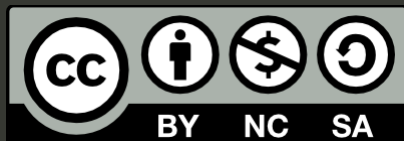


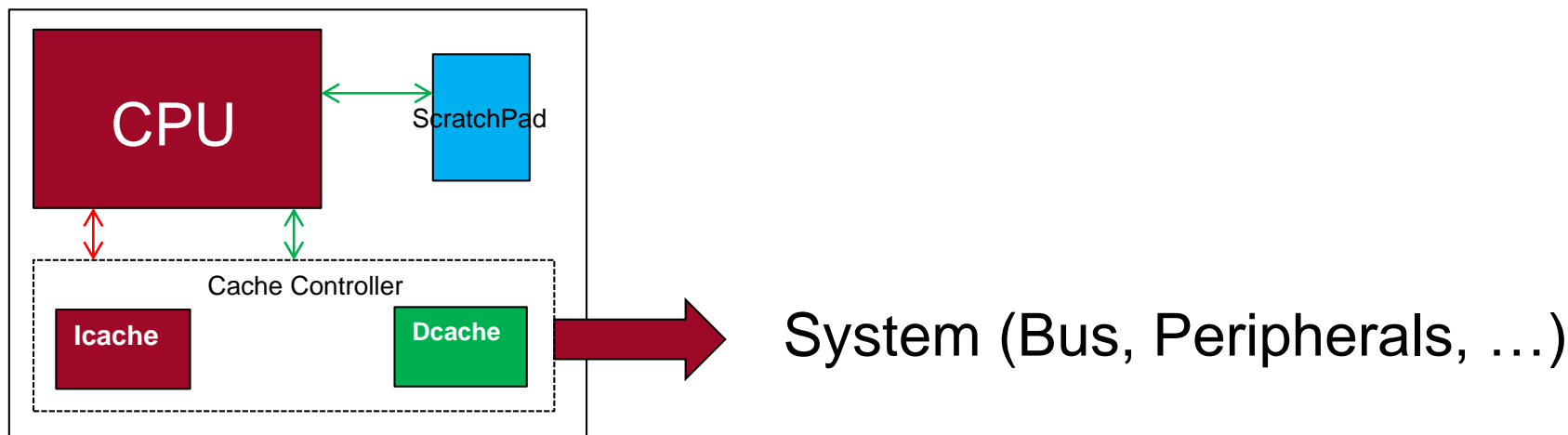
# ENSC254 – Cache and Processor Architectures

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/>. Please share all edits and derivatives with the below authors.

© 2021 -- Fabio Campi and Craig Scratchley  
School of Engineering Science  
Simon Fraser University  
Burnaby, BC, Canada



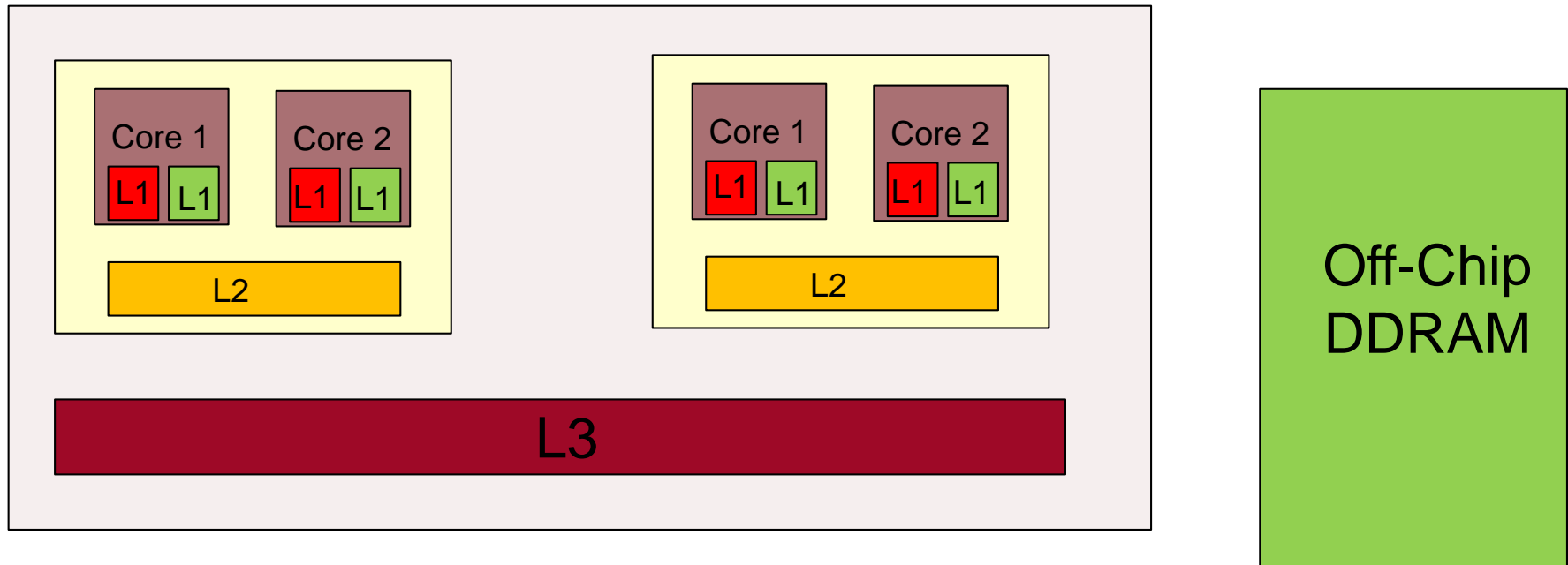
# Cache and Scratchpad memories



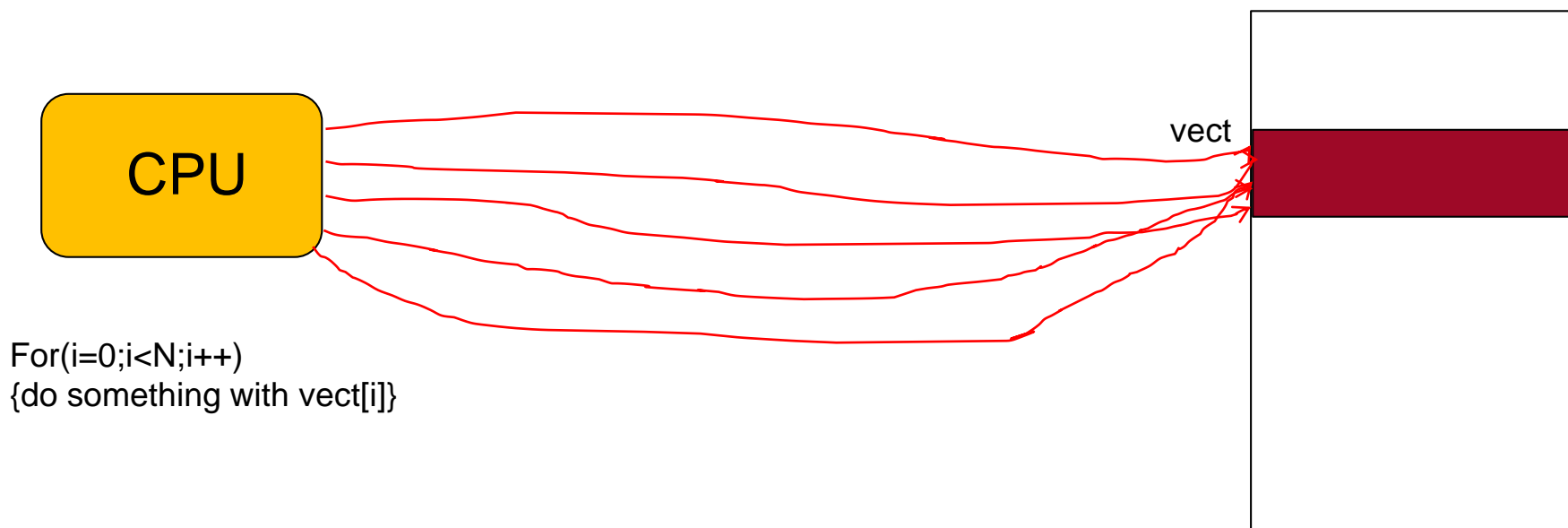
- Often, the CPU benefits from a small local memory to perform critical computation.
  - If this memory is controlled by the CPU itself, it is usually defined **SCRATCHPAD**
  - We can build a system that *automatically* stores locally to the CPU data and instructions that are considered relevant for the CPU itself. In this case, the local memory is called **CACHE**
  - ***For Harvard architectures, it is quite customary to have separate Instruction and Data Caches, both connected to a single memory space***

# Levels of Cache

- Today's CPUs (embedded or not) have multiple layers of cache (Defined L1 cache, L2 cache, ...LN cache).
- In the case of multi-core architectures, more and more common in today's system-on-a-chip ICs, the cache controllers are also a handy way to maintain data consistency between the different cores

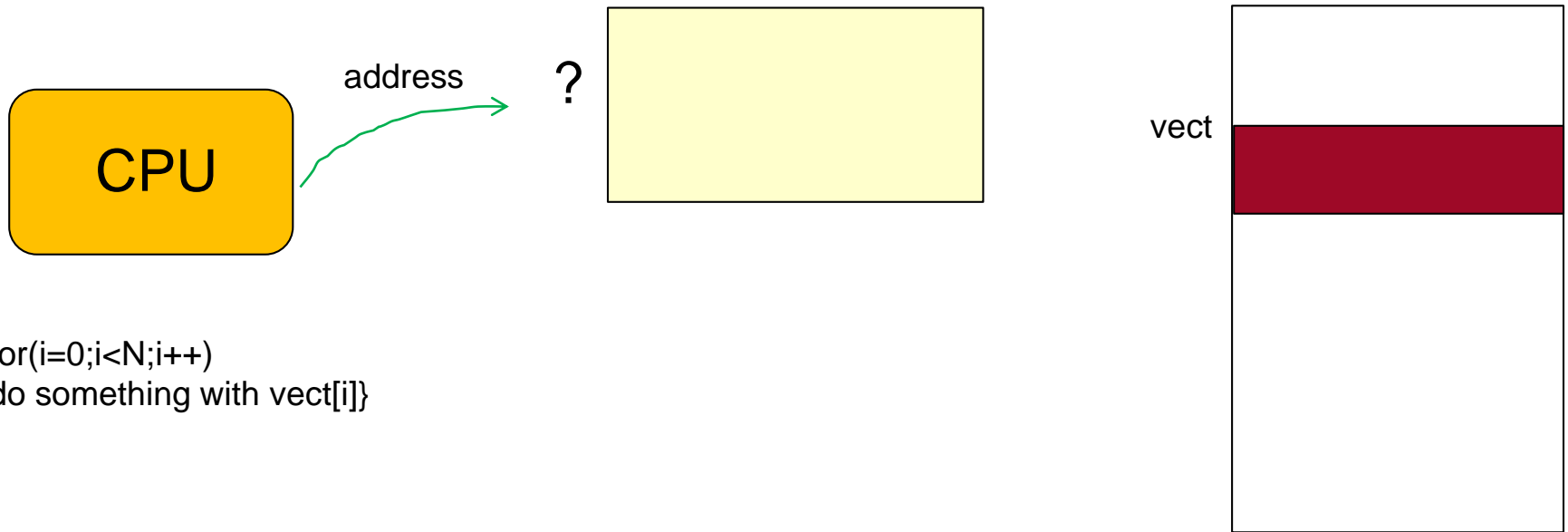


# Cache Memory: Read Operations



- The idea behind a cache memory is to keep a local copy of some portions of the memory to have faster/cheaper access
  - *Most of our computation is organized in loops over large streams of data (sounds, images, antenna samples)*
  - *It is very likely that CPU memory accesses are **localized** (i.e. if I am addressing IMEM or DMEM at 0x40040, I will probably perform my next access in 0x40044 or somewhere around that)*

# Cache Memory: Cache MISS



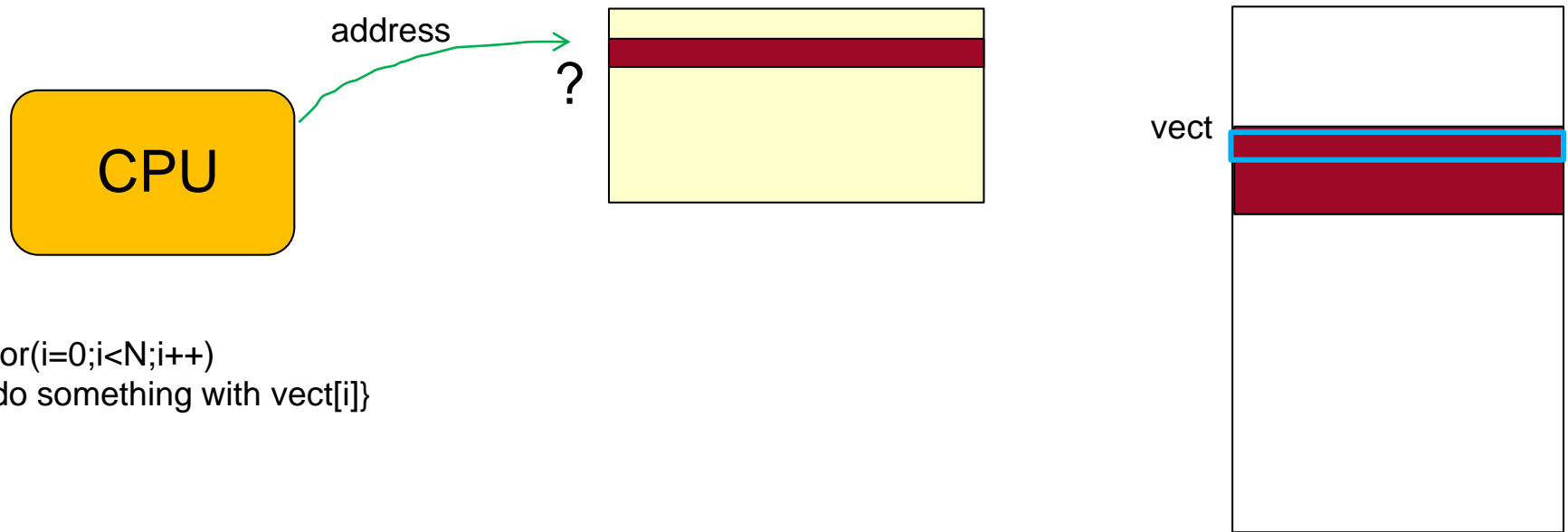
- Whenever we have a memory access, we check in the cache first:
  - IF THAT ADDRESS IS NOT IN THE CACHE, We have a “CACHE MISS”

# Cache Memory: Cache MISS



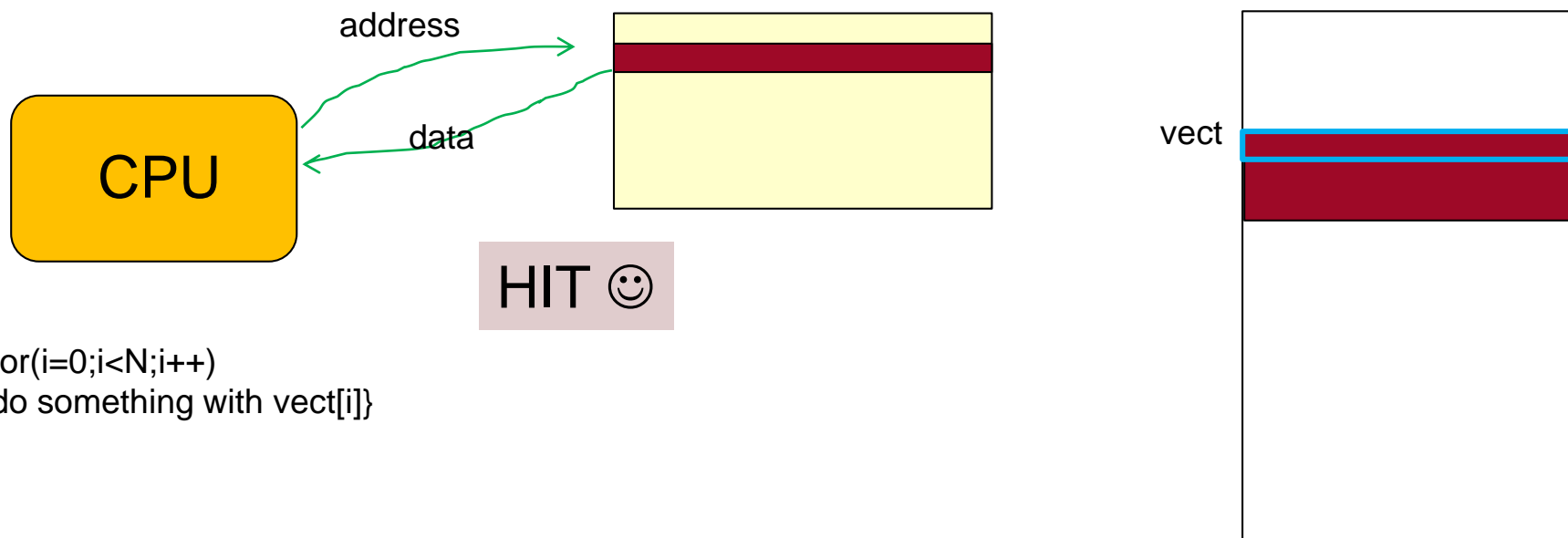
- Whenever we have a memory access, we check in the cache first:
  - IF THAT ADDRESS IS NOT IN THE CACHE, We have a “CACHE MISS”
  - As a consequence, WE COPY A FULL “LINE” from main memory to the Cache
  - Line=Set of memory words (8,16,32,64, ....)

# Cache Memory: Cache MISS



- Whenever we have a memory access, we check in the cache first:
  - IF THAT ADDRESS IS IN THE CACHE, We have a “CACHE HIT”

# Cache Memory: Cache MISS

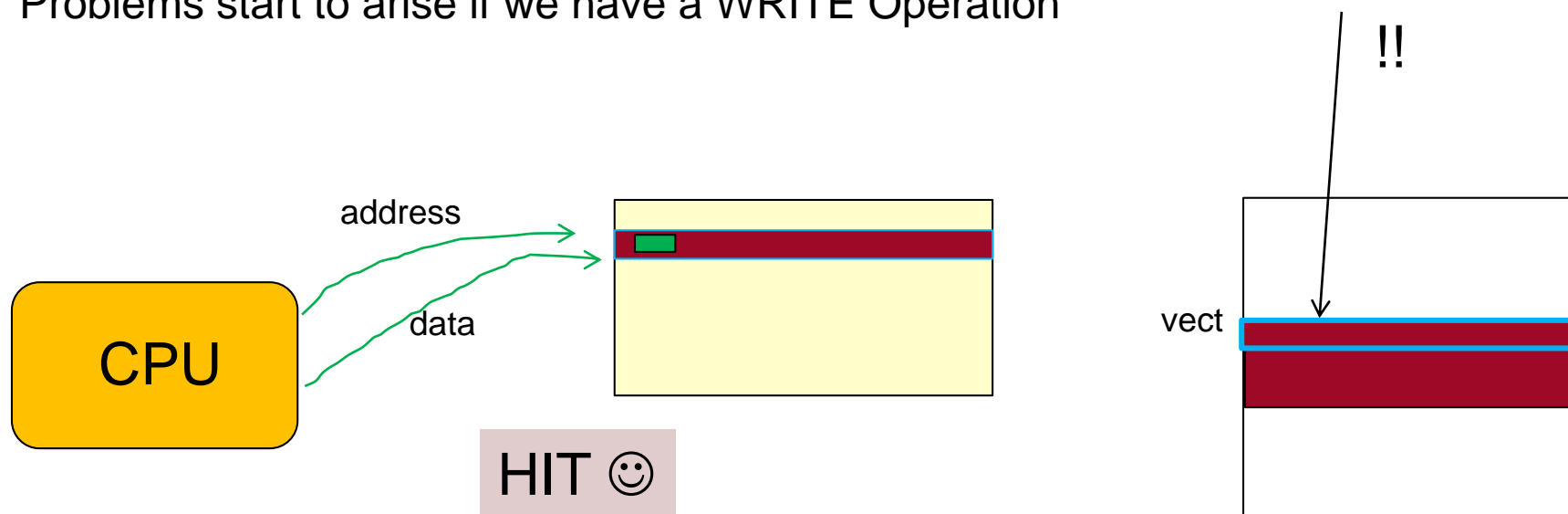


- Whenever we have a memory access, we check in the cache first:
  - IF THAT ADDRESS IS IN THE CACHE, we have a “CACHE HIT”.
  - In that case we can load data from cache memory, without the need of going to the main memory, and saving a lot of cycles and energy.
  - **HIT Rate** = % of Hits over memory accesses. Ideally we want it ~100% but hit rate depends on the cache size, cache policies and the algorithm being run



# Cache Memory: Write Operations

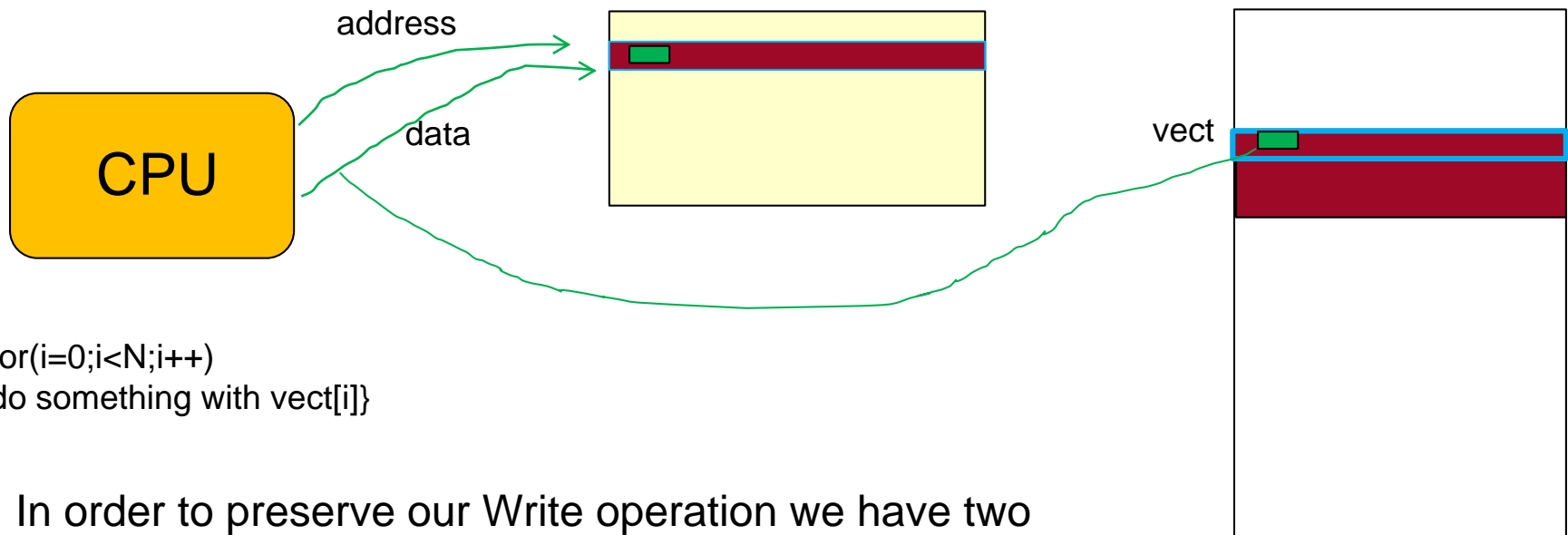
- Problems start to arise if we have a WRITE Operation



```
For(i=0;i<N;i++)  
{do something with vect[i]}
```

- Suppose we have a cache HIT: we can write the new data in the cache, but then the line in the main memory will be inconsistent
- Sooner or later the line will be removed from the cache to make space for new lines. At that point, without further action, we will lose the stored information.

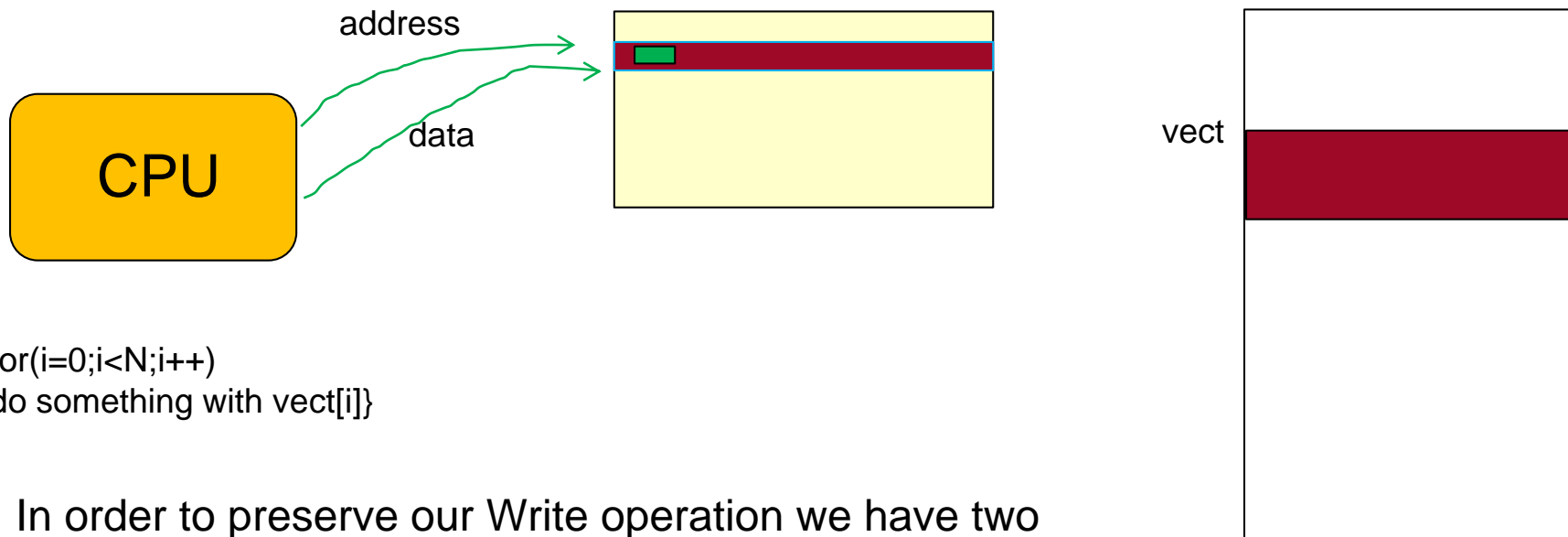
# Cache Write policies: WRITETHROUGH



```
For(i=0;i<N;i++)  
{do something with vect[i]}
```

- In order to preserve our Write operation we have two strategies:
  - **WRITETHROUGH**: We use the cache for reading, but when we write we write concurrently on cache and on the main memory

# Cache Write policies: WRITEBACK

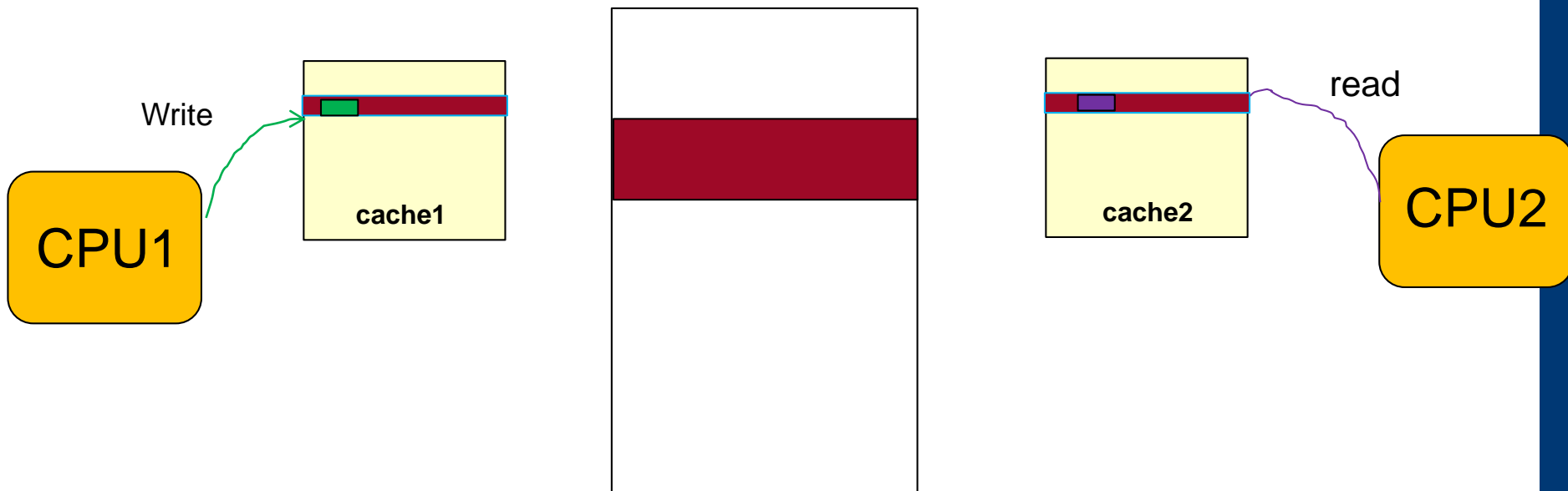


```
For(i=0;i<N;i++)  
{do something with vect[i]}
```

- In order to preserve our Write operation we have two strategies:
  - **WRITETHROUGH**: We use the cache for reading, but when we write we write concurrently on cache and on the main memory
  - **WRITEBACK**: We use the cache for reading and writing, but when the line is eventually removed from the cache we will have to WRITE BACK on the main memory the entire line

# Writeback is not possible if we have multiple CPUs

- Note that modern ICs may contain multiple CPU cores, that may have independent caches but refer to a single main memory: in this case we **SHOULD NOT** use write-back strategies, and we need to writethrough **EVERY** write operation:
- Also, in the example below cache 2 will have to be **REFRESHED** every time a location in cache 1 is being written to

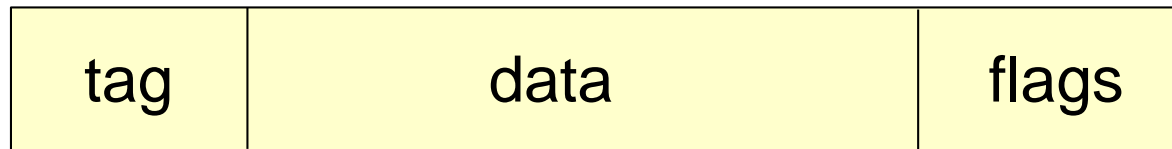


# Replacement Policies

- When we have a cache miss, a new line is loaded from the main memory into the cache
- The cache may be full, so that we need to replace the new line erasing one of the lines that are currently *cached*
- A **Replacement Policy** is the strategy used by CPU controller HW to select the line to be replaced: the most common is LRU (Least Recently Used)
- NOTES:
  - The user may know that some locations are accessed only very rarely, so it does not make much sense to load the relative line in the cache. In this case, that memory range is marked as **non-cacheable**: this improves performance by avoiding caching of memory regions that are rarely re-accessed (avoids overhead of loading into the cache a line that is not going to be used).
  - Some addresses must be by definition NON-CACHEABLE. Think about a peripheral: one of its registers may change values between two different accesses, so caching would be disruptive by preventing the CPU from seeing changes in the peripheral state

# Cache line Architecture

- The “cache” is composed of a HW controller and an SRAM Memory. The SRAM memory storage is organized as follows



Line “address”, used to **associate** the address produced by the CPU (physical memory address) with the data in the cache

Information on the line (example, is it empty, has it been written, when was it last used)

Data Line, containing the N memory words loaded from the Main memory

# Associativity

- **Associativity is the way a given cache line is associated to its original memory address**
- **The most relevant aspect of associativity is the replacement policy**
- *The replacement policy decides where in the cache a copy of a particular entry of main memory will go (we must try to avoid replacing a line that we will be using soon).*
- The more general solution is also the costlier from a hardware perspective: in a **FULLY ASSOCIATIVE Cache**, any new line can be placed in any line location in the cache. This makes cache access complex, because the controller needs to maintain a table of correspondence with the data address in main memory
- The cheaper solution is the DIRECT MAPPED CACHE: if each entry in main memory can go in just one place in the cache, the cache is *direct mapped*.

# Associativity

- Most caches implement a compromise in which each entry in main memory can go to any one of N places in the cache, and are described as N-way set associative

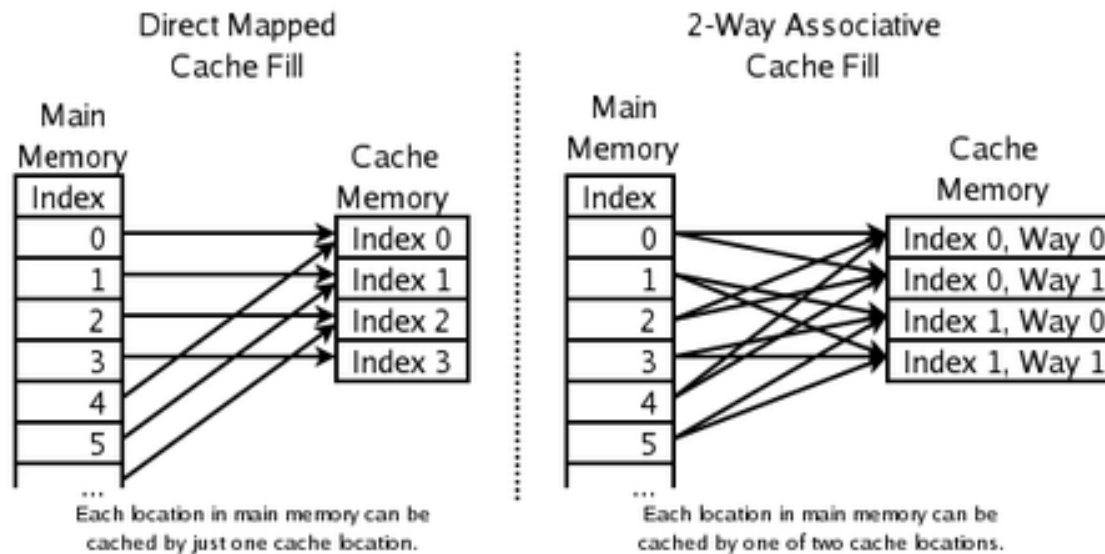


Image source <https://commons.wikimedia.org>

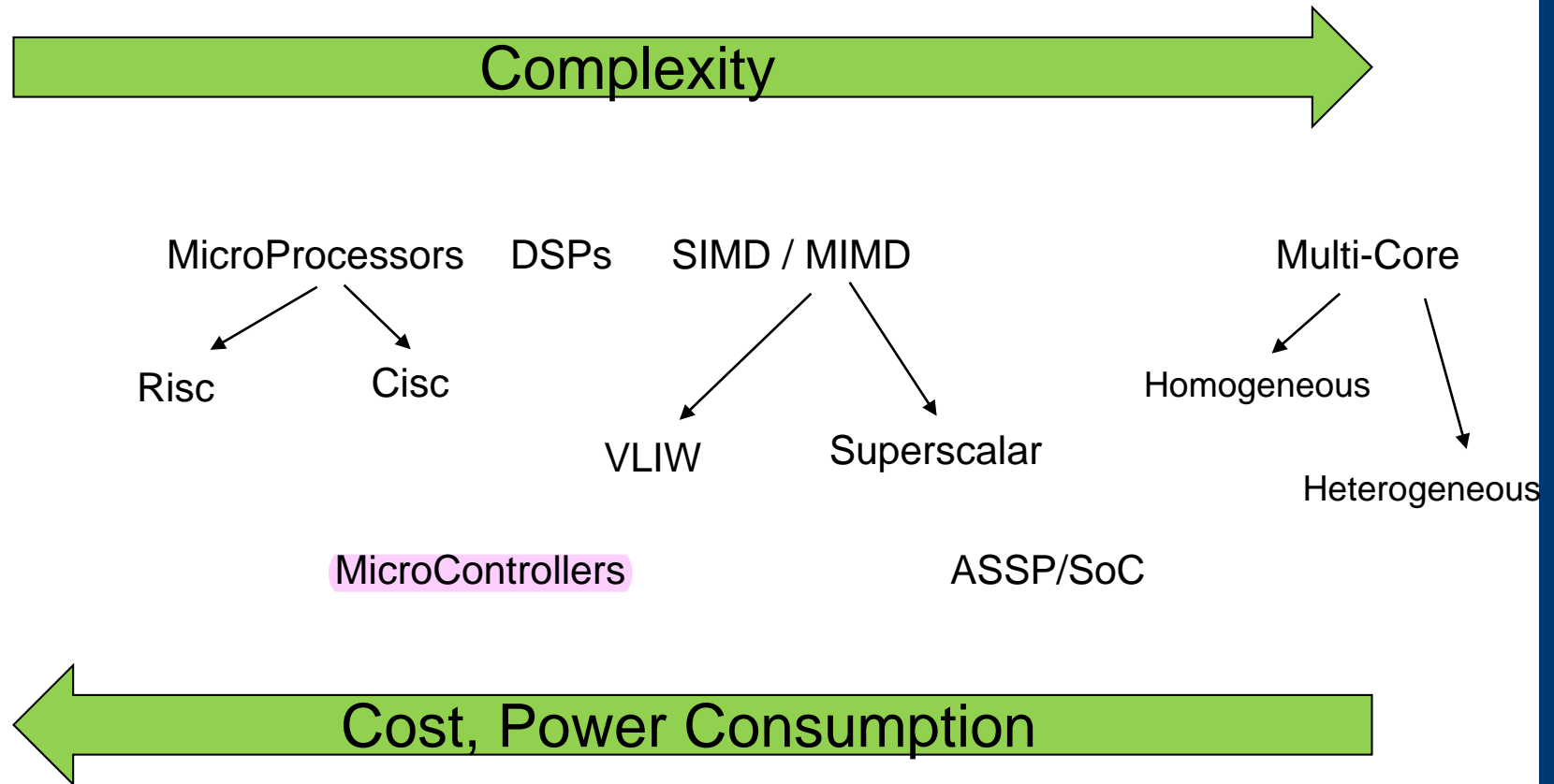


# Example: Student residences

- *Student residences in SFU can be considered an example of a Cache System: the addressing space is the whole of Canada (The world?) and SFU is the CPU. The data (Students) are located in the cache (residence) so that SFU has easier access to them*
- Now, every time SFU accesses a new student, it needs to replace one from the cache, and kick him/her out of residence.
- If the residence is *fully associative*, a new student will go to any available room, but then SFU must maintain a list associating any student's old address with the number of his/her residence room.
- We can have *direct mapped* residences: one residence room per Canadian province. If SFU needs to fit in a student from Nova Scotia, it will just kick away the previous Nova Scotian, even if the room for PEI is empty
- Or, a more human approach is to reserve  $N$  rooms for Atlantic provinces,  $N$  for Quebec,  $N$  Ontario, and  $N$  for the rest of Canada: this is an  $N$ -way associative cache: We still need to keep 4 lists, but lists are smaller and easier to work with



# Processor Classification



# MicroProcessor

- A microprocessor is a CPU architecture suitable for General Purpose computation.
- Nowadays it typically features a complex memory subsystem (Memory Management Unit, caches, DMA), and computation features such as embedded multiplier, FPU, and sometimes a larger number of registers
- Microprocessors nowadays operate typically at higher frequencies (Hundreds of MHz to a few GHz)
- In the 1990s, a differentiation emerged between EMBEDDED PROCESSORS (including ARM, MIPS) and Processors for Desktop/Laptop Computers (Intel x86 and compatible ones by AMD) with the former focusing on minimizing power consumption (embedded systems often have no cooling and sometimes are not plugged in to the power grid). The differentiation still exists, but the lines are getting more blurry. Apple is now using ARM for everything (though is using “high-end” ARM designs where needed), and Intel tried to get into the embedded processor market with products like the Atom processors.

# DSP (Digital Signal Processor)

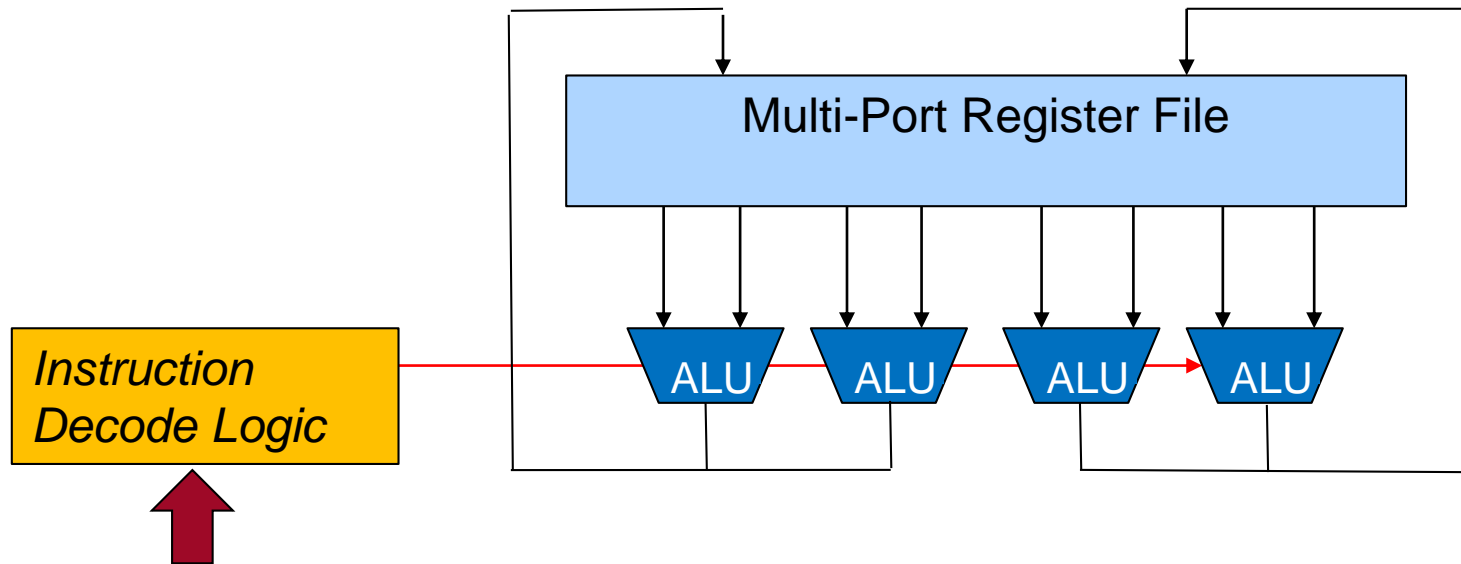
- The term DSP was introduced on the processor market in the 1980s by Texas Instruments.
- A DSP is essentially a processor that has **specific** hardware features to allow more efficient computation of **Specific** signal processing kernels: the most common features of DSP are Multiplication/Multiply-Accumulation Circuits, HW FPU, fast loop implementation, parallel access to memory
- DSPs are always targeted at a specific Application Field, using specific hardware acceleration for a specific subset of applications (Example: Sound processing, telecommunications, etc)
- *Today's processors like recent ARMs are more and more performant and sophisticated, and include most features of the DSP of the 90s (Multiplier, FPU, MLA), so the specific market segment of DSPs is somehow disappearing*
- *Same as most microcontrollers, today most DSPs are IC products featuring ARM+specific peripherals rather than a different CPU architecture*

# Computation parallelism in processor Architectures

- If the processor architecture needs very high performance, we first increase instruction throughput. *If an increased instruction throughput is not enough, then we need to resort to using more parallelism in the computation*
- Parallelism in processor architectures can be implemented in two strategies:
  - **SIMD** = Single Instruction Multiple Data
  - **MIMD** = Multiple instructions, Multiple Data

# SIMD Architectures

A **SIMD** (Single Instruction Multiple Data) architecture is capable of performing the same operation over multiple data: suppose we have many data that need the same instruction (e.g. multiple pixels of the same image), we can fetch one single instruction and control multiple ALUs

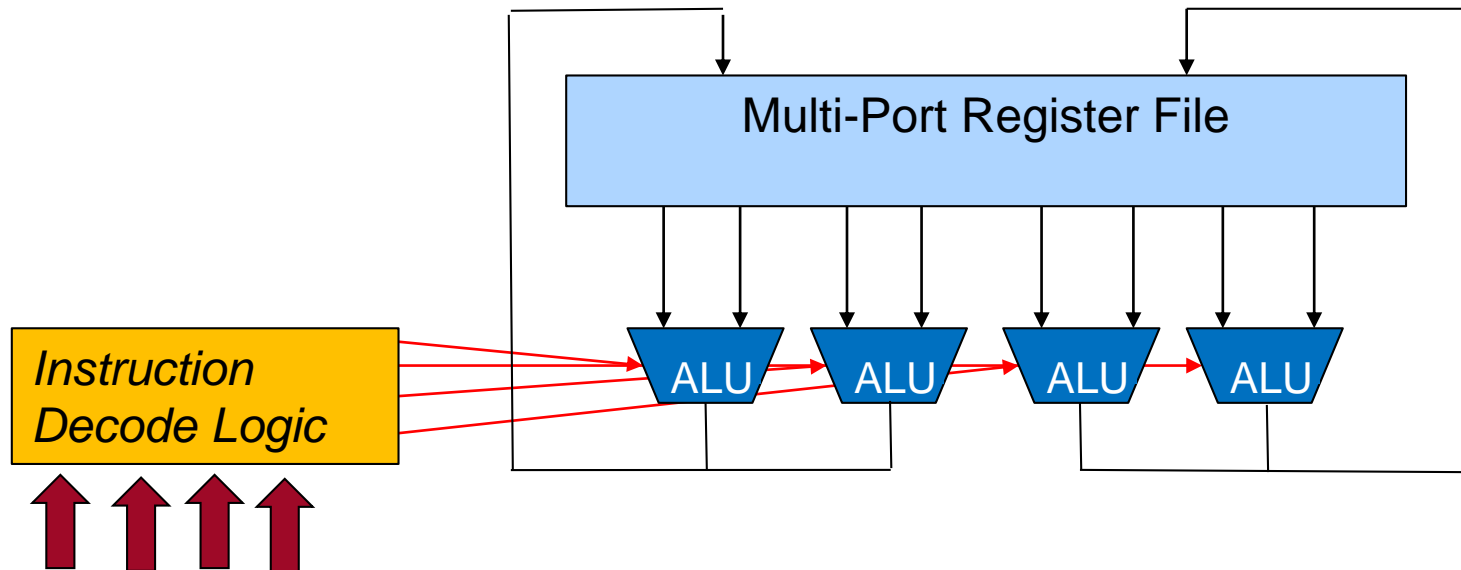


Instruction  
From IMemory

Example: Intel MMX (1996)  
More generally, Vector processors and earlier GPUs

# MIMD Architectures

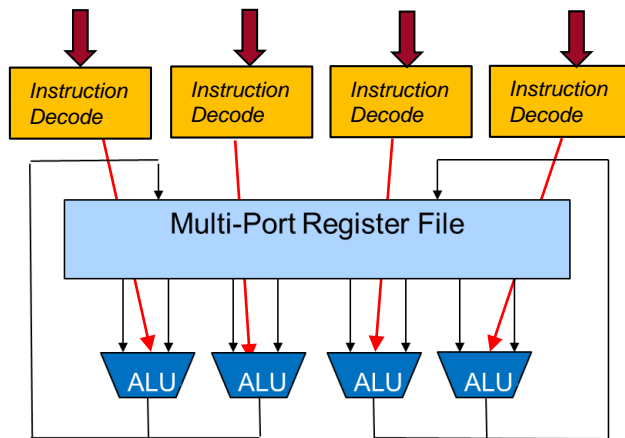
A **MIMD** architecture (Multiple Instruction Multiple Data) is capable of performing different concurrent operations over multiple data.



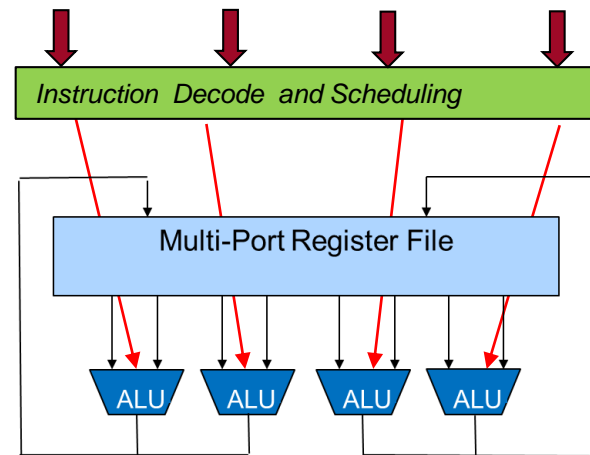
The problem with these architectures is that sometimes the result of an operation is the input of the following, making parallelism difficult to exploit

# VLIW and Superscalar processors

- Two classic examples of MIMD architectures are superscalar AND VLIW processors:
  - VLIW (Very Long Instruction Word) are capable of fetching/decoding more than one instruction at a time, and processing them concurrently. It is the compiler's responsibility to order the instructions in groups that can be computed together
  - Superscalar processors have dynamic scheduling capabilities, so they are capable of REORDERING fetched instructions on-the-fly in order to maximize parallelism.



**VLIW**

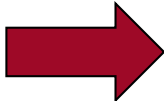


**Superscalar**



# VLIW and Super-Scalar Machines

- Investigation of parallel processor architectures was VERY “Hip” in 1980s/1990s, and several aspects have been borrowed from these architectures into modern processors, but these architectures have been outdated by Multi-Core processors



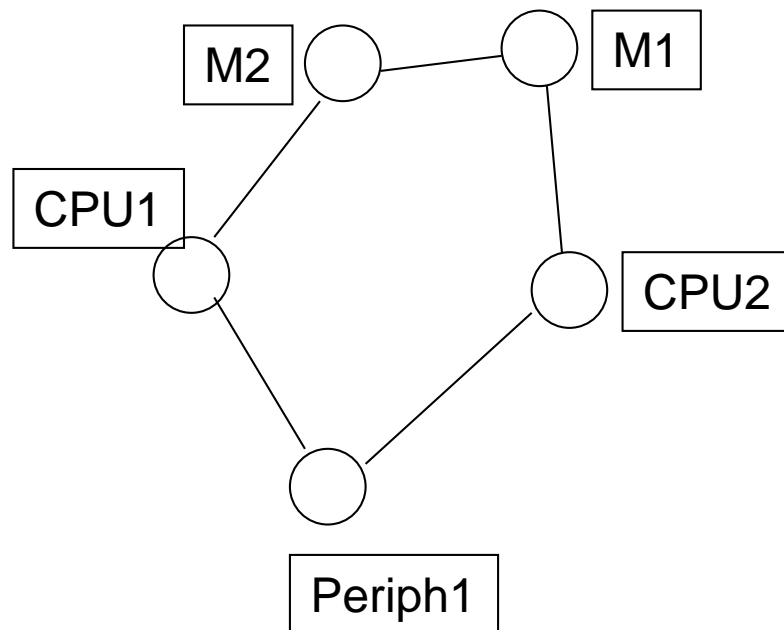
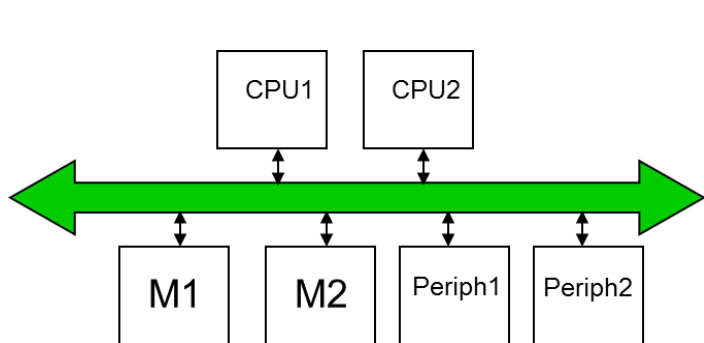
*In VLIW and superscalar CPUs it is up to the user/compiler to write code in order to maximize the available instruction parallelism. Writing parallel instructions is not natural for programmers, and not even for programming languages.*

*It is easier to run two programs concurrently than to build a single parallel program.*

**With the advance of technology, it makes more sense to be redundant and build several independent processors than build a single processor with multiple ALUs**

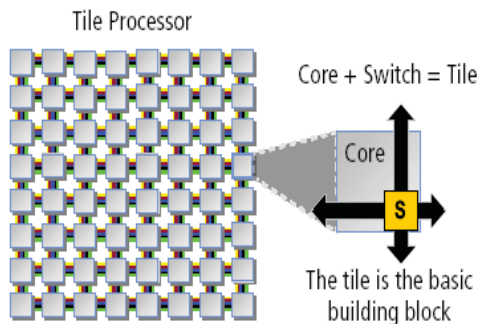
# Multi-Core Processors

- As we have solid compilers, processor architectures, and technology allowing us to map huge amounts of hardware onto a single chip, it makes sense to be redundant and realize multi-processor architectures on a chip.
- This allows one to exploit on-chip network protocol stacks developed across the years for computer networks.
- ***TODAY, on-chip communication happens through Network-on-Chip rather than through a bus***



# Multi-Core Processors

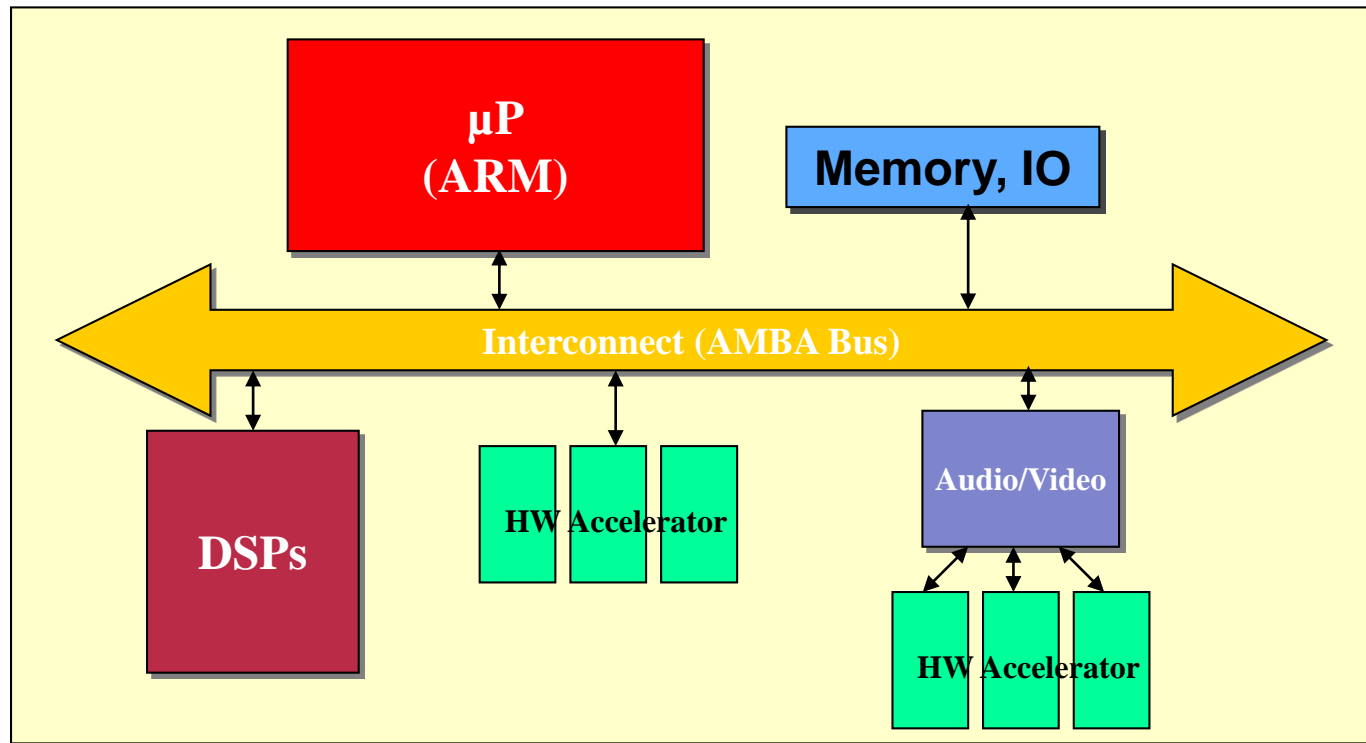
- Multi-Core computers range from Dual-core processors to Multi/Many Core systems, featuring a large array of processors (that manually or automatically switch tasks between them).
- A Multi-Core system can be HOMOGENEOUS or HETEROGENEOUS. Heterogeneous systems can be more efficient, but are more difficult to program
- Other than hardware redundancy, the most relevant issues for Multi-Core processors are
  - **TASK SCHEDULING** (Most of the time it is handled by OS)
  - **POWER CONSUMPTION** (Managed using Dynamic voltage/frequency scaling-> Every core can change its clock and voltage dynamically)
  - **SHARED MEMORY ACCESS CONSISTENCY** (Managed by ad-hoc cache strategies)



# Microcontroller

- A **microcontroller** (**uC**, or **MCU** for *microcontroller unit*) is a small [computer](#) on a single [integrated circuit](#). A microcontroller contains one or more processor cores along with [memory](#) and programmable [input/output](#) peripherals. (Wikipedia)
- A microcontroller often has a smaller processor focused at CONTROL tasks (Shuffling data around, providing commands and synchronization to industrial machinery, controlling a PCB board, reading sensor data).
- It usually provides limited computational capability and has **LOW COST, LOW POWER.**
- It has limited DMEM, large IMEM (Sometime non-volatile), few HW processor features (no Multiplier, No FPU), no memory management unit. Can be 8- , 16- or 32-bits. Frequencies are in the range of tens of MHz to limit power consumption.
- Typical case: PIC 8-bit Microcontroller family by MicroChip.
- Recently, however, ARM is so power efficient that it is taking over the microcontroller market.

# System-on-a-Chip (SoC)



- With the development of technology all components of a computation system can converge in a single large Integrated Circuit (System-on-a-Chip) where all architecture flavours can co-exist and cooperate. CPUs based on the SoC concept are often called “Application Processors”.
- Most smartphone processors belong to this category