Lecture 8

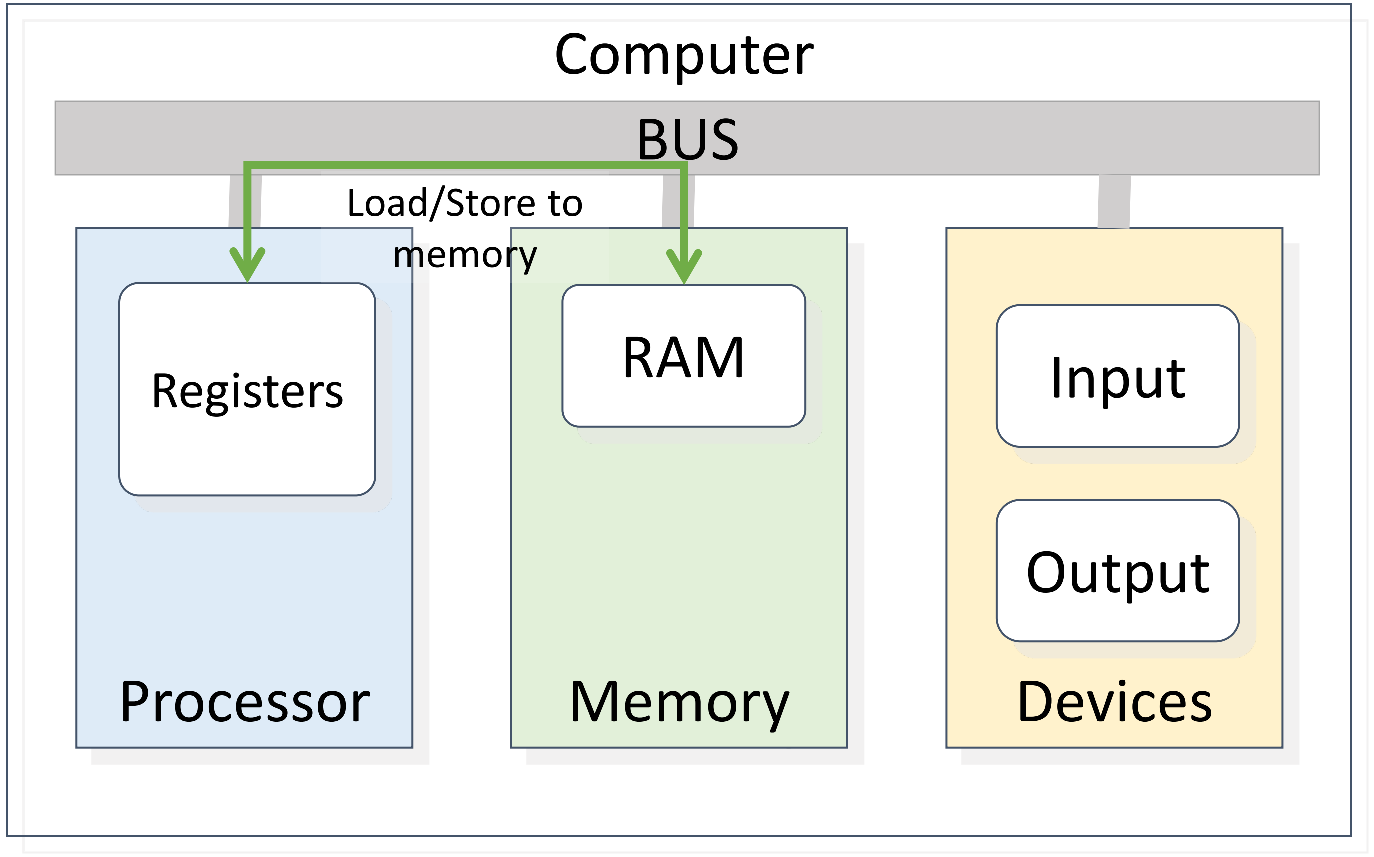
# Memory & Cache

## Overview

Memory
- Program usage of memory
- Hardware
  - Volatile vs Non-Volatile Memory
  - DRAM vs SRAM
  - Current trend of DRAM

Cache
- What is it?
- How does it work?
- Current trend of Cache

## Data Transfer: The BIG picture

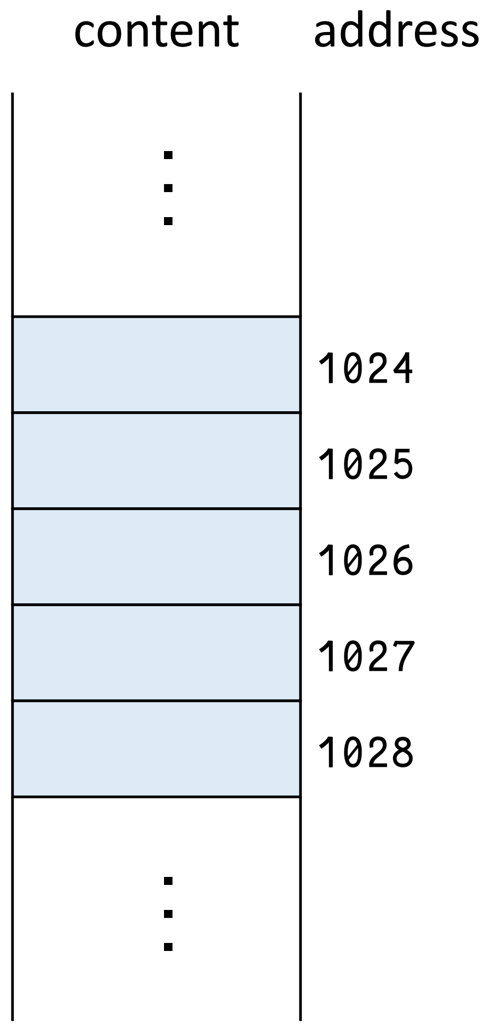


Registers are fast storage in the processor. If operands (data values) are in memory we have to load them to processor (registers), operate on them, and store them back to memory

## Main Memory Storage

### Physical memory storage

- Random Access Memory (RAM)
- Can be viewed as an array of bytes
- Each byte has a unique index
  - Known as physical address



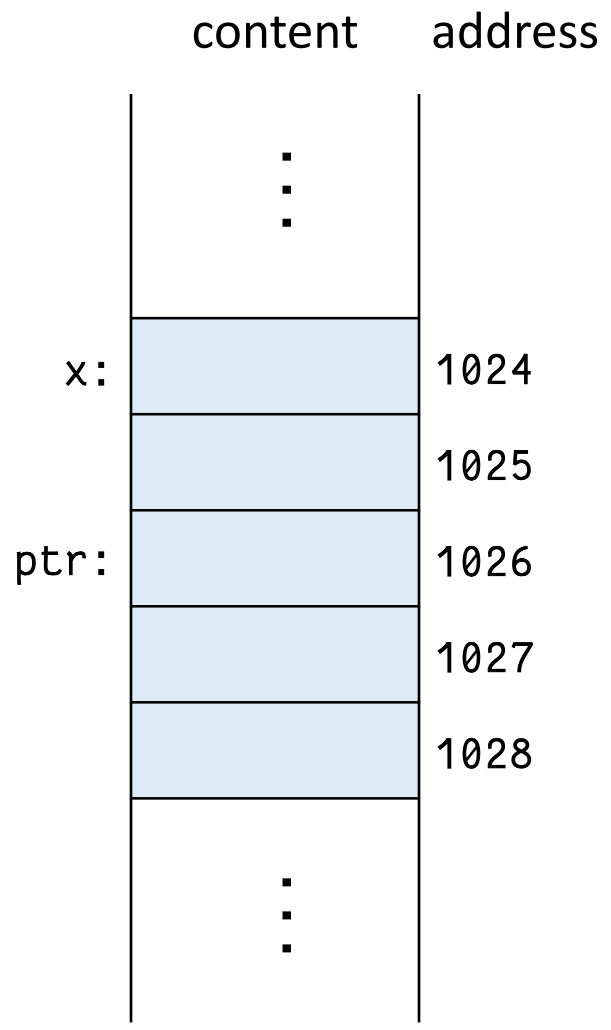
## Variable and Pointer

```
int x;
int *ptr;
ptr = &x;
*ptr = 123;
```

Normal variable is referred by variable name in code

– No need to know the address

A pointer contains the address of a memory location



## Pointers and Arrays

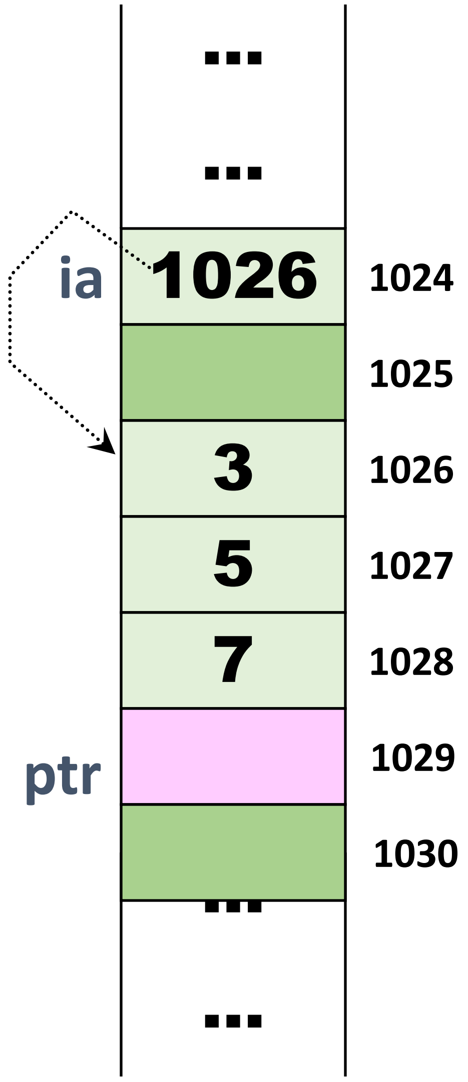
Array name is a **constant pointer**

– Points to the zeroth element

```
int ia[3] = {3, 5, 7};
```

- Is the following valid?

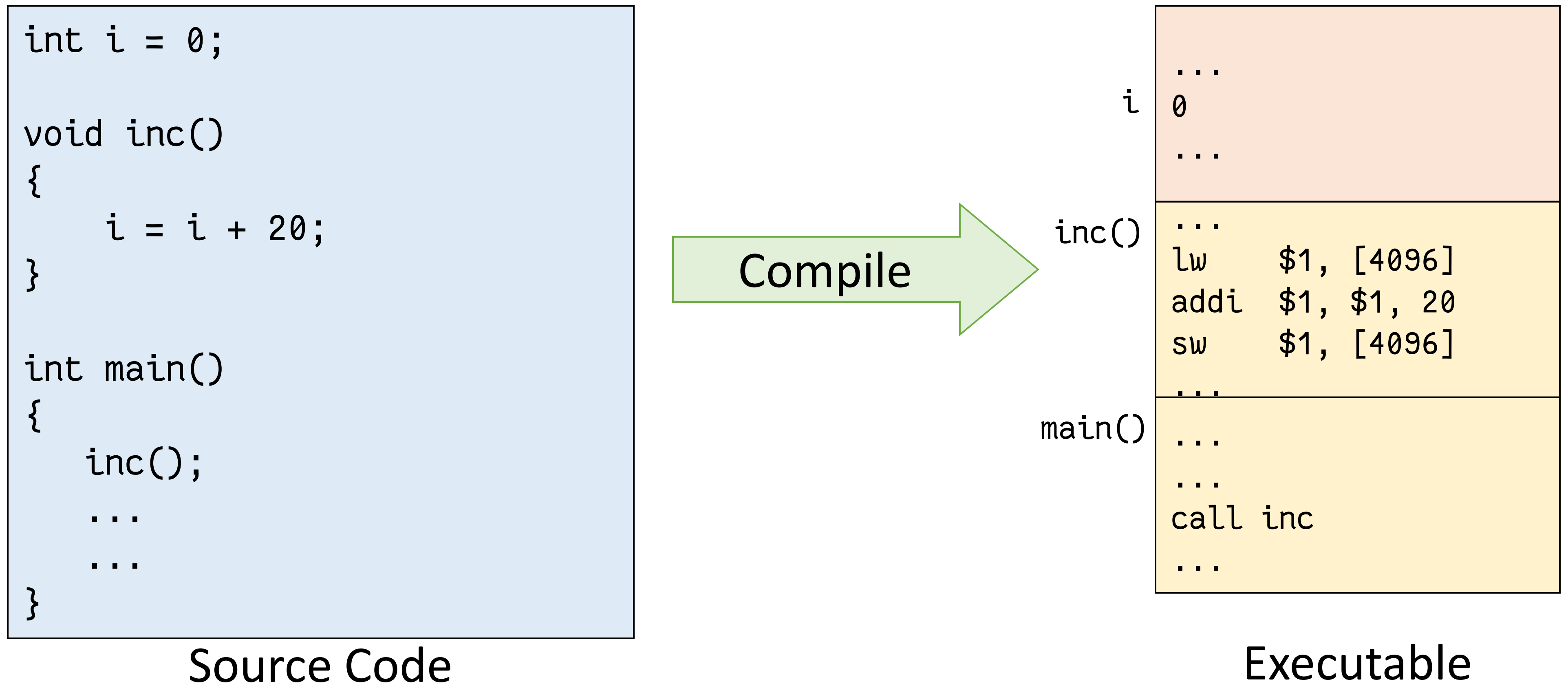
```
int* ptr;

ptr = ia;
ia = ptr;
ptr[2] = 9;

ptr = &ia[1];
ptr[1] = 11;
```



## Source Code → Executable

Executable typically contains:

– Code and Data layout



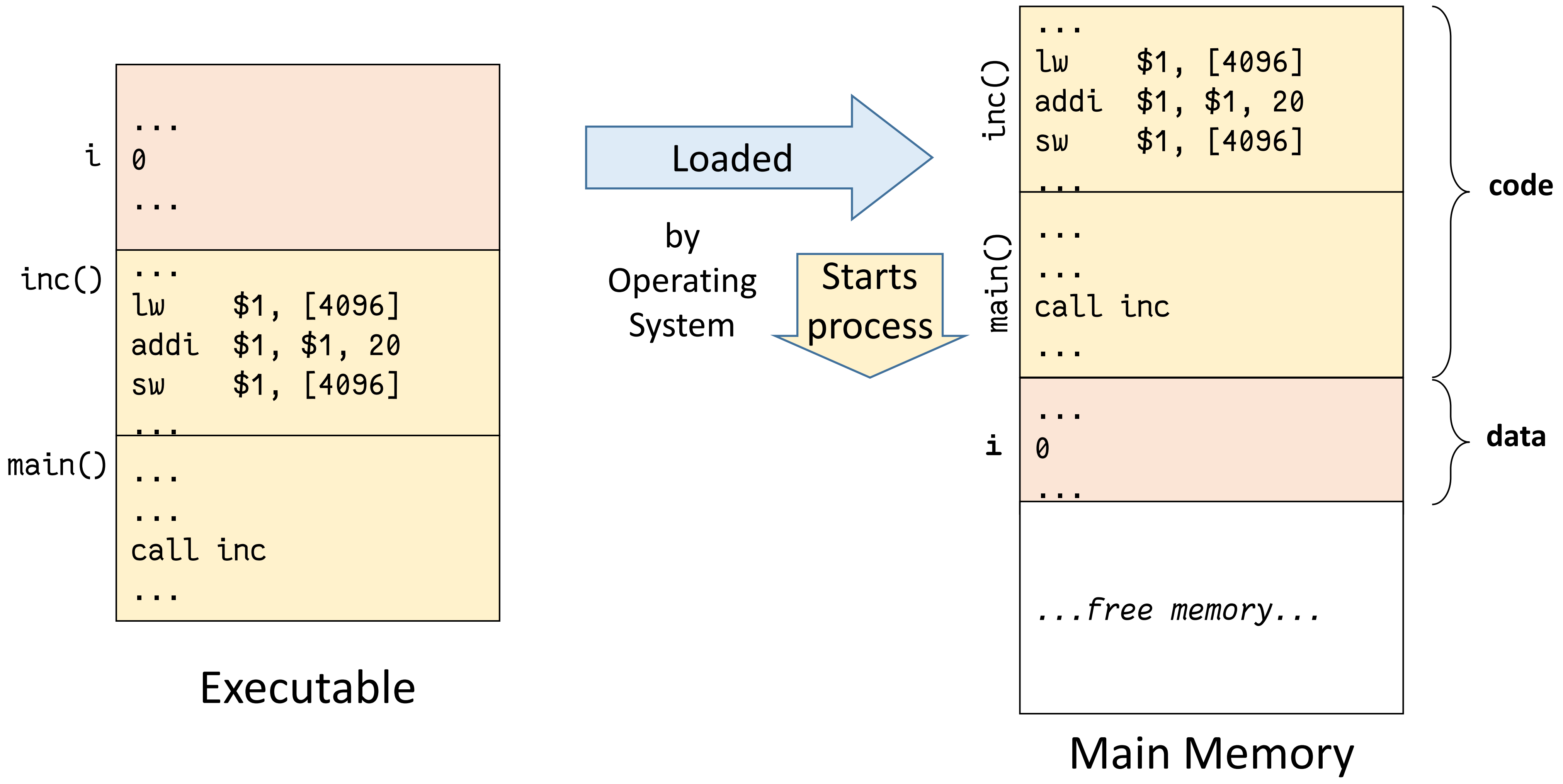
```
int i = 0;

void inc()
{
    i = i + 20;
}

int main()
{
   inc();
   ...
   ...
}
```

Source Code

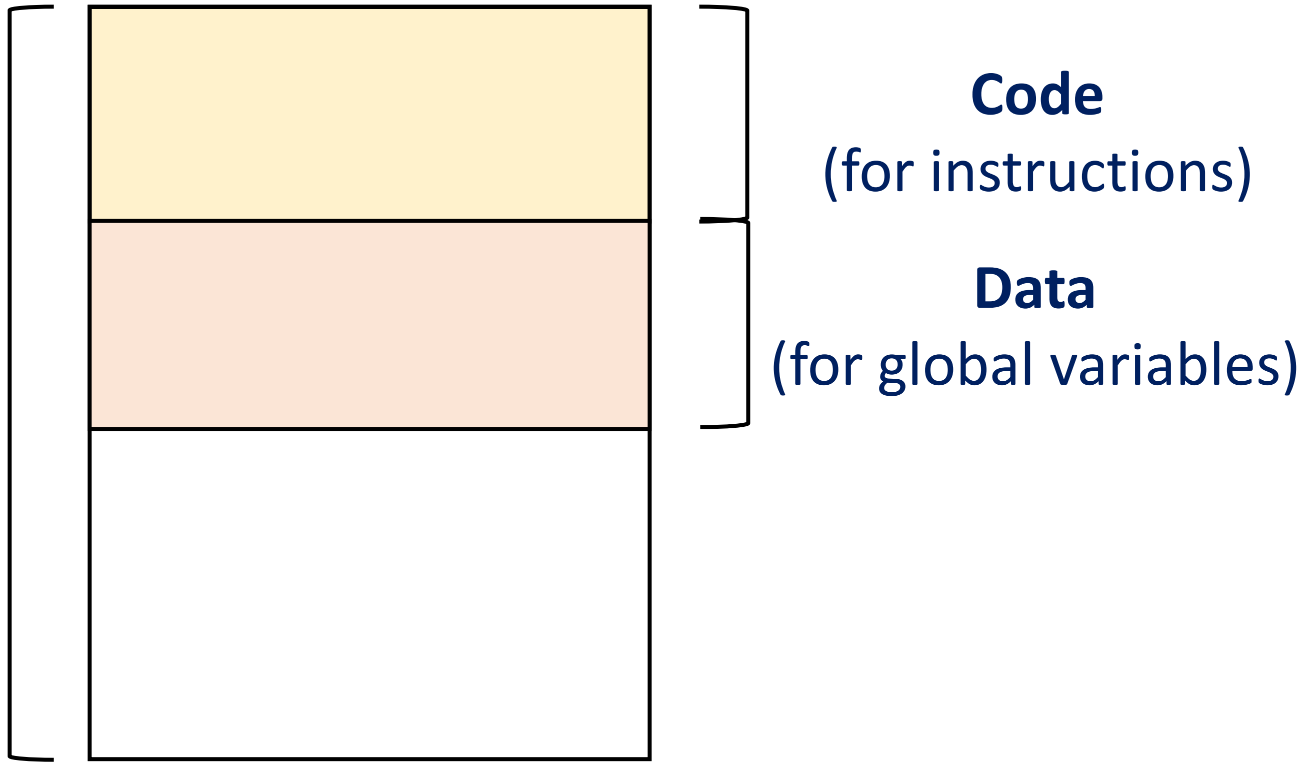
```
i        ...
         0
         ...
inc()    ...
         lw    $1, [4096]
         addi  $1, $1, 20
         sw    $1, [4096]
         ...
main()   ...
         ...
         call inc
         ...
```

Executable

# Executable → Loaded into Memory



## Memory Usage of Process Simplified



How about variables defined in a function?

- e.g. Parameters, Local variables

## Variables in function: Scoping

```
void f()
{
   int x;
   g(123);
   ...
}
void g(int x)
{
   h();
   ...
}
void h()
{
   int x;
}
```

– Consider the example of the left

– The 3 "x" variables are actually different from each other

– Also, these "x" exists **only when** the respective function is executing!

## Stack Memory

### Stack Memory Region

- New memory region (environment) to store data for function calls
- Environment of each function call is known as a stack frame

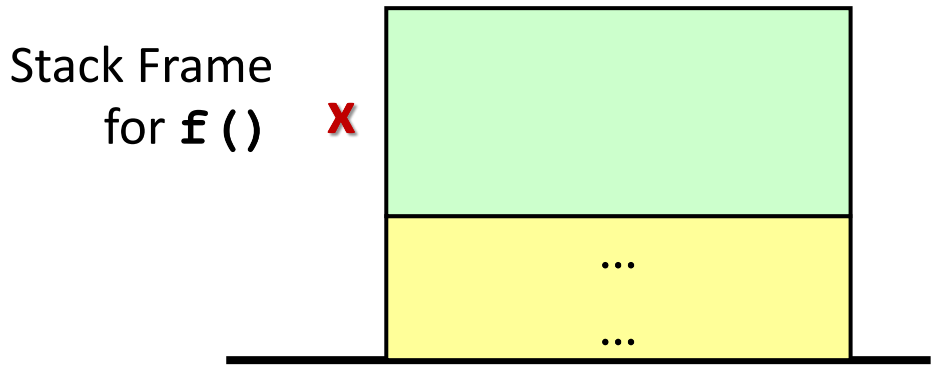
### Stack frame contains:

- Arguments (actual parameters) for the function
- Local variables
- Other information…. (not covered in this course)

## Illustration: **Stack Memory Usage** (1 / 5)

```
void f()
{
    int x;
    g(123);
    ...
}

void g(int x)
{
    h();
    ...
}

void h()
{
    int x;
}
```

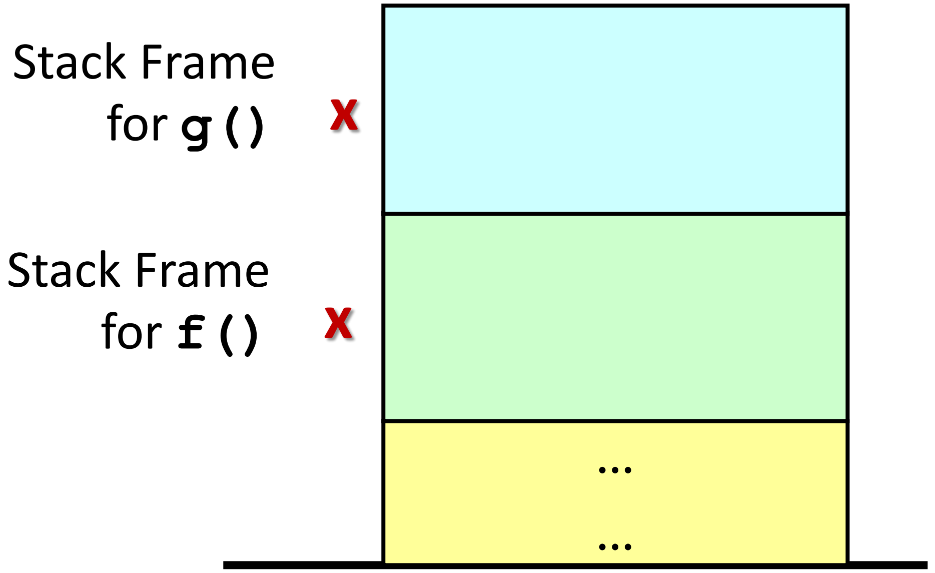
At this point



# Illustration: **Stack Memory Usage** (2 / 5)

```
void f()
{
    int x;
    g(123);
    ...
}

void g(int x)
{
    h();
    ...
}

void h()
{
    int x;
}
```

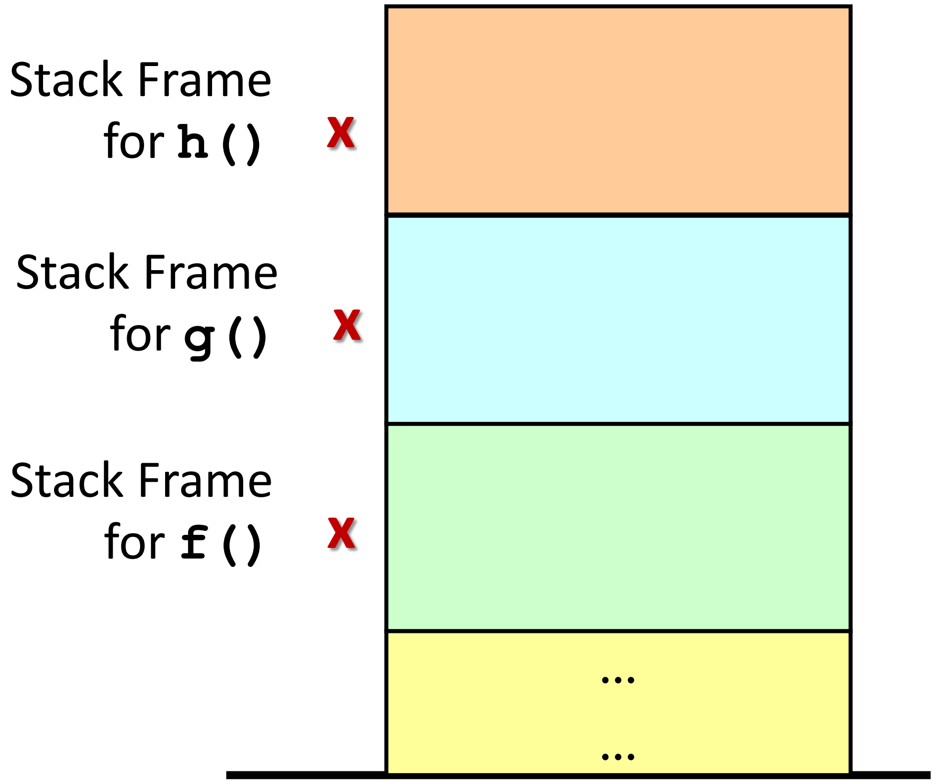
At this point



# Illustration: **Stack Memory Usage** (3 / 5)

```
void f()
{
    int x;
    g(123);
    ...
}

void g(int x)
{
    h();
    ...
}

void h()
{
    int x;
}
```

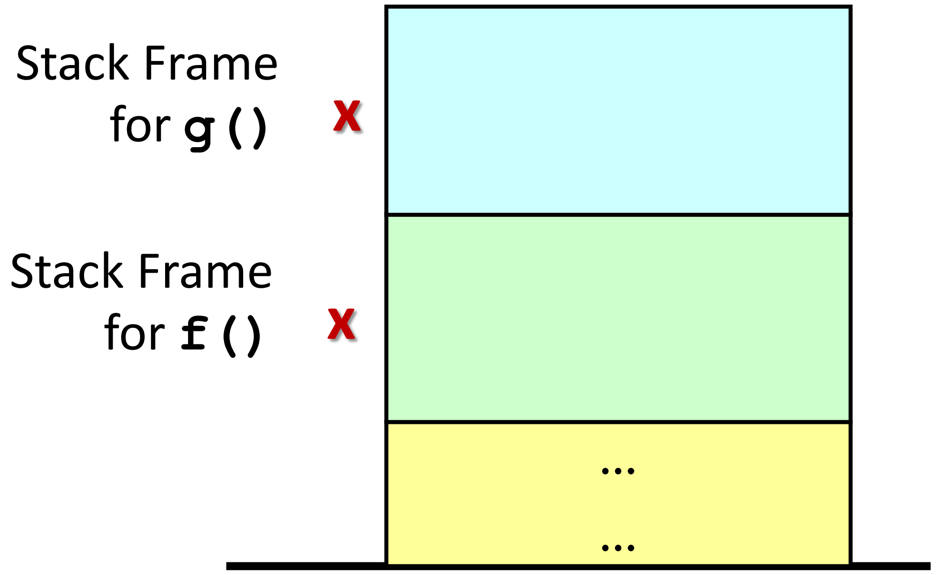
At this point



## Illustration: **Stack Memory Usage** (4 / 5)

```
void f()
{
    int x;
    g(123);
    ...
}

void g(int x)
{
    h();
    ...
}

void h()
{
    int x;
}
```

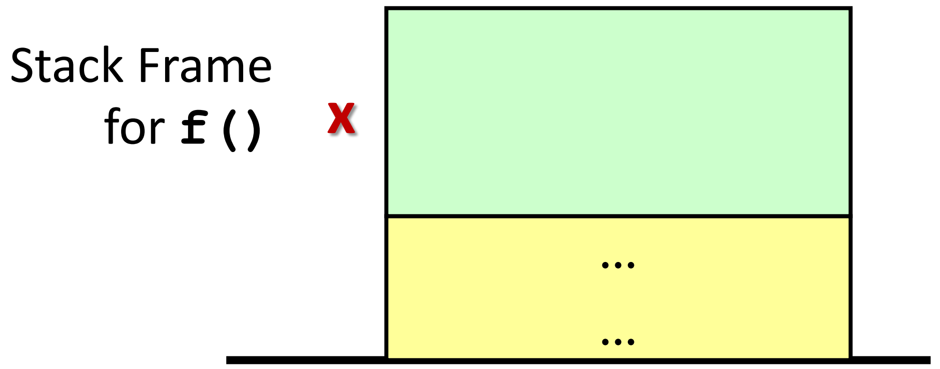
At this point



# Illustration: **Stack Memory Usage** (5 / 5)

```
void f()
{
    int x;
    g(123);
    ...
}

void g(int x)
{
    h();
    ...
}

void h()
{
    int x;
}
```

At this point



## Stack Memory: Key Ideas

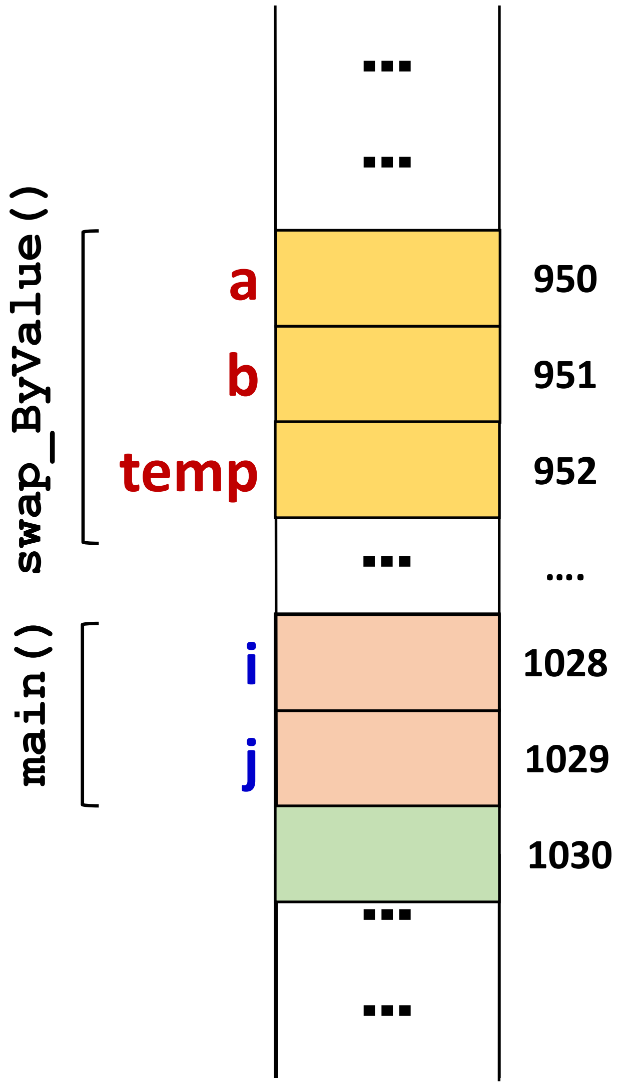
Stack frames are created / destroyed at runtime:

- Entering a function: new stack frame
- Exiting a function: stack frame destroyed

Hence, we can have multiple variables with the same name in different function (and scope)

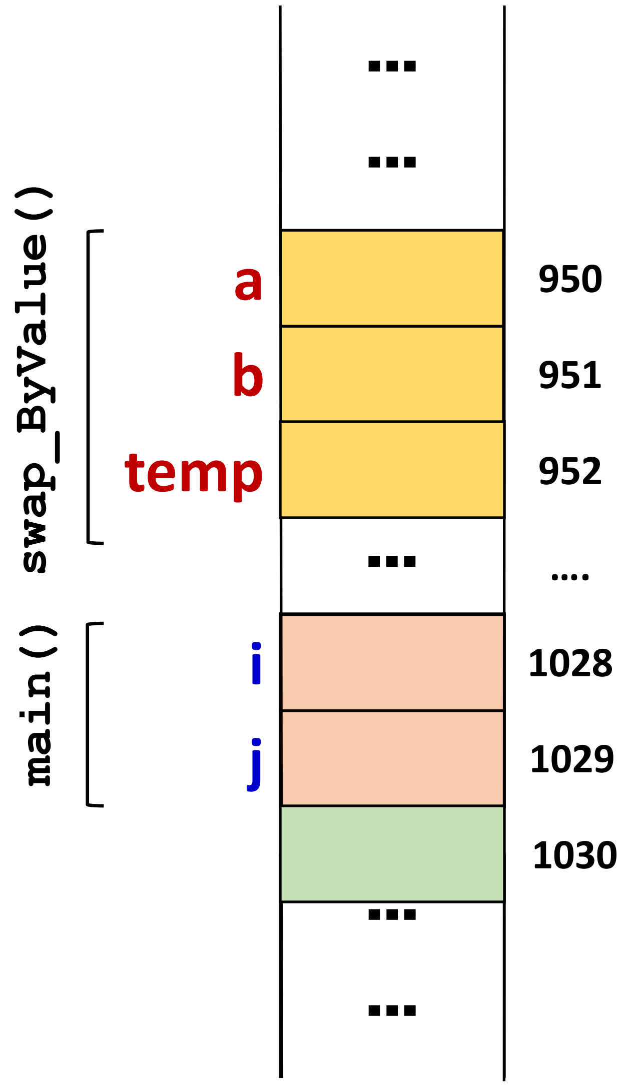
# Function : Pass by value

```
void swap_ByValue( int a, int b )
{    int temp;

     temp = a;
     a = b;
     b = temp;
}

int main()
{
     int i = 123, j = 456;

     swap_ByValue( i, j );

     cout << i << endl;
     cout << j << endl;
}
```



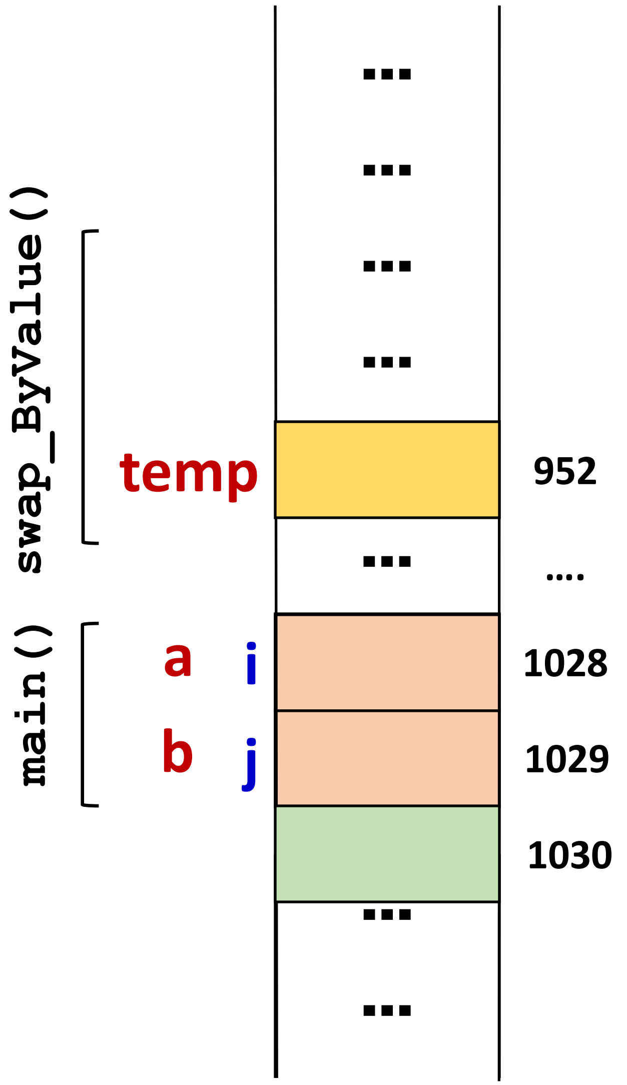
# Function : Pass by address/pointer

```
void swap_ByAdr( int* a, int* b )
{   int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int i = 123, j = 456;

    swap_ByAdr( &i, &j );

    cout << i << endl;
    cout << j << endl;
}
```

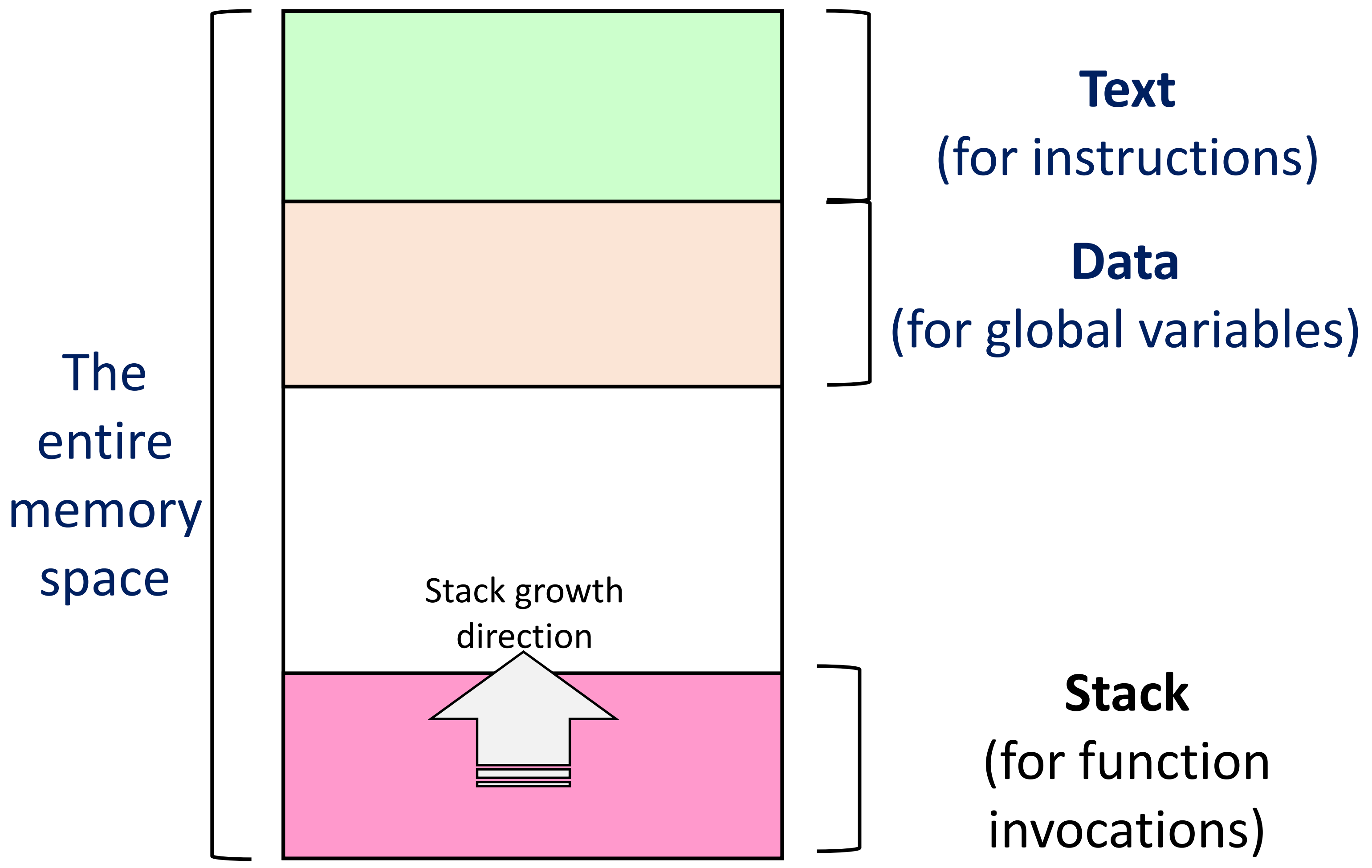


# Function : Pass by reference

```
void swap_ByRef( int& a, int& b )
{    int temp;

     temp = a;
     a = b;
     b = temp;
}

int main()
{
     int i = 123, j = 456;

     swap_ByRef( i, j );

     cout << i << endl;
     cout << j << endl;
}
```



# Memory Usage of Process Updated



## Memory Hardware

# Volatile and Non-Volatile Memory

Volatile Memory (main focus):

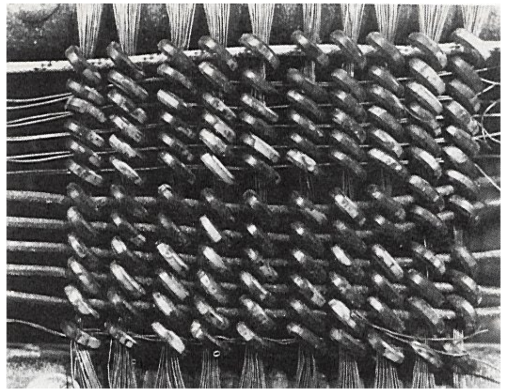
- Storage that loses information when electrical power is interrupted
- e.g. RAM (Random Access Memory)
- Main storage of Process Memory discussed

Non-Volatile Memory

- Storage that can maintain information even without power
- e.g. ROM (Read-only memory), Harddisk, SSD (Solid State Drive), etc.
- Commonly store only persistent information, e.g. file

# Memory Technology : 1950s

**1948: Maurice Wilkes examining EDSAC's delay line memory tubes 16-tubes each storing 32 17-bit words**



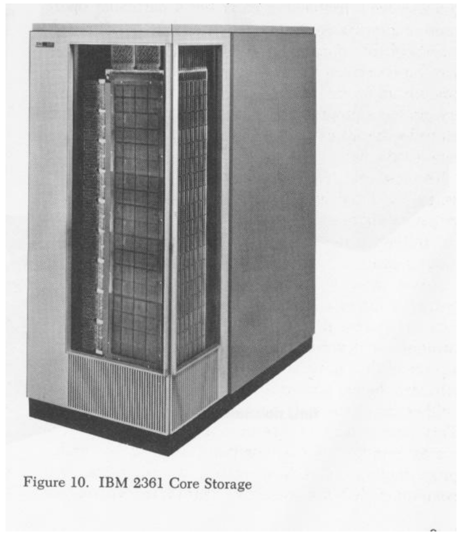


Figure 10. IBM 2361 Core Storage

**1952: IBM 2361 16KB magnetic core memory**

**Maurice Wilkes: 2005**

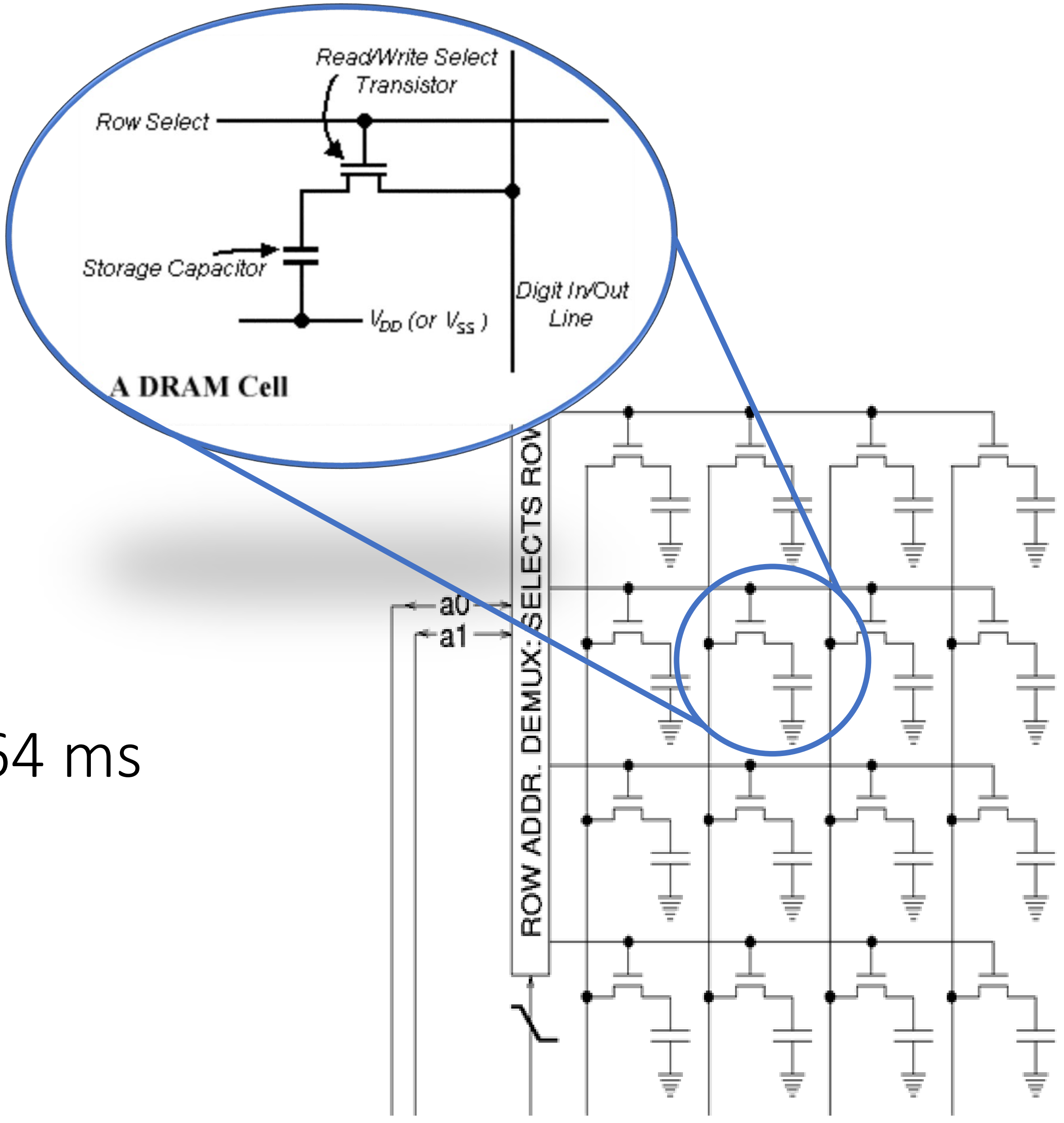
# Dynamic RAM

## High density

- 1 transistor + 1 capacitor per memory cell
- Cheap!

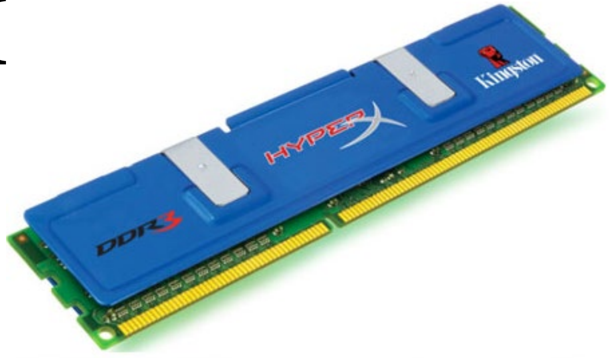
## Slow access

- Latency of 50-70 ns
- Require periodic refresh every 64 ms
  - Or each cell every 7.8 µs
- Memory blocked for 75-120 ns



16 DRAM cells, i.e. 16bits

# Memory Technology Today: SDRAM

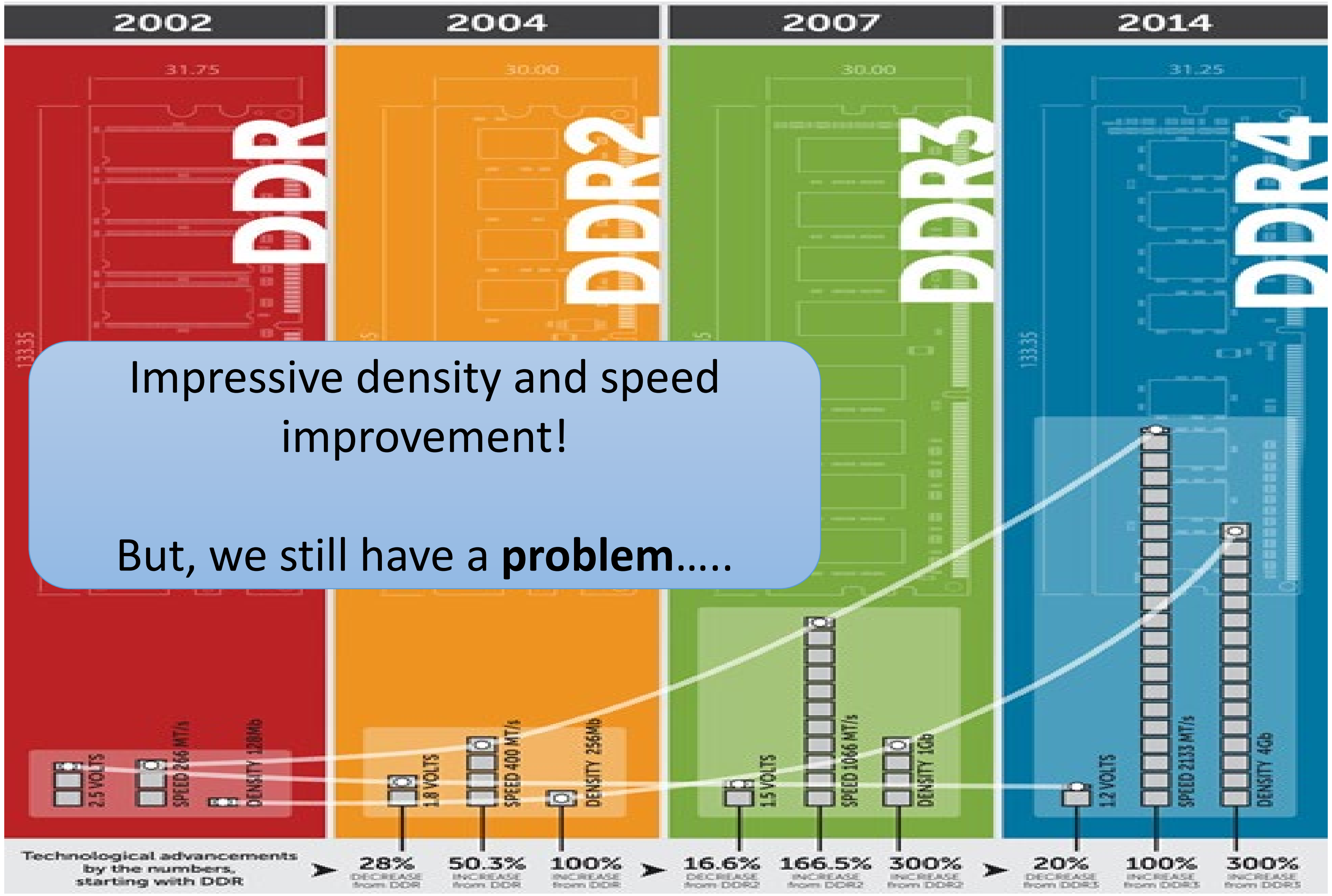


## DDR SDRAM

– Double Data Rate Synchronous Dynamic RAM
– The dominant memory technology in PC market
– Delivers memory on the positive and negative edge of a clock (double rate)

| DDR SDRAM Standard | Internal rate (MHz) | Bus clock (MHz) | Prefetch | Data rate (MT/s) | Transfer rate (GB/s) | Voltage (V) |
|---|---|---|---|---|---|---|
| SDRAM | 100-166 | 100-166 | 1n | 100-166 | 0.8-1.3 | 3.3 |
| DDR | 133-200 | 133-200 | 2n | 266-400 | 2.1-3.2 | 2.5/2.6 |
| DDR2 | 133-200 | 266-400 | 4n | 533-800 | 4.2-6.4 | 1.8 |
| DDR3 | 133-200 | 533-800 | 8n | 1066-1600 | 8.5-14.9 | 1.35/1.5 |
| DDR4 | 133-200 | 1066-1600 | 8n | 2133-3200 | 17-21.3 | 1.2 |

# Snapshot: DDR RAM



Impressive density and speed improvement!

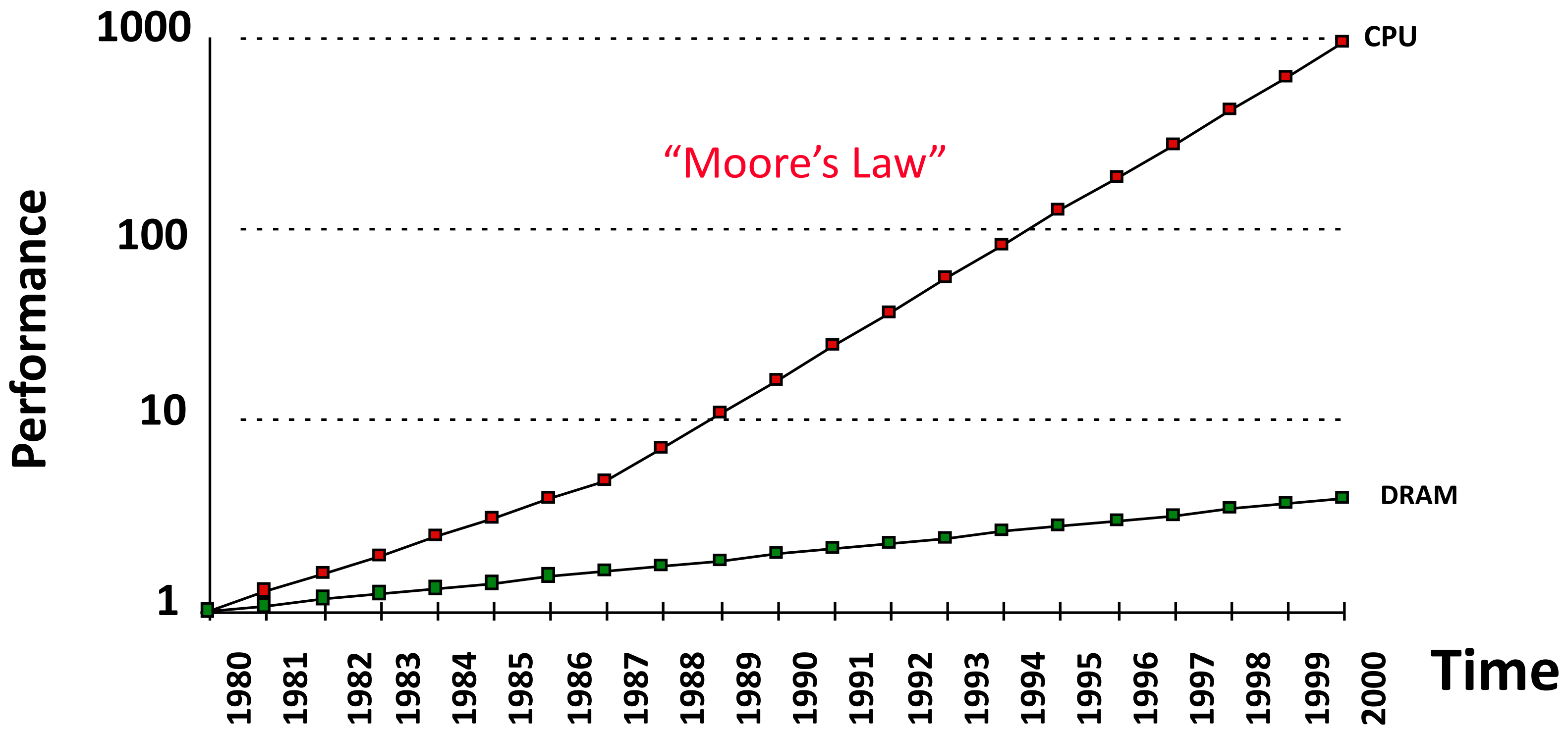
But, we still have a **problem**.....

Source: http://www.crucial.com/usa/en/memory-ddr4-info

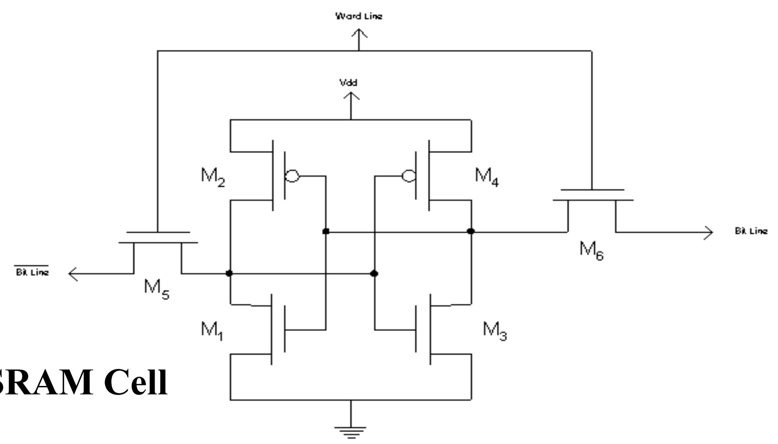
# Processor-DRAM Performance Gap

## Memory Wall:

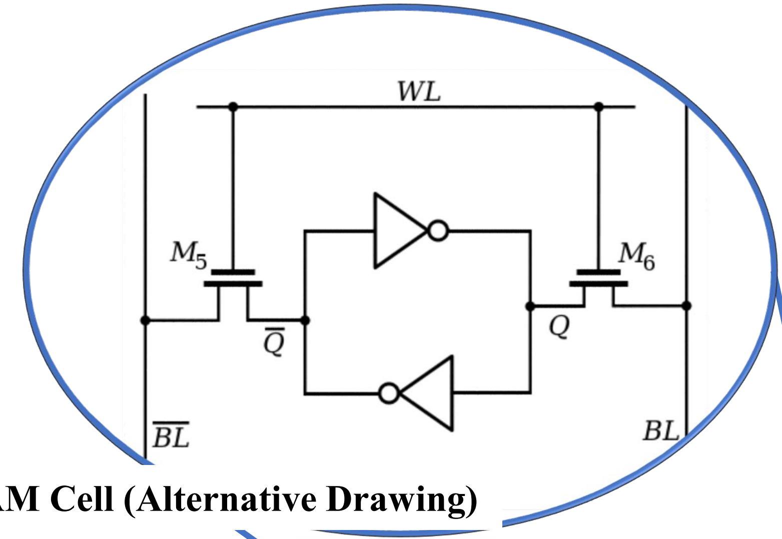
- 1 GHz Processor → 1 ns per clock cycle
- 50 ns for DRAM access → 50 processor cycles per memory access!!



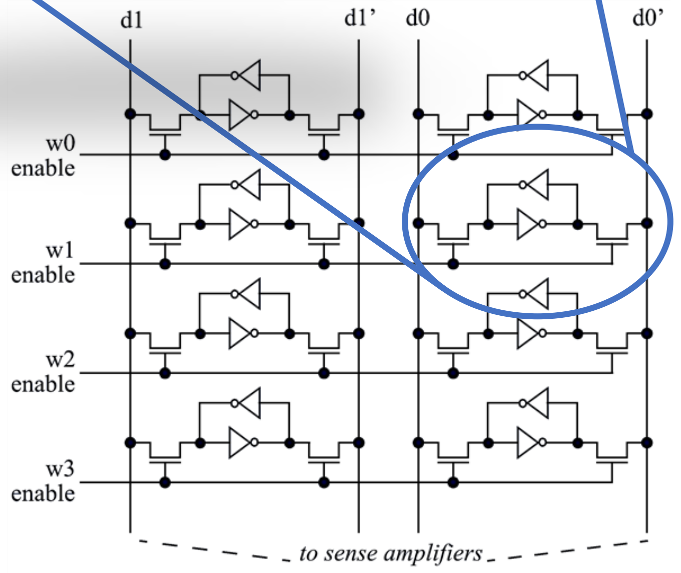
# Faster Memory Technology: Static RAM



A SRAM Cell



A SRAM Cell (Alternative Drawing)



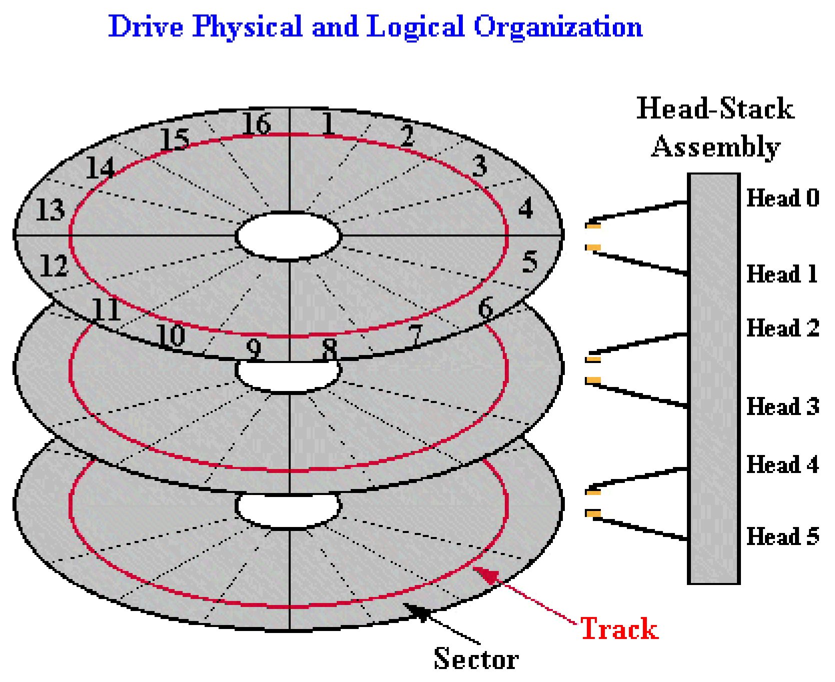
8 SRAM Cells, i.e 8 bits

- Low density
  - 6 transistors (2 inverters) per memory cell
  - Expensive
- Fast access
  - Latency of 0.5-5 ns
  - No refresh needed

# Slow Memory Technologies: Magnetic Disk

Typical high-end hard disk:

- Average Latency: 4 - 10 ms
- Capacity: 500-5,000GB



# Slow Memory Technologies: Flash Memory

## Used in

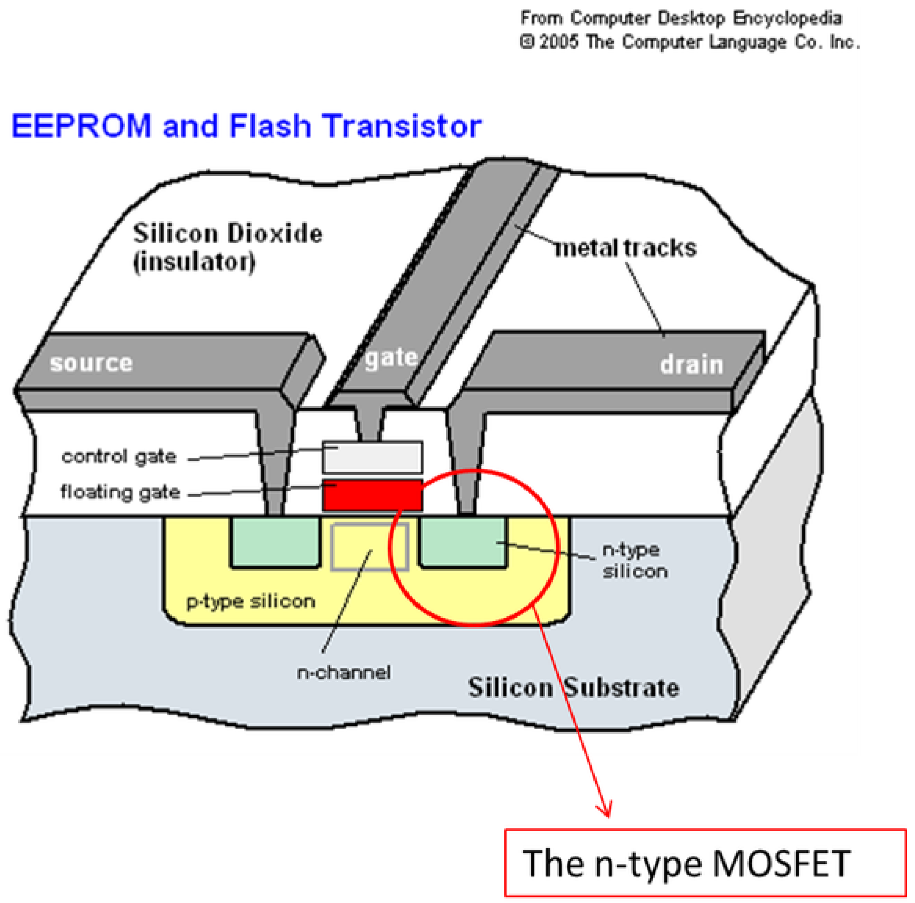
- Thumbdrives
- Memory Cards
- Solid State Drives (SSD)

## Specially designed transistors

- Trap electrons between gates

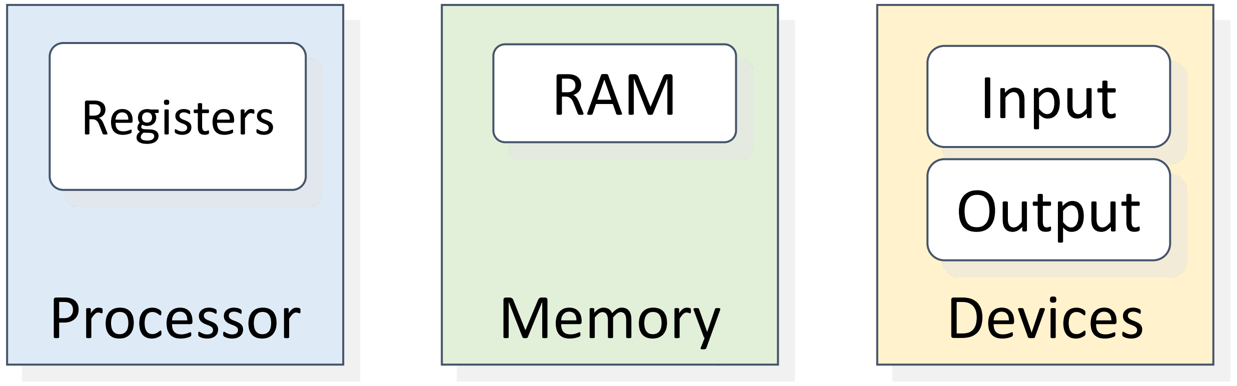
## High-end SSD

- Latency of 50 µs


From Computer Desktop Encyclopedia
© 2005 The Computer Language Co. Inc.
EEPROM and Flash Transistor
Silicon Dioxide (insulator)
metal tracks
source
gate
drain
control gate
floating gate
n-type silicon
p-type silicon
n-channel
Silicon Substrate
The n-type MOSFET

*Credit: Original picture from http://encyclopedia2.thefreedictionary.com/EEPROM*

# Quality vs Quantity



| | Capacity | Latency | Cost/GB |
|---|---|---|---|
| Register | 100s Bytes | 20 ps | $$$$ |
| SRAM | 100s KB | 0.5-5 ns | $400/GB |
| DRAM | 10s GB | 50-70 ns | $8/GB |
| Hard Disk | 1000s GB | 5-20 ms | $0.025/GB |
| **Ideal** | **1 TB** | **1 ns** | **Cheap** |

## Best of Both Worlds

### What we want

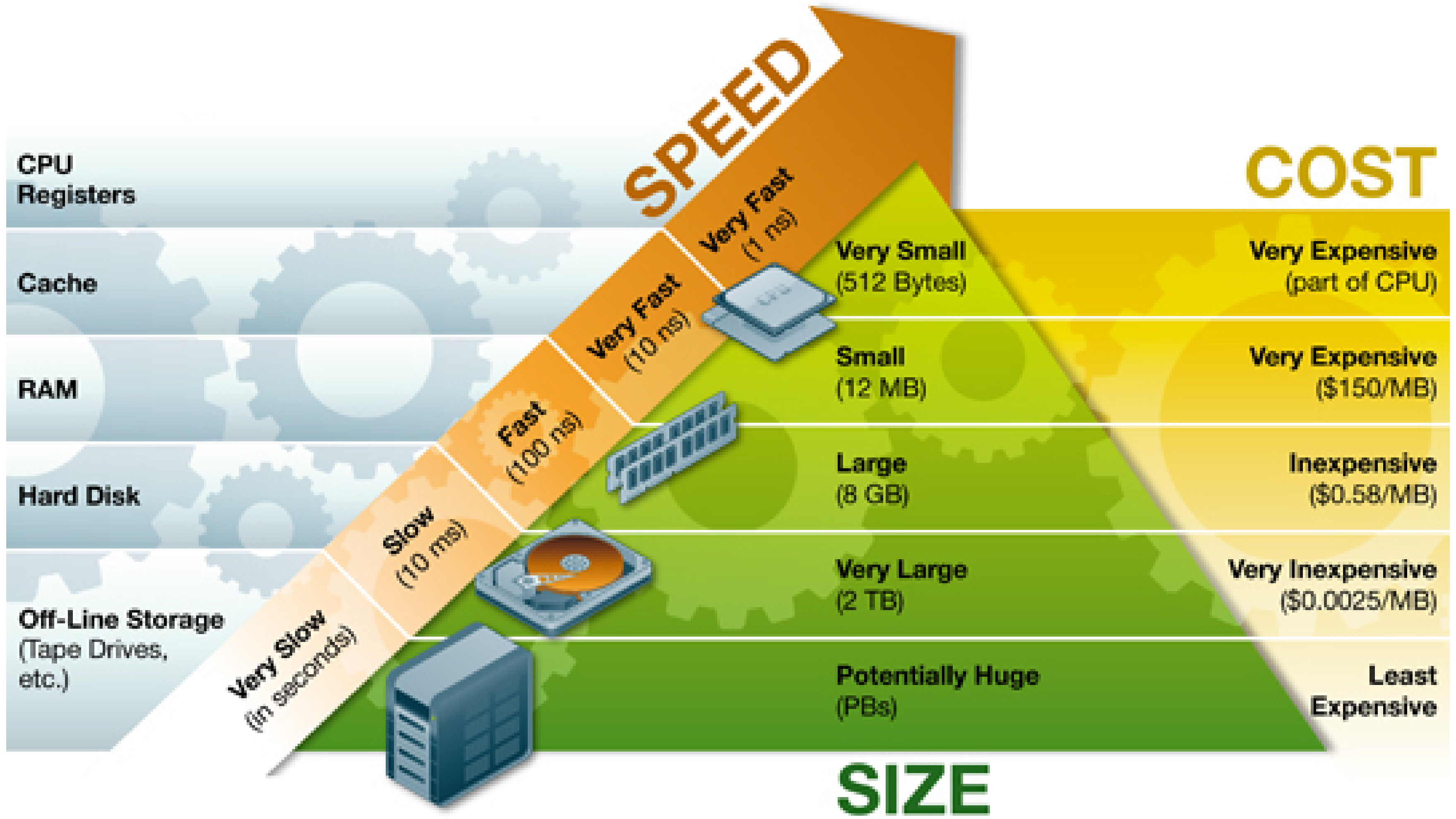
- A BIG and FAST memory
- Memory system should perform like 1GB of SRAM (1 ns access time) but cost like 1GB of slow memory

### Key Idea

### Use a hierarchy of memory technologies

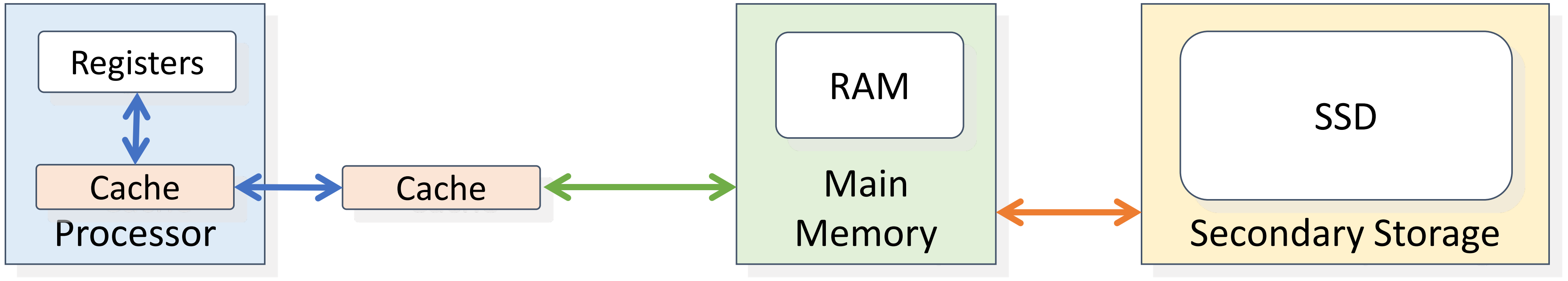
- Small but fast memory near CPU
- Large but slow memory farther away from CPU

# The Memory Hierarchy



Source: http://www.as.techdata.eu/uk/products_and_solutions/storage/hierarchy.html/

# Two Aspects of Memory Access



make SLOW main memory appear **faster**?

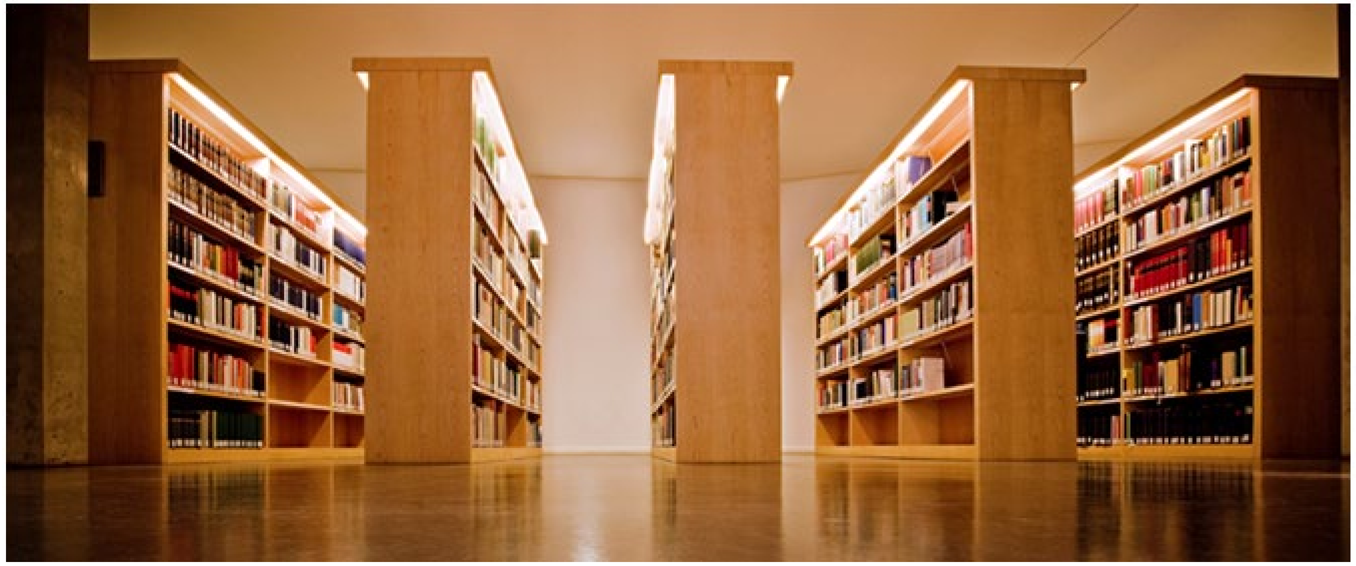
- Cache: a small but fast SRAM near the CPU
- Hardware managed: Transparent to programmer

make SMALL main memory appear **bigger**?

- Virtual memory
- OS managed: Transparent to programmer

## Cache

## The Library Analogy



Imagine you are forced to put back a book to its bookshelf before taking another book.......

## Solution: Book on the Table!

What if you are allowed to take the books that are **needed right now** with you and place them nearby on the desk?

## Cache: The Basic Idea

Keep the frequently and recently used data

- in smaller but faster memory

Refer to bigger and slower memory:

- Only when you cannot find data/instruction in the faster memory

Why does it work?

**Principle of Locality**

Program accesses only a small portion of the memory address space within a small time interval

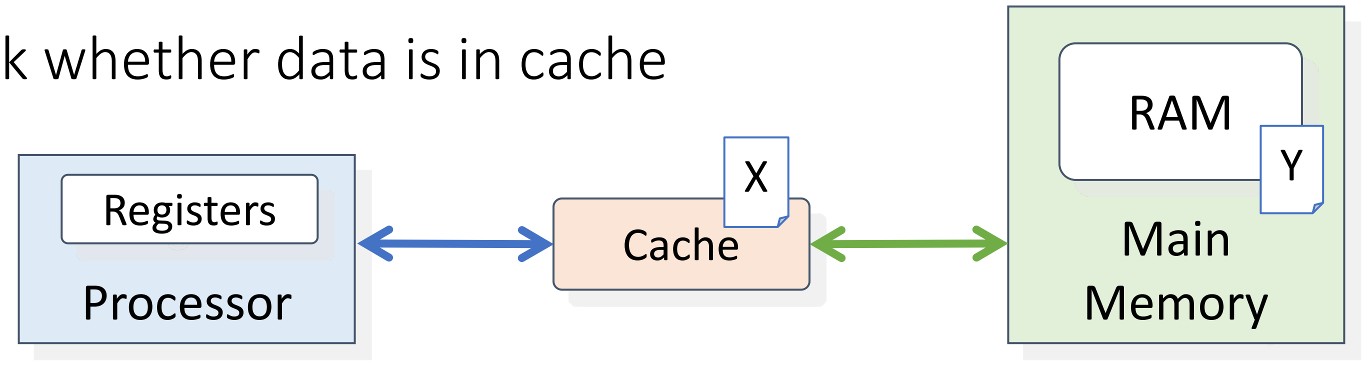
## Types of Locality

```
void cumul_sum(int arr[], int size) {
    for (int i = 1; i < size; ++i) {
        arr[i] = arr[i] + arr[i-1];
    }
}

int main(void) {
    int i, sq[SIZE];

    for (i = 0; i < SIZE; i++) {
        sq[i] = i* i;
    }

    cumul_sum(sq, 10);
    return 0;
}
```

- Temporal locality
  - If an item is referenced, it will tend to be referenced again soon
- Spatial locality
  - If an item is referenced, nearby items will tend to be referenced soon

# Cache: Basic Usage and Terminology

Processor check whether data is in cache



Cache Hit: Data is in cache (e.g., X)

- Data is returned to processor from cache
- This is the ideal case as cache is fast

Cache Miss: Data is not in cache (e.g., Y)

- Data is loaded from memory
- [Simplified] Data is placed in cache for future reference
- Will incur the full memory latency (i.e. slow)

# MEMORY ←→ CACHE MAPPING

Where to place the memory block in cache?

# Terminology: Cache Block/Line

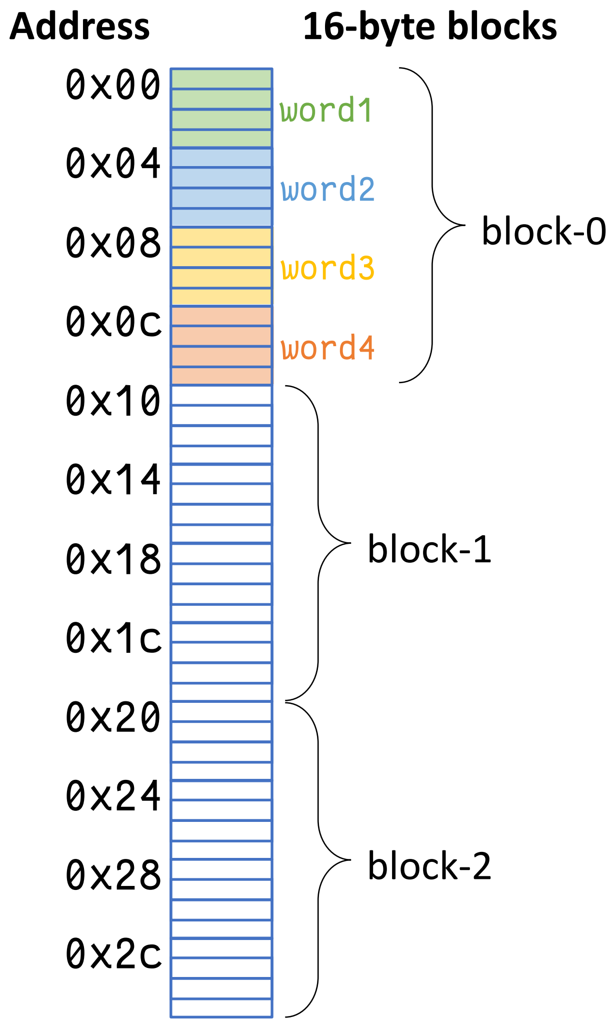
## Cache Block/Line:

- Unit of transfer between memory and cache

## Block size is typically one or more words

- e.g.: 16-byte block ≅ 4-word block
- 32-byte block ≅ 8-word block

## Why block size is bigger than word size?



## Fully Associative Cache

Having total freedom

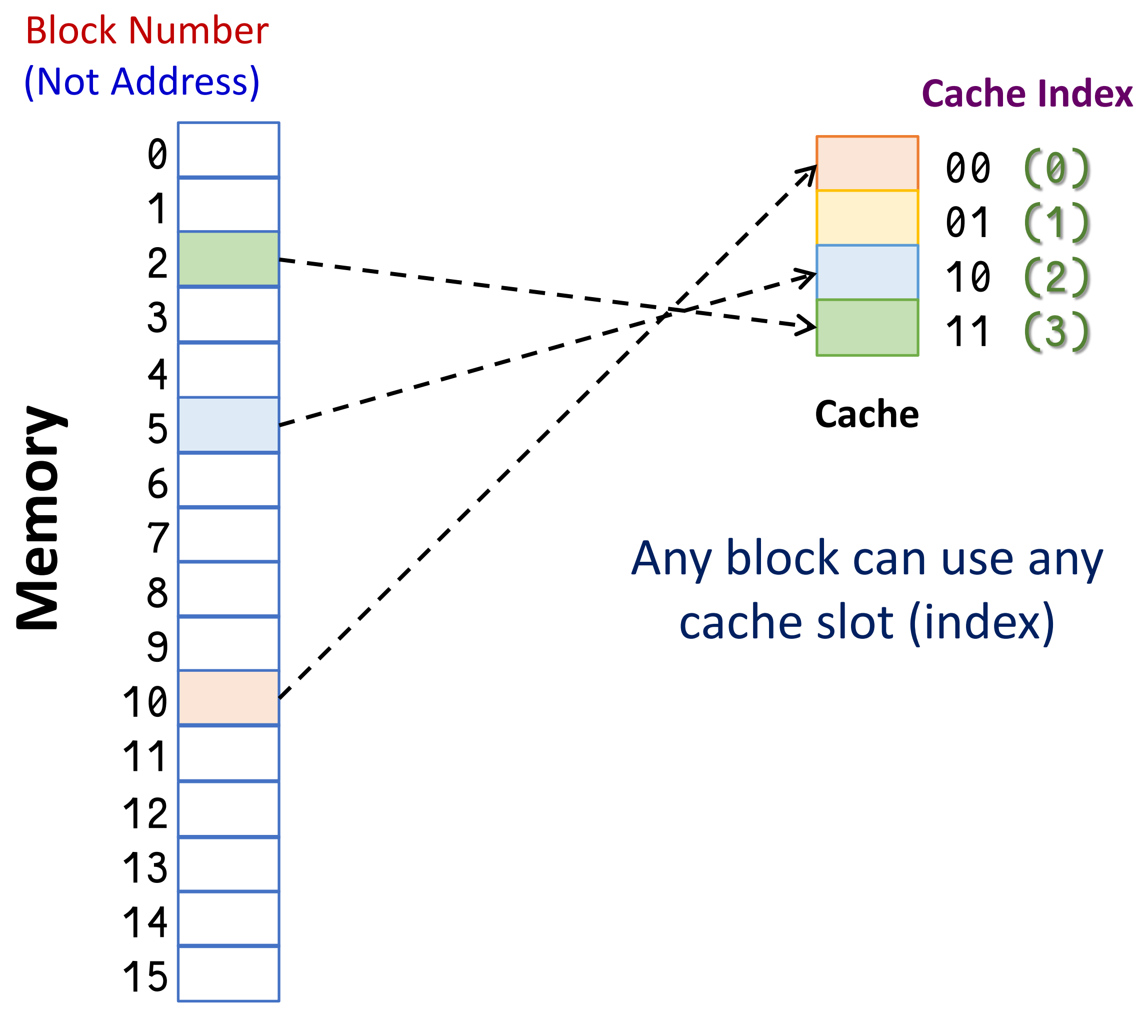
## Fully-**A**ssociative (**FA**) Analogy

A book can go into any location on the desk!

# Fully Associative Cache

Question:

- How to identify which block is in the cache?

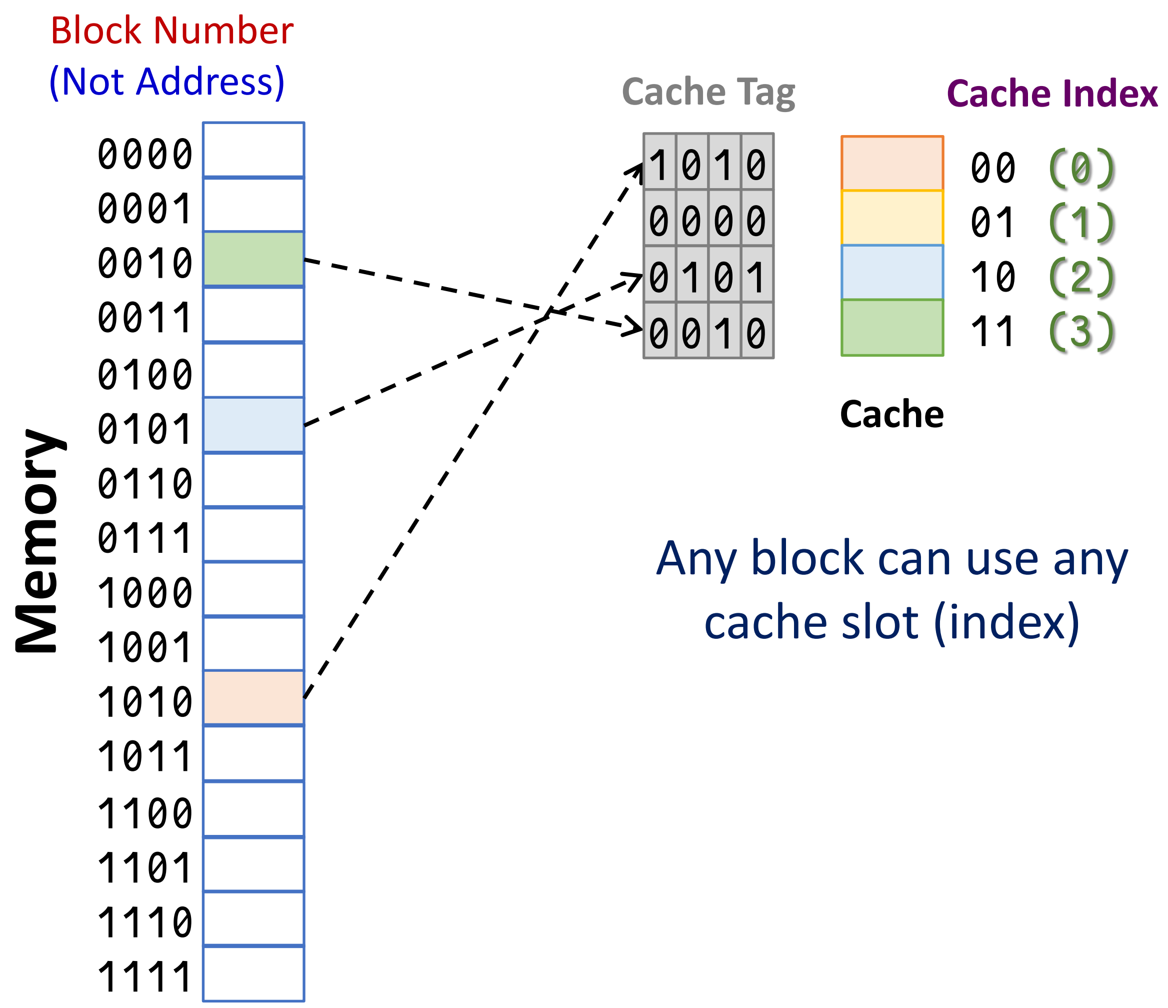


Any block can use any cache slot (index)

# Fully Associative Cache

## Cache Tag

- An identifier (id) to identify the current block in the cache
- Entire block number must be used



## FA Cache: Pros / Cons

Intuitive

- Any block can go into any cache location

Resource intensive

- Need to search all locations at once
- need a lot of hardware comparators
- take up significant space on chip!

➔Fully associative cache is seldom used

➔If used, usually in very limited capacity

## Direct Mapped Cache

Let’s restrict the freedom a little?

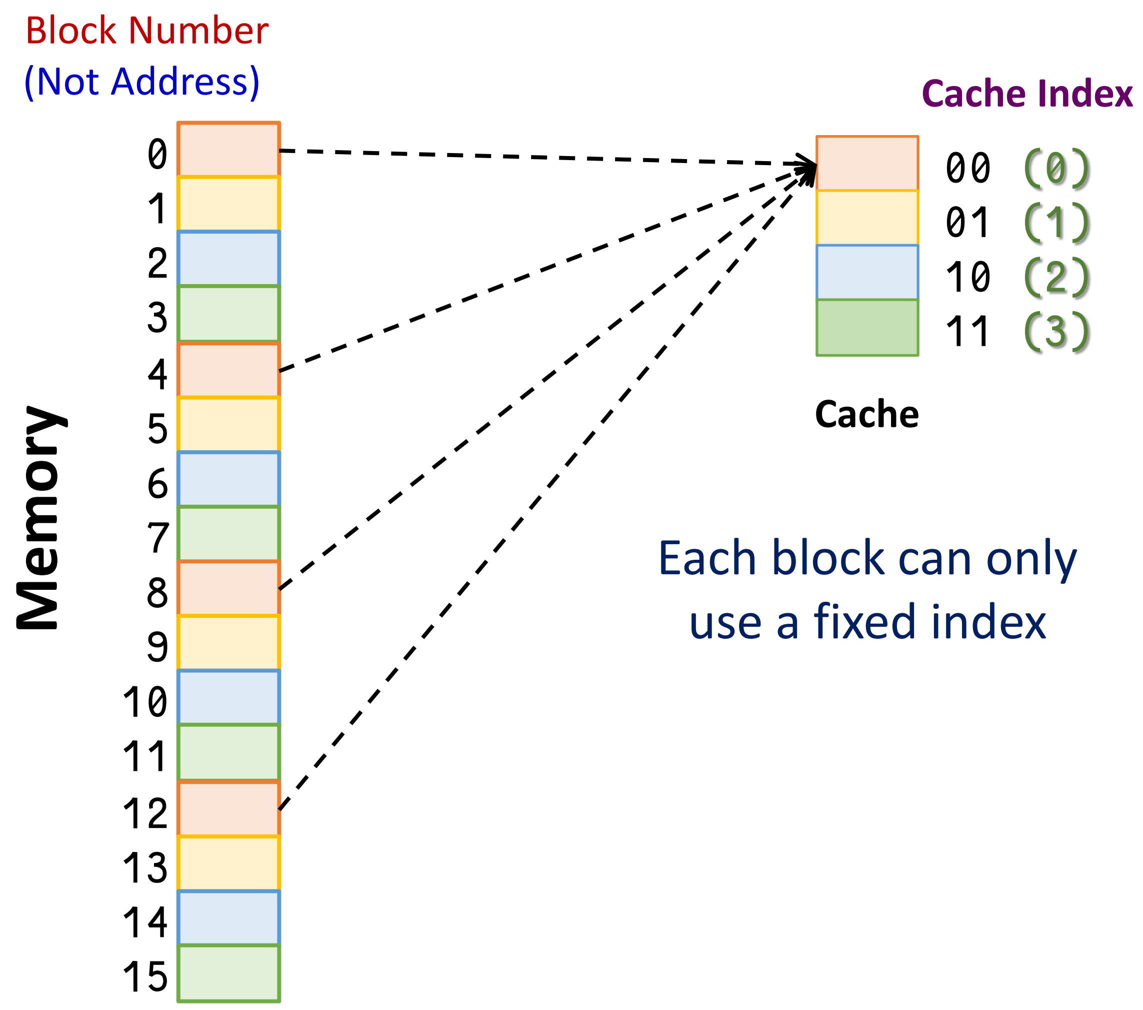
## Direct Mapping (DM) Analogy

Imagine there are 26 "locations" on the desk to store book. A book's location is determined by the first letter of its title.
➔ Each book **has exactly one location**

## Direct Mapped Cache

Question:

– Given a block number, identify the cache index it should go



Each block can only use a fixed index

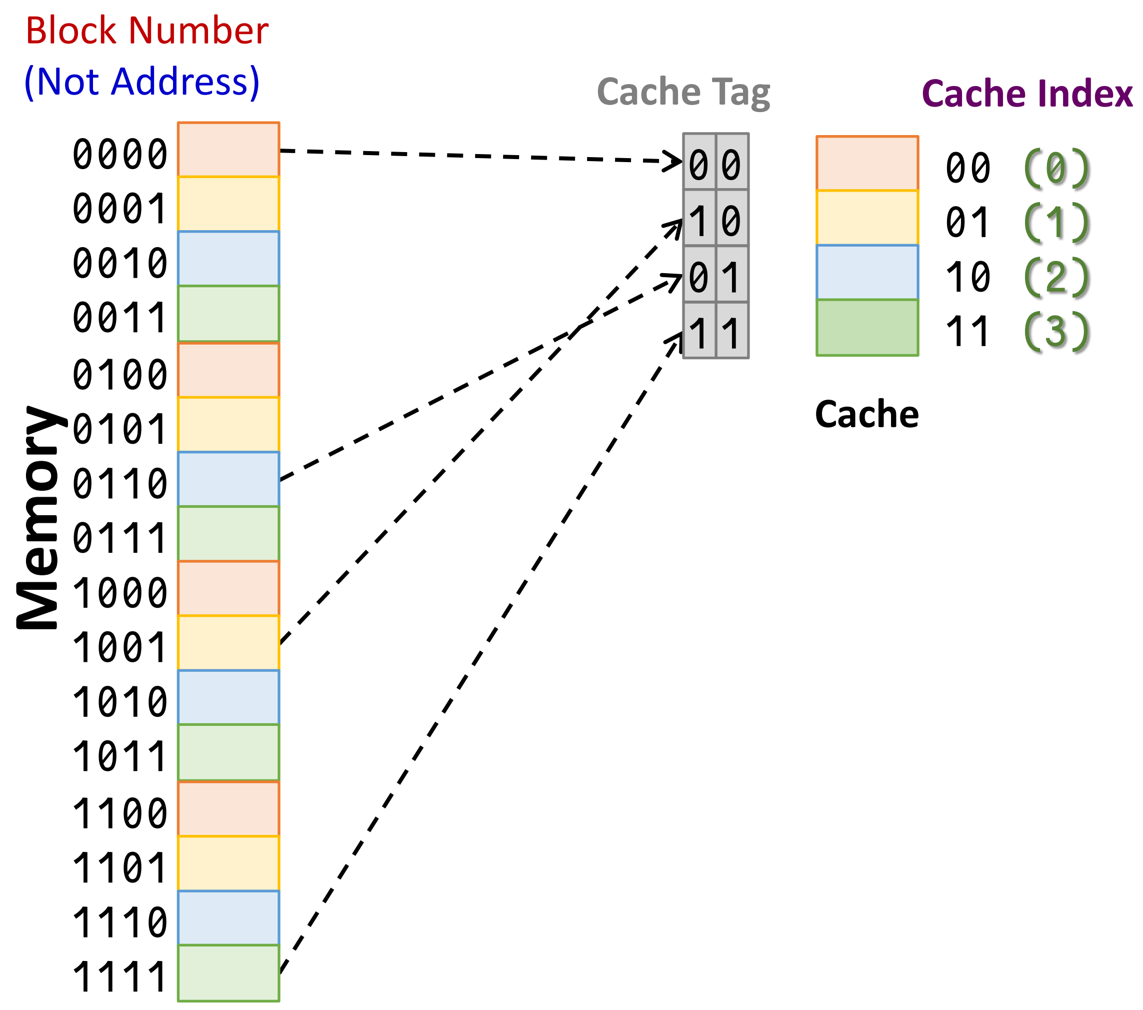
# Direct Mapped Cache

## Question:

- Given a block number, identify the cache index it should go
- How to identify block in cache?

## Cache Tag

- Only requires most significant part of block number



## DM Cache Pros / Cons

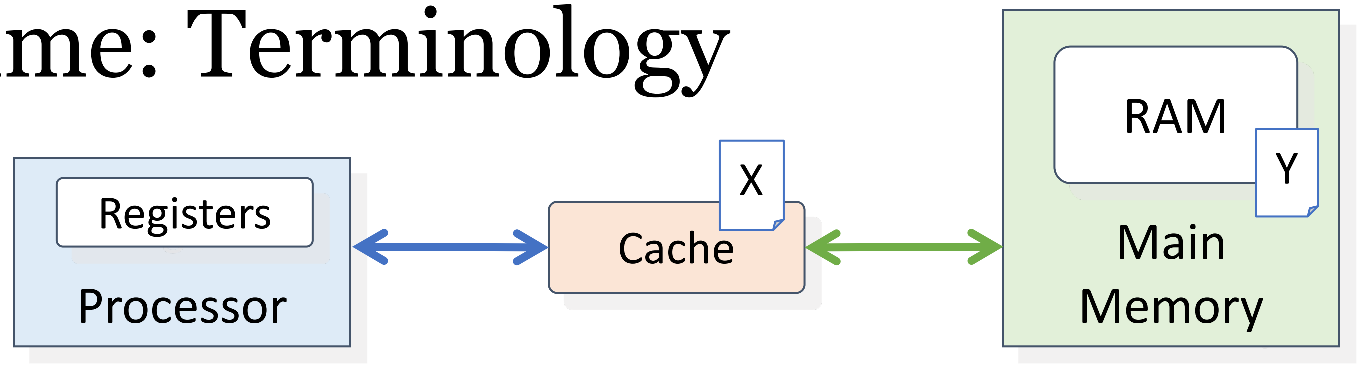
**Better use of hardware resource:**

- Only one possible cache location for any block
- Can have 1 single hardware comparator

**Potential of cache conflicts:**

- What if we happen to use different blocks that maps to the same cache index?

# Memory Access Time: Terminology



Hit: Data is in cache (e.g., X)

- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache

Miss: Data is not in cache (e.g., Y)

- Miss rate = 1 – Hit rate
- Miss penalty: Time to replace cache block + deliver data

Hit time < Miss penalty

## Memory Access Time: **Formula**

Average Access Time

= Hit rate x Hit Time + (1-Hit rate) x Miss penalty

**Example:**

Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**? For simplicity, we assume the miss penalty is the same as the DRAM access time.

## Cache: Exploration

Alternative mapping scheme:

- Set associative (hybrid between FA and DM)

More block replacement algorithm:

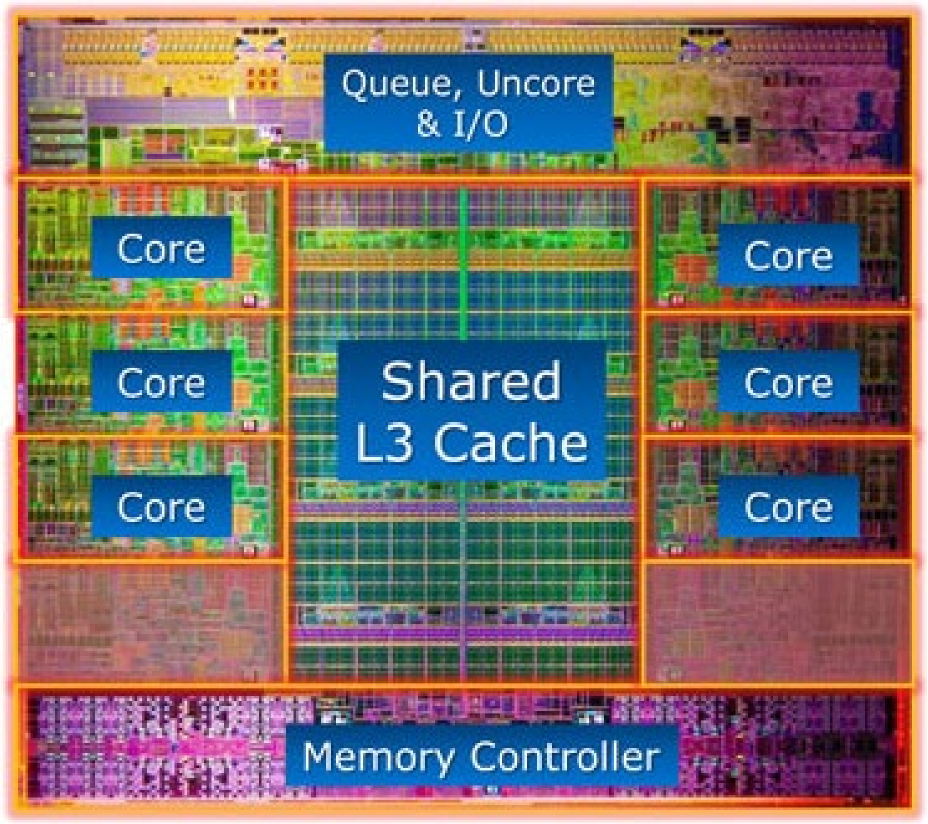
- Which block to replace when the cache is full?

Multi-level cache

We simplified:

- Cache address calculation
- Memory access between a load / store

# Cache Sample: Intel Core i7-3960K



-2.27 billion transistors

-15MB shared Inst/Data Cache (LLC)

**Per Core:**

-32KB L1 Inst Cache

-32KB L1 Data Cache

-256KB L2 Inst/Data Cache

-up to 2.5MB LLC

END