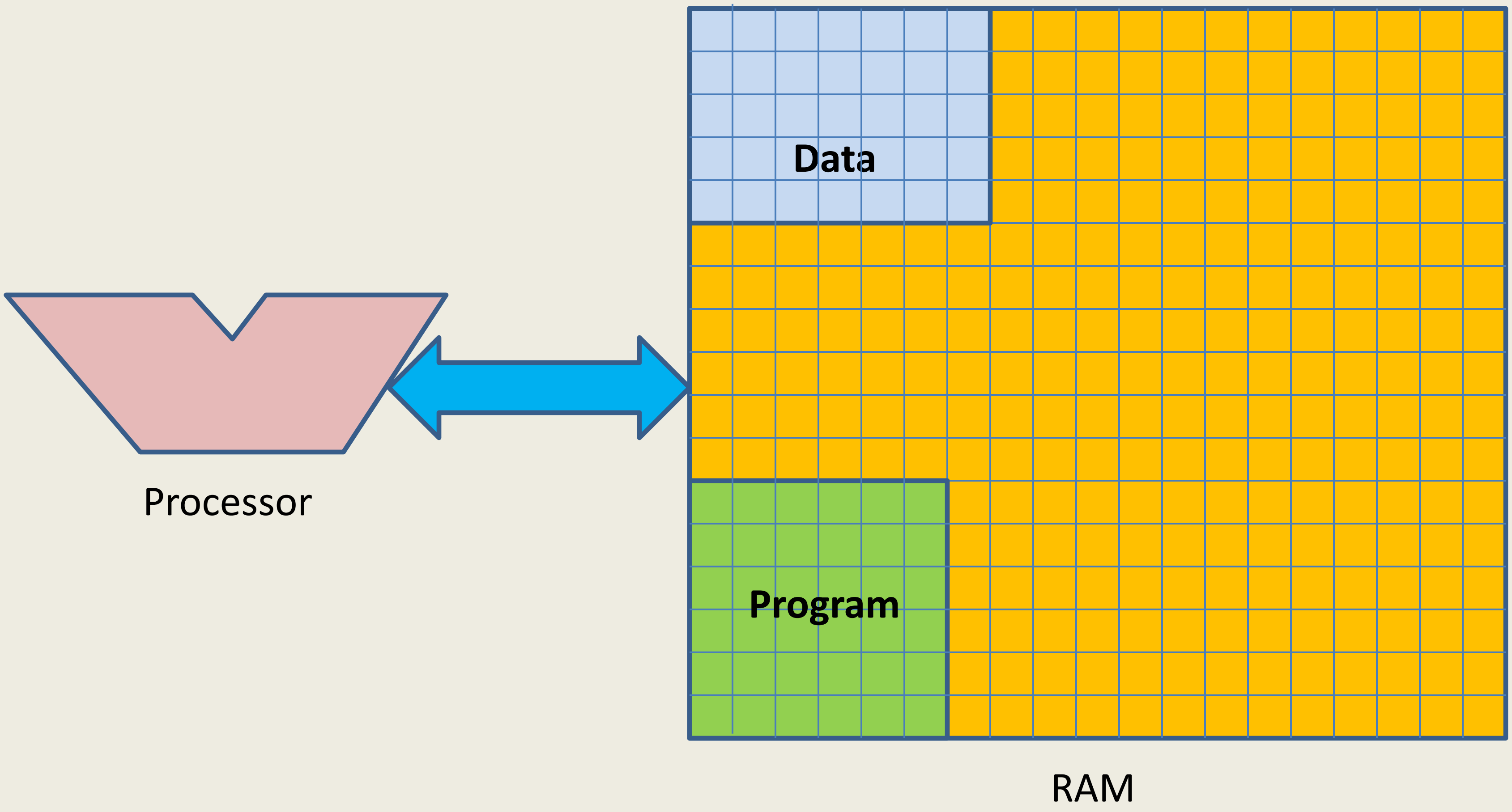

This procedure is called **Partition**.

It **rearranges** the elements so that all elements less than x appear to the left of x and all elements greater than x appear to the right of x.

# **Word RAM model of computation**

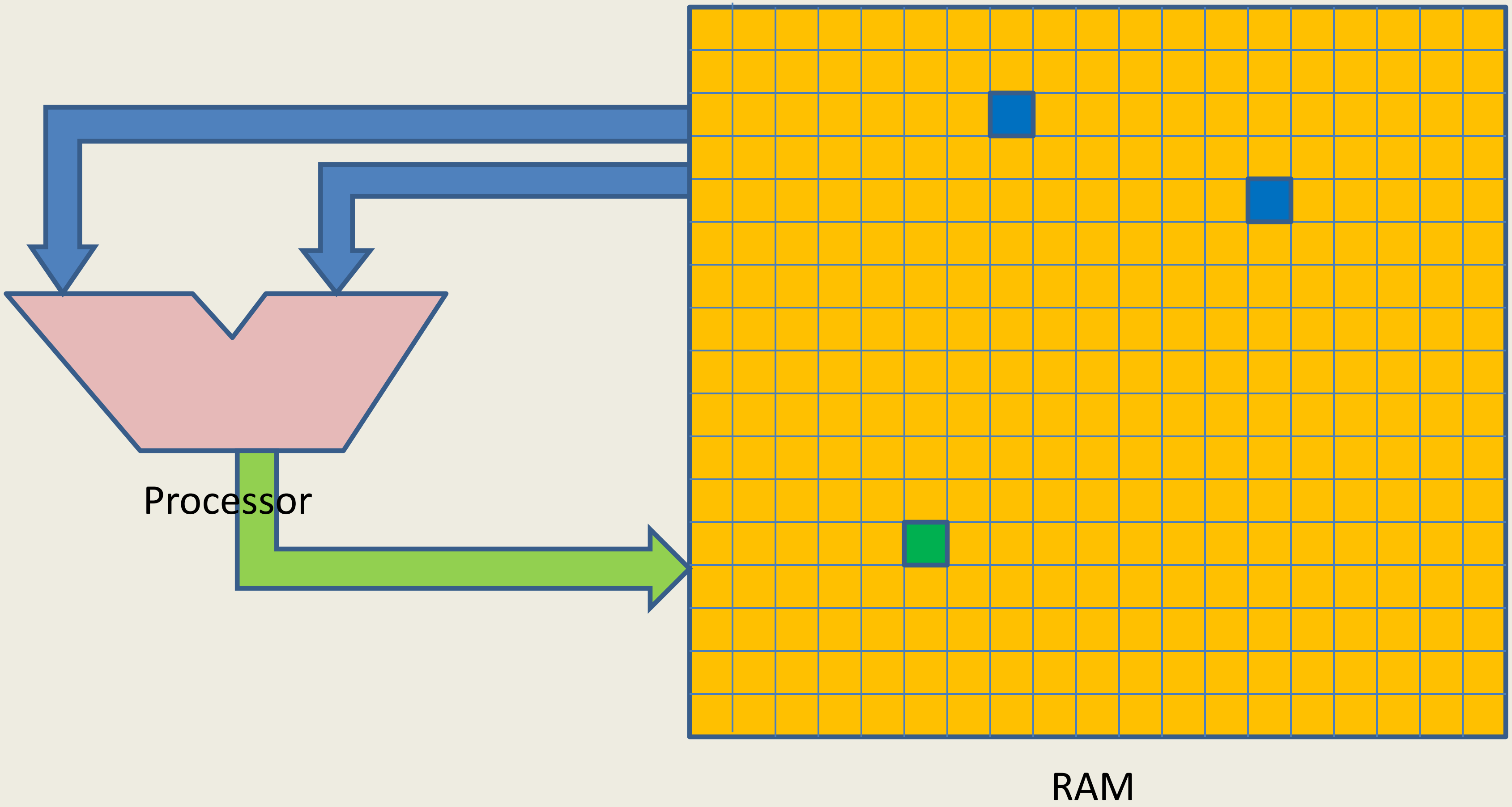
## **Further refinements**

# word RAM : a model of computation



# Execution of a instruction

(fetching the operands, arithmetic/logical operation, storing the result back into RAM)



# A more realistic RAM

**$n$ : input size**

**Input** resides completely in **RAM**.

**Question:** How many bits are needed to access an input item from **RAM** ?

**Answer:** At least **log $n$.**

($k$ bits can be used to create at most $2^k$ different addresses)

**Current-state-of-the-art computers:**

- RAM of size **4GB**

  Hence 32 bits to address any item in RAM.

- Support for **64-bit arithmetic**

  Ability to perform arithmetic/logical operations on any two 64-bit numbers.

# word RAM model of computation: Characteristics

- Word is the **basic storage** unit of RAM. Word is a collection of few bytes.
- Data as well as Program **reside fully** in RAM.
- Each input item (number, name) is stored in **binary format**.
- RAM can be viewed as a huge array of words. Any arbitrary location of RAM can be **accessed** in the same time **irrespective** of the location.
- Each arithmetic or logical operation (+,-,*,/,or, xor,…) involving **O( log $n$) bits** takes **a constant number of steps** by the CPU, where $n$ is the number of bits of input instance.