ESRG
Embedded Systems
Research Group

CENTROALGORITMI

Advanced Computer Architectures

Caches

Filipe Salgado,
PhD Student @ ESRG

Daniel Oliveira,
Master Student @ ESRG
Jorge Pereira,
Master Student @ ESRG

School of Engineering,
University of Minho, Guimarães – Portugal

Universidade do Minho
Escola de Engenharia

COMPETE

QREN Quadro de Referência Estratégico Nacional

União Europeia
Fundo Europeu de Desenvolvimento Regional

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR

Contents

- Introduction
  - What is a cache memory?
  - Why to cache?
- Memory Hierarchy
- Cache Vs Memory
- What to cache?
  - Locality of Reference
- Cache Architecture
- Types of Cache
- Cache Policy
- Cache Implementation "From Scratch"

Introduction

- What is a cache memory?
  - A *cache* is a small, fast array of memory placed between the processor **core** and **main memory**.
  - The processor uses cache memory instead of main memory whenever possible to increase system performance.
- Why to cache?
  - The goal of a cache is to reduce the memory access bottleneck imposed on the processor core by slow memory.
  - By storing memory content in a fast access (but area expensive) memory, closer to the processor core this bottleneck can be attenuated.

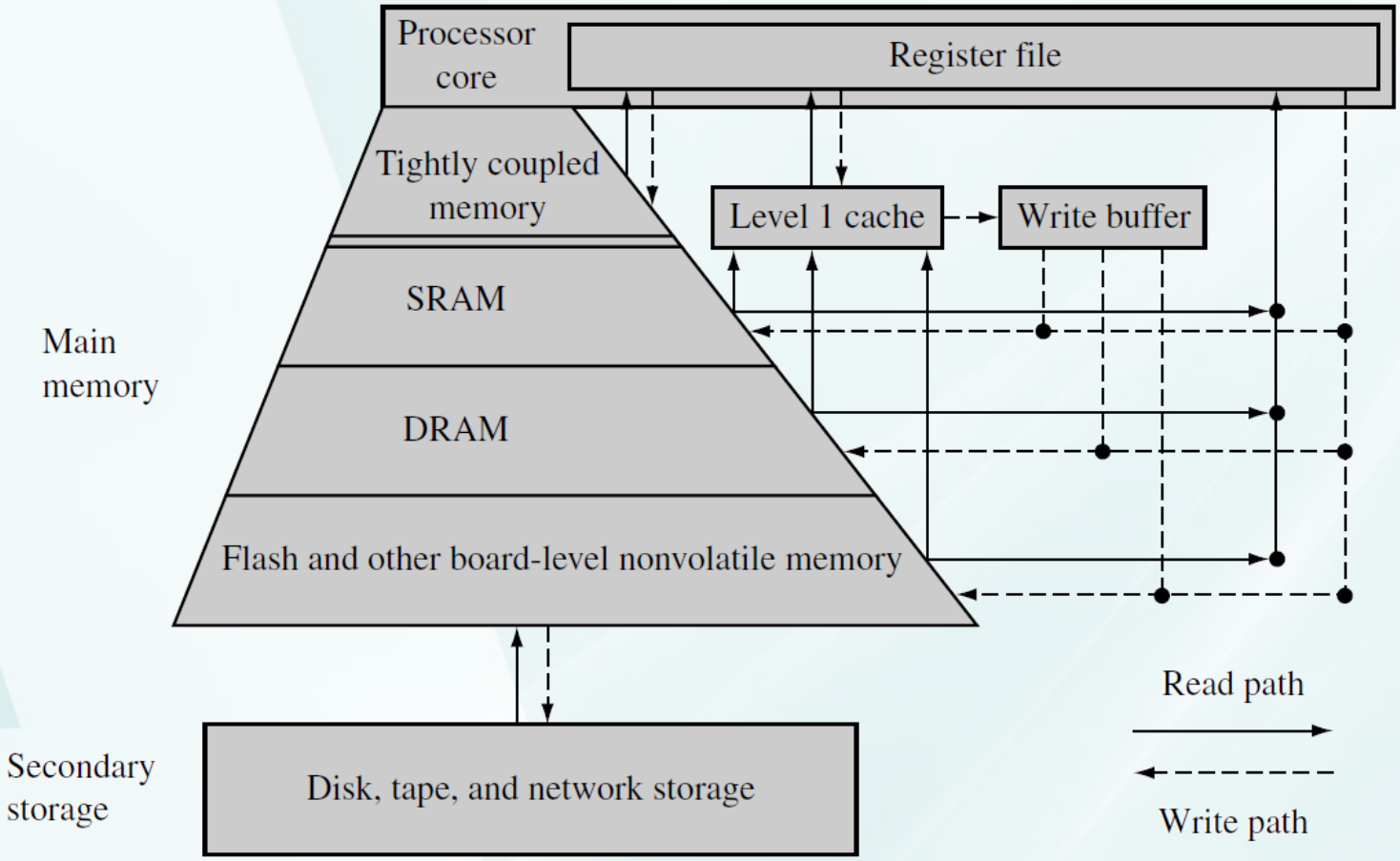
Introduction

- Are there any drawbacks introduced by a cache memory?
  - The main drawback is the difficulty of determining the execution time of a program.

i. A cache memory only represents a small portion of main memory. Thereby, the cache fills quickly during program execution.
ii. Once full, the cache controller frequently evicts existing code or data from cache to make more room for the new code or data.
iii. Thus, at any given instant in time, a value may or may not be stored in cache memory, since the process of eviction occurs randomly.

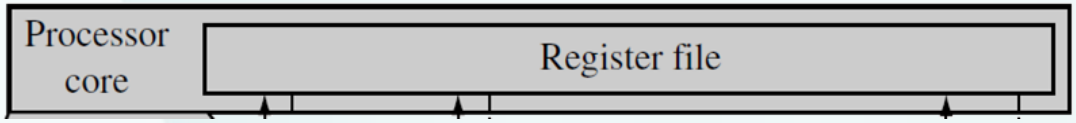
iv. In that way, **a routine may vary slightly from run to run**, since the data can be loaded from the cache memory or from the main memory.

Memory Hierarchy



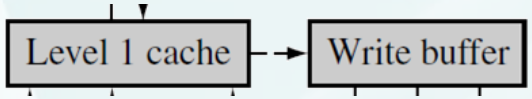
Memory Hierarchy

- 1st Level – Register File:



  - Tightly coupled to the processor;
  - RF provide the fastest possible memory access in the system.

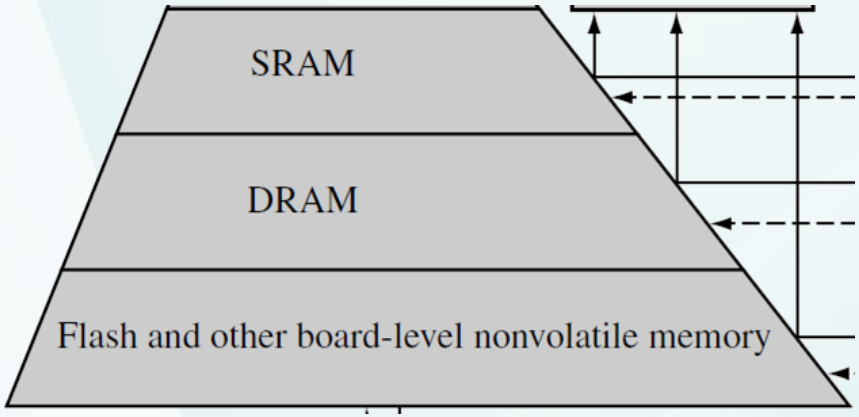
- 2nd Level – L1 Cache:



  - Can be incorporated between any level in the hierarchy where there is a significant access time difference between memory components.
  - Array of high-speed, on-chip memory that temporarily holds code and data from a slower level.

Memory Hierarchy

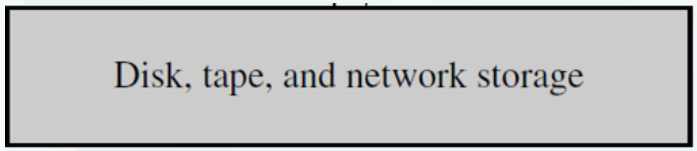
- 3th Level – SRAM/DRAM/nonvolatile components: (main memory):



  - The purpose of main memory is to hold programs while they are running on a system.

Memory Hierarchy

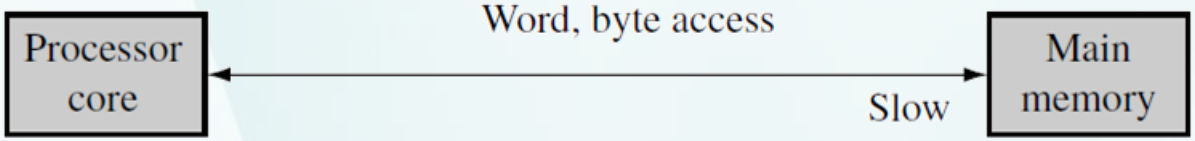
- 4th Level – Secondary storage (disk drives or removable drives):



  - Large, slow, relatively inexpensive mass storage.
  - Also data arrived from peripheral devices , which are characterized by their extremely long access times.

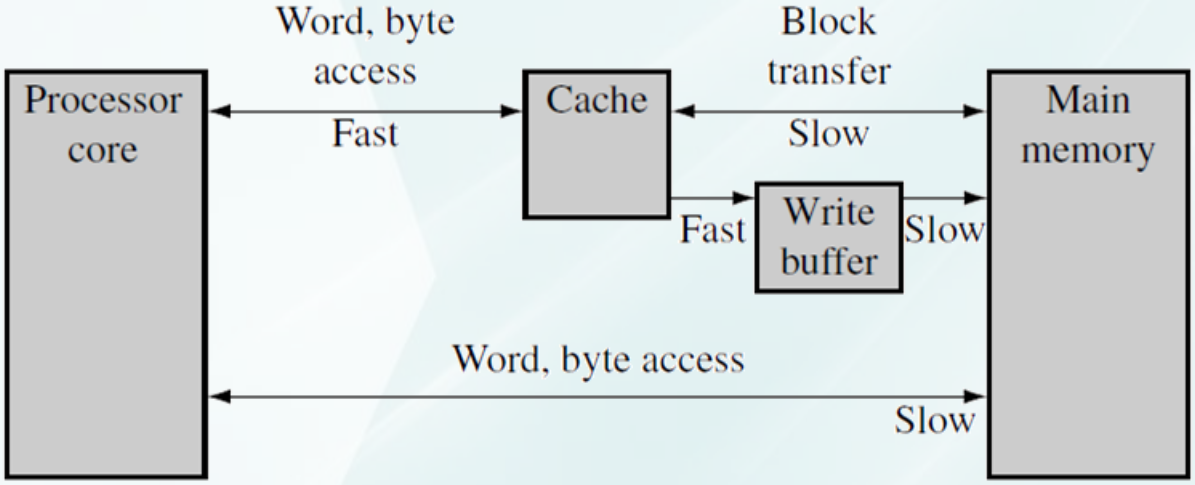
Cache Vs Memory:
The Relationship

- Non-cached system:



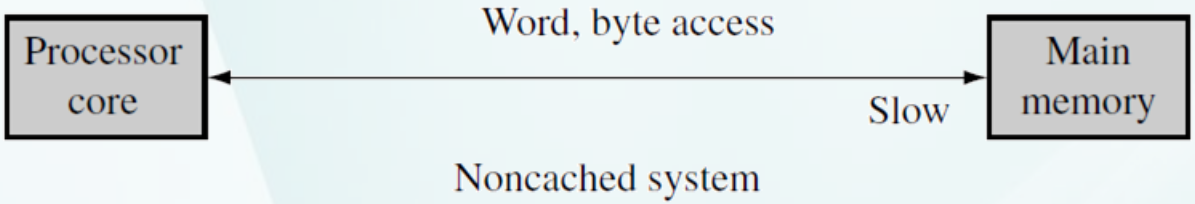
VS

- Cached system:



Cache Vs Memory:
The Relationship

- Non-cached system:

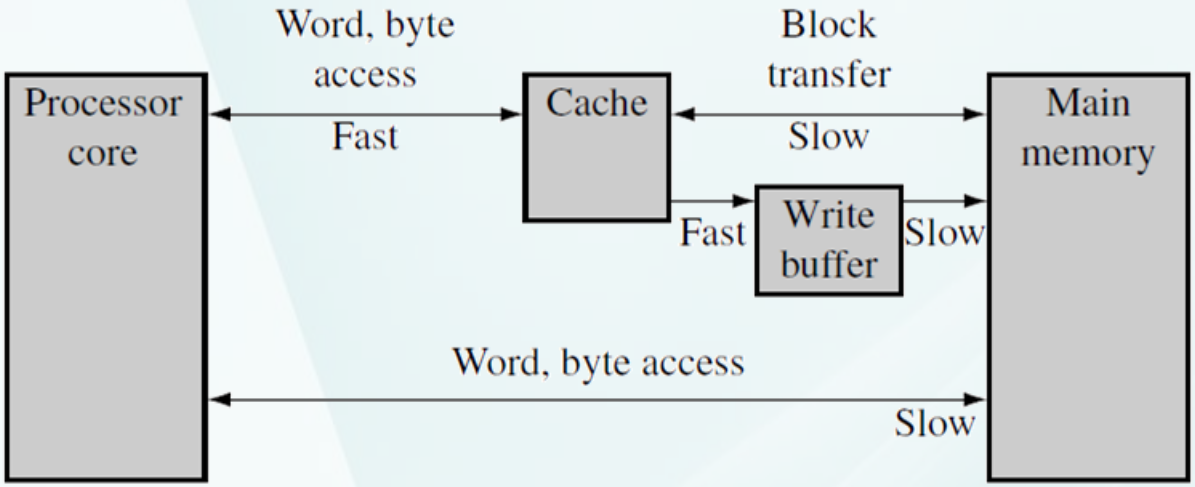


Noncached system

  - Main memory is accessed directly by the processor core using the datatypes supported by the processor core.

Cache Vs Memory:
The Relationship

- Cached system:



  - Cache memory is much faster than main memory and thus responds quickly to data requests by the core.
  - **The Relationship** involves the transfer of small blocks (the cache lines) of data between the slower main memory to the faster cache memory.

What to cache?

- Predictable program execution is the key to the success of cached systems.
  - If a program's accesses to memory were random, a cache would provide little improvement to overall system performance.

- Locality of Reference:
  - Principle that explains the performance improvement provided by the addition of a cache memory to a system.
  - States that software programs frequently run small loops.

This repetitiveness is the reason that a cache improves performance.

By loading the referenced code or data in to faster memory when first accessed, each subsequent access will be much faster.

What to cache?
Locality of Reference

- Principle of Temporal Locality:

"If a particular memory location at one point is required, it is very likely that it will be necessary again in a near future."

  - Example – a convenience store:
    - If you go there to buy sugar and they don't have, when they go to the warehouse, they'll get more than one packet.

  - Usage Example: Data memory (variables in loop cycles).

What to cache?
Locality of Reference

- Principle of Spatial Locality:

"If a particular memory location at one point is require, it is very likely that the nearby locations will be referenced too in a near future."

  - Example – a convenience store:
    - If you go there to buy potatoes and they don't have, when they go to the market, they'll take advantage of the trip and also buy carrots, tomatoes, etc.

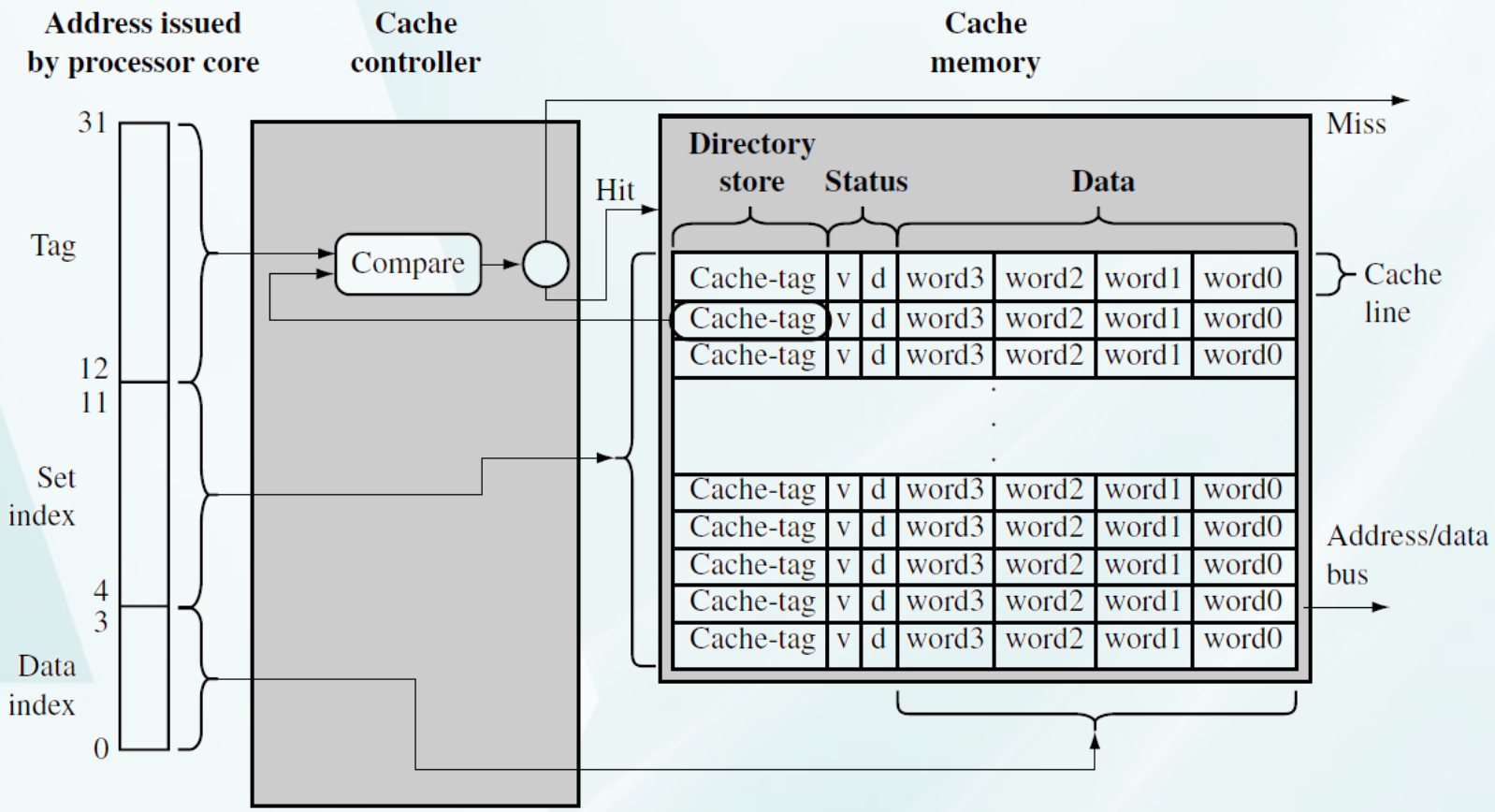
  - Usage Example: Instructions stored in code memory.

Cache Architecture

- Cache memory:
  - Dedicated memory array accessed in units called *cache lines*.
- Cache controller:
  - Hardware that copies code or data from main memory to cache memory automatically.
- In processor cores using Von Neumann architecture there is a single cache used for instruction and data.
- In processor cores using Harvard architecture there are two caches, one for instruction and the other one for data.

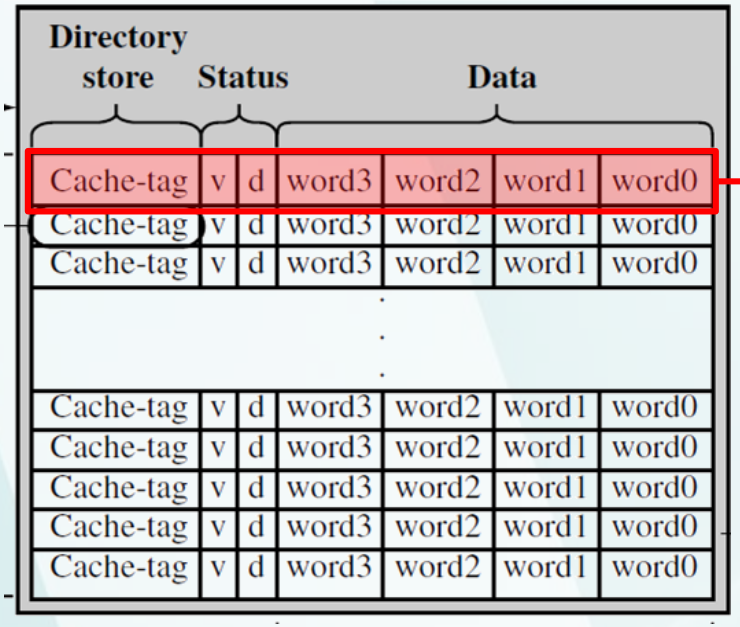
Cache Architecture

- Basic Cache Architecture:



Cache Architecture

- Basic architecture of a **Cache Memory**:



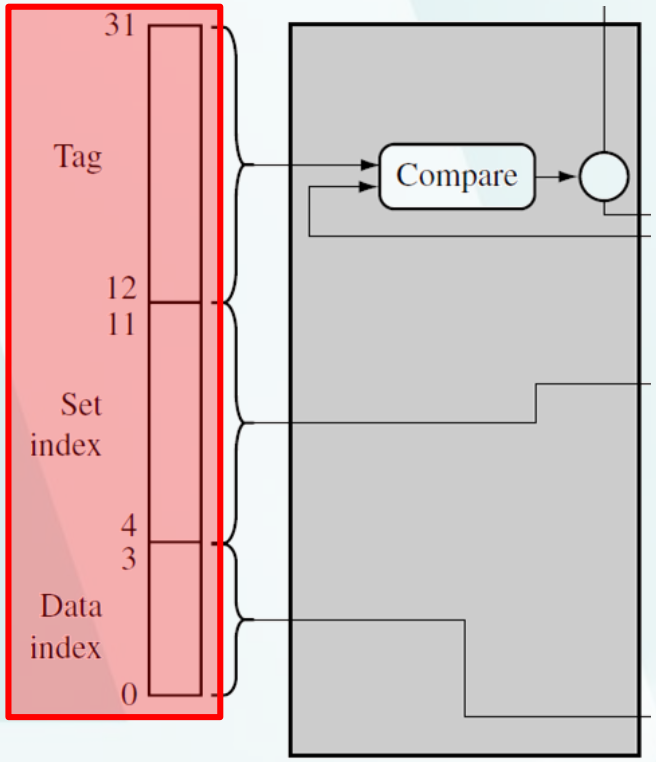
Cache Line
- Directory store: *cache-tag*
- Data section: *word's*
- Status information: *valid and dirty bit's*

- **Directory store:** holds the address where the cache line was copied from main memory.
- **Data section:** holds the data read from main memory.

- **Status information:**
  - Valid bit marks a cache line as active (data is valid and equals to the main memory).
  - Dirty bit defines whether or not a cache line contains data that is different from the one present in main memory.

Cache Architecture

- Basic operation of a **Cache Controller**:



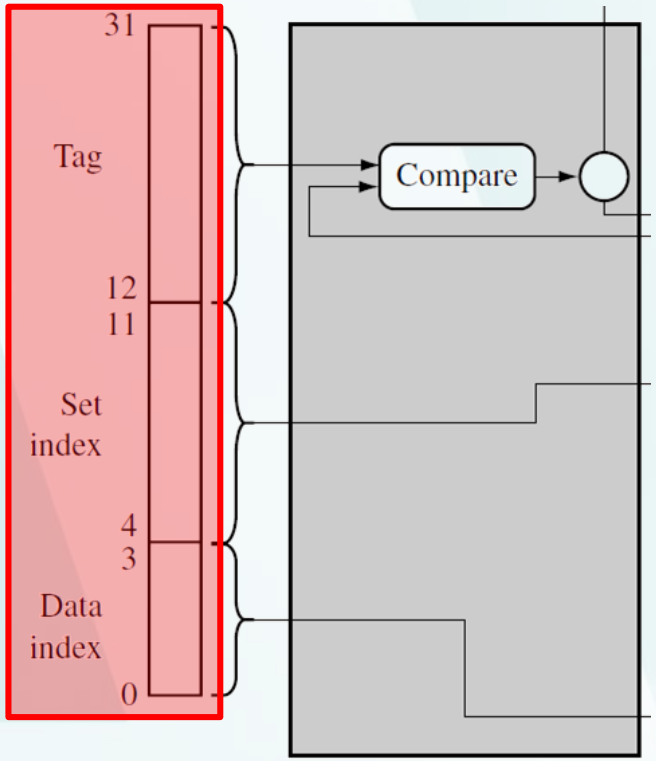
Divides the address of the read or write memory request in:

- Tag field;
- Set index field;
- Data index field.

1. CC uses the Set index to locate the cache line.
2. Checks the valid bit to determine if the cache line is active.
3. Compares the cache tag to the tag field.

Cache Architecture

- Basic operation of a **Cache Controller**:



Divides the address of the read or write memory request in:

- Tag field;
- Set index field;
- Data index field.

1. CC uses the Set index to locate the cache line.
2. Checks the valid bit to determine if the cache line is active.

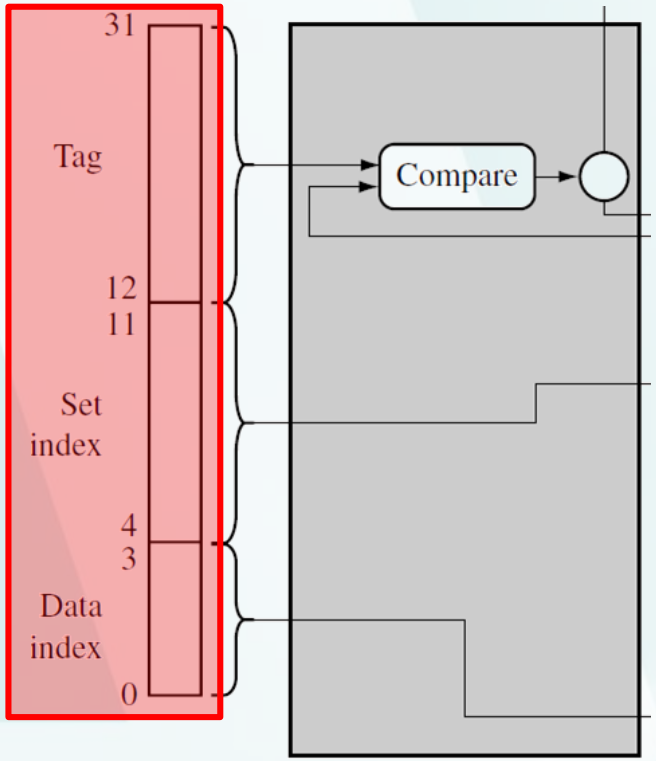
AND

1. Compares the cache tag to the tag field.

CACHE HIT!

Cache Architecture

- Basic operation of a **Cache Controller**:



Divides the address of the read or write memory request in:

- Tag field;
- Set index field;
- Data index field.

1. CC uses the Set index to locate the cache line.
2. Checks the valid bit to determine if the cache line is active.

AND/OR

1. Compares the cache tag to the tag field.

CACHE MISS!

Cache Architecture

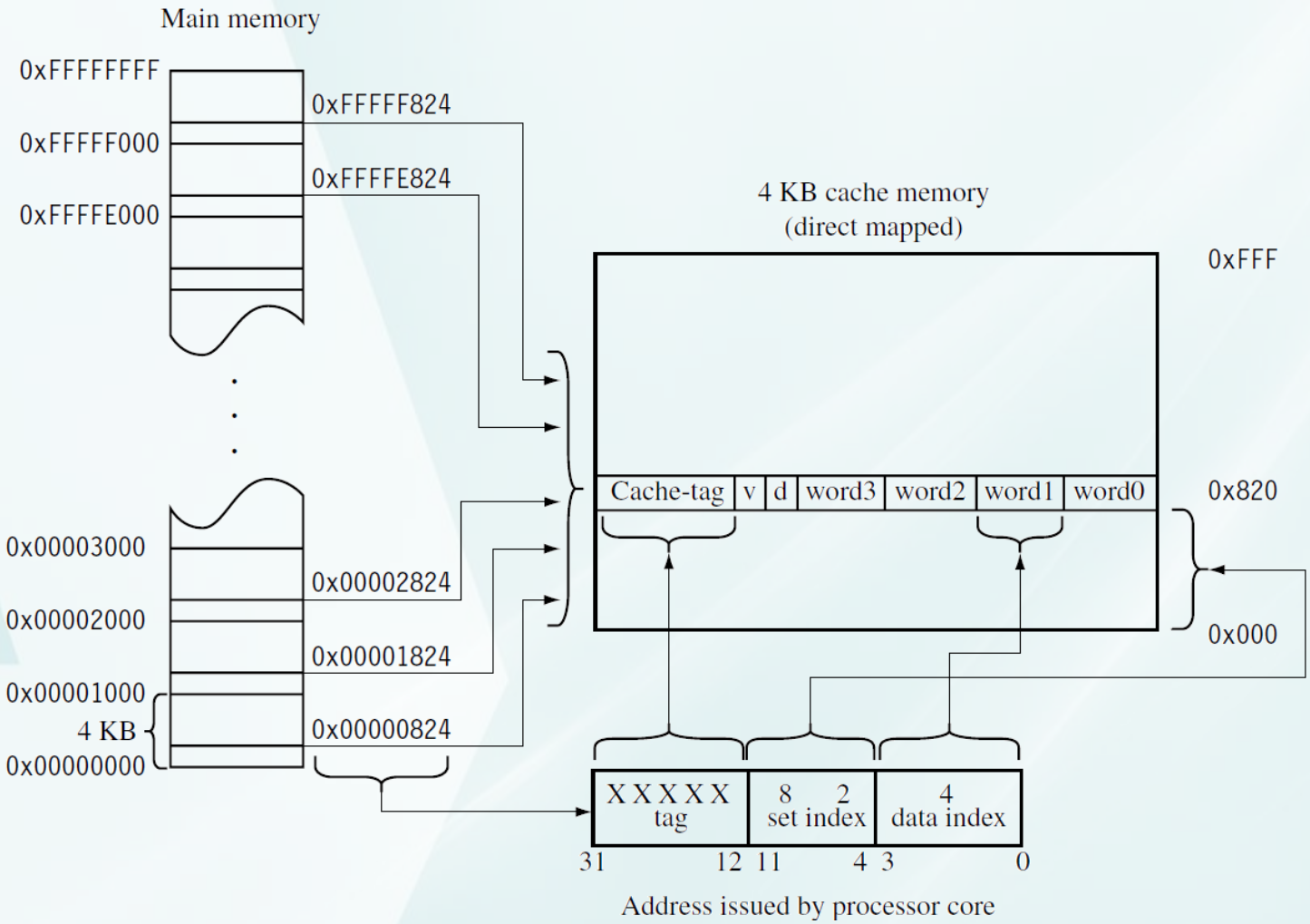
- Basic operation of a **Cache Controller**:
  - Cache Miss operation:
    - The controller copies an entire cache line from main memory to cache memory and provides the requested code or data to the processor. (cache line fill).
  - Cache Hit operation:
    - The controller supplies the code or data directly from cache memory (data field) to the processor.

Types of Cache

- Some organizational models used:
  - **Direct-mapped**
  - **N-way set associative**
  - **Hyper associative**
  - **Fully associative**

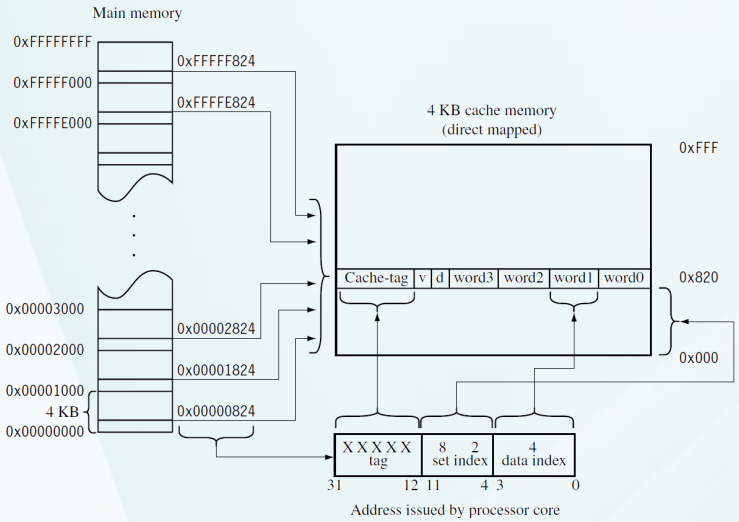
Types of Cache

- **Direct-mapped** cache:



Types of Cache

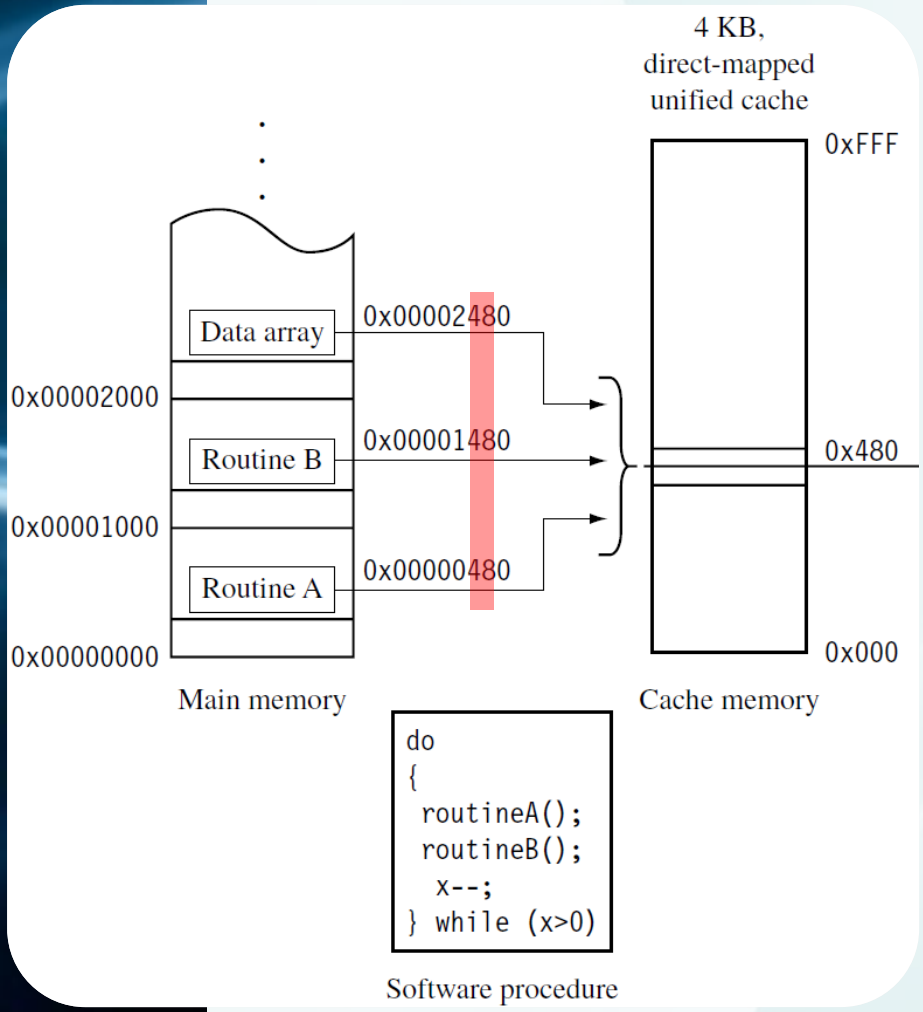
- **Direct-mapped** cache:



  - Each address location in main memory maps to a single location in cache memory;
    - There will be many addresses in main memory that map to the same single cache line (**0x824** example)
  - Is a simple solution that is easier to implement;
  - One major problem: high level of trashing.
    - A software battle for the same location in cache memory
    - Results: **repeated loading and eviction of a cache line!**

Types of Cache

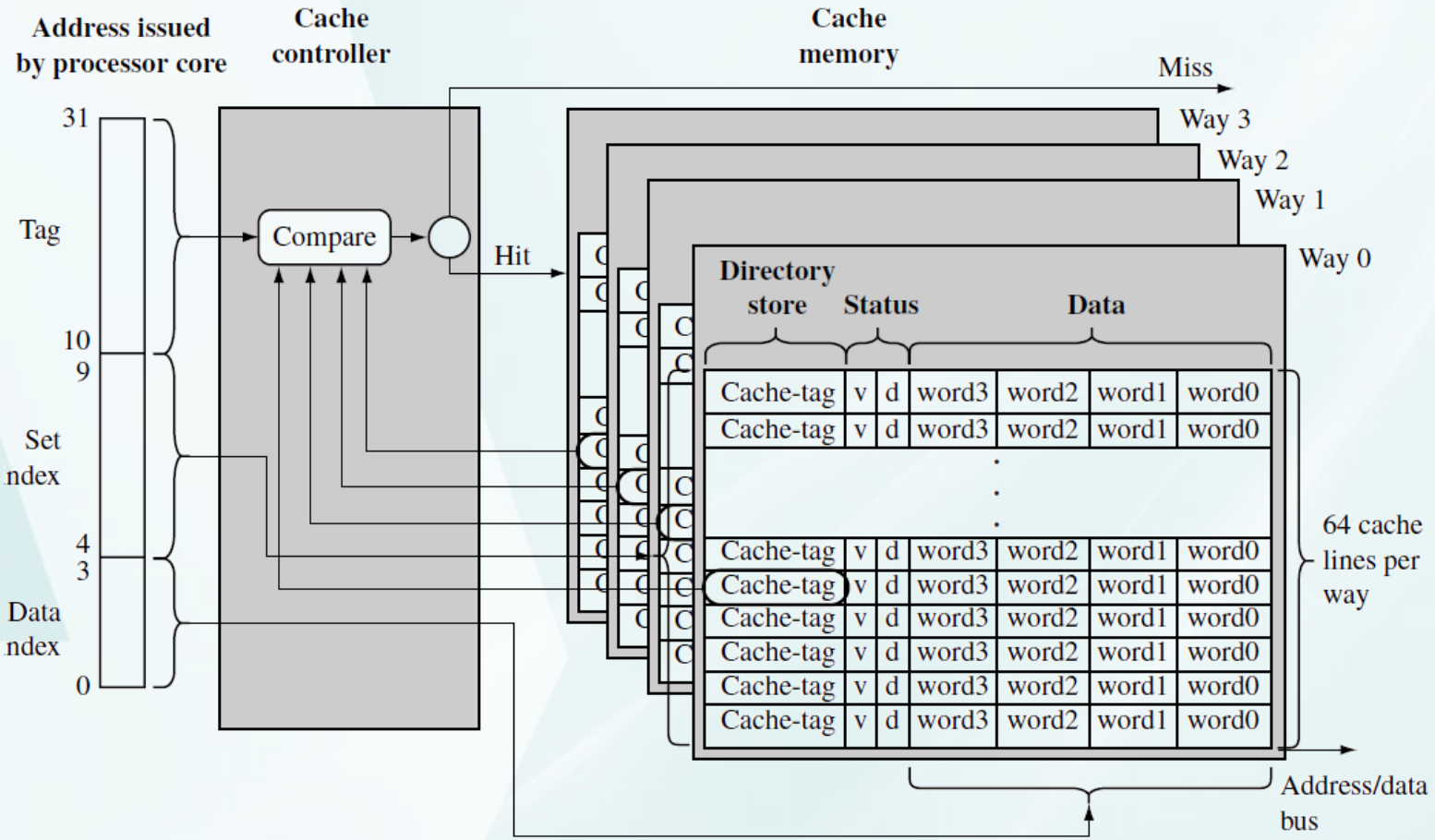
- **Trashing** problem demystified:



- The procedure calls two routines repeatedly in a **do/while loop**;
- routineA() & routineB() have the same **set index address** (in red - 48)
  - ...the routines are found at addresses in physical memory that map to the same location in cache memory.
- Repeated cache misses result in continuous eviction of the routine that not running. This is **CACHE TRASHING**.

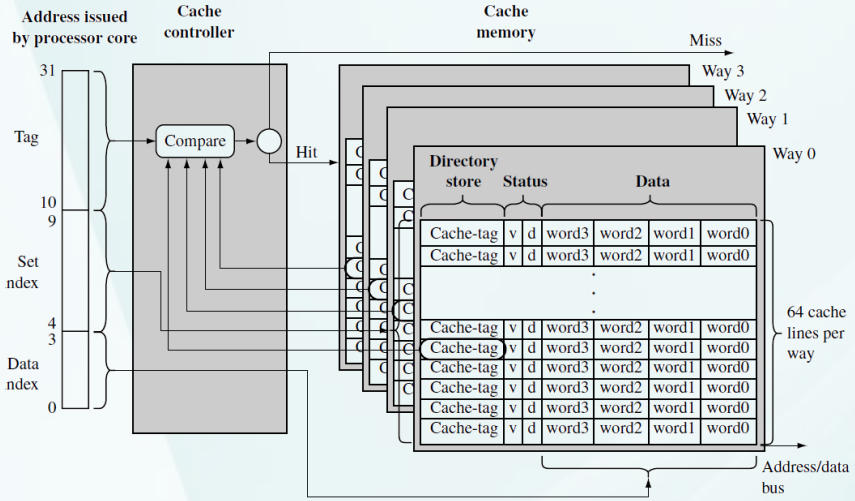
Types of Cache

- **N-way set associative** cache (typically N is 2,4,8 sets):



Types of Cache

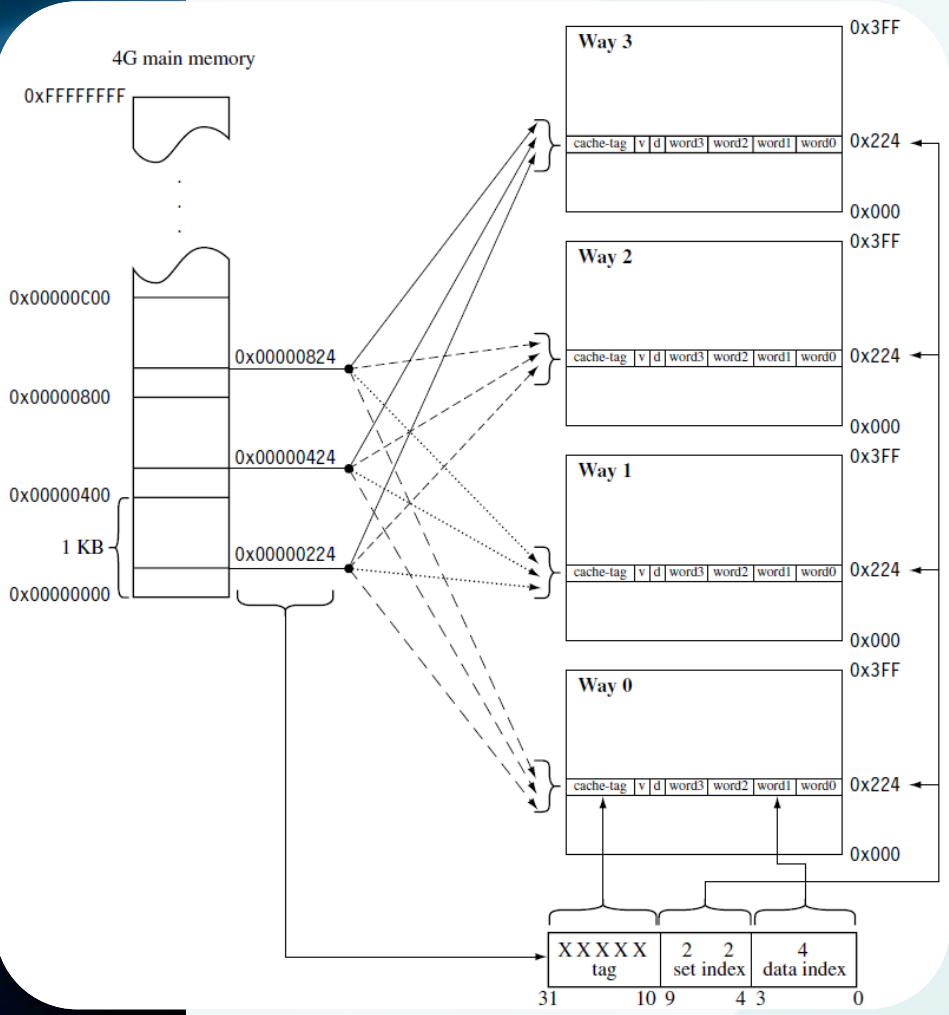
- **N-way set associative** cache (typically N is 2,4,8 sets):



  - Used to reduce the frequency of trashing;
  - Divides the cache memory into smaller equal units, called *ways*;
  - Same principle of direct mapped cache;
  - Example: 4-way set associative 4KB cache
    - Each way has **1KB with 64 lines**.
    - The **set index** addresses **more than one cache line** (it points to one cache line in each way – in this case 4)

Types of Cache

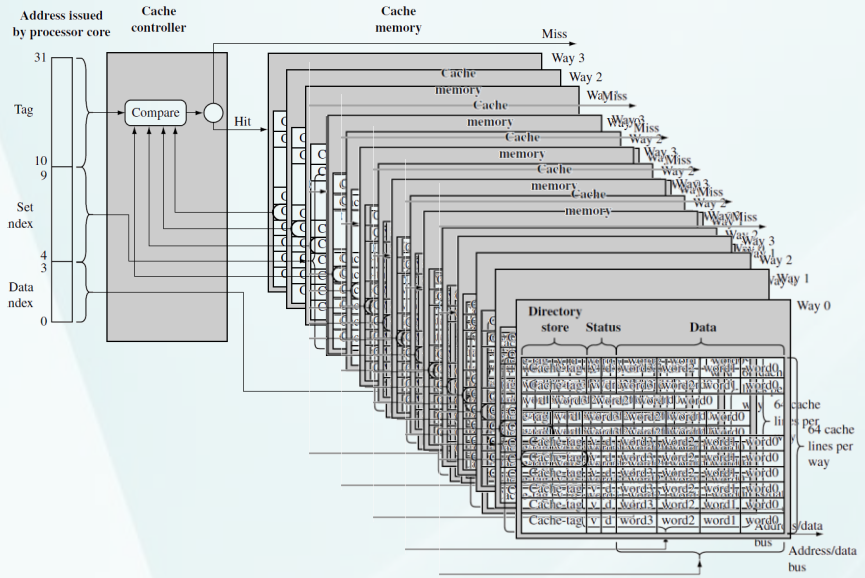
- **N-way set associative** cache (typically N is 2,4,8 sets):



  - The set of caches lines pointed to by the set index are **set associative**;
  - The figure on the left shows the mapping of main memory to a 4-way set associative cache:
    - Any single location in main memory now maps to four different locations in the cache.

Types of Cache

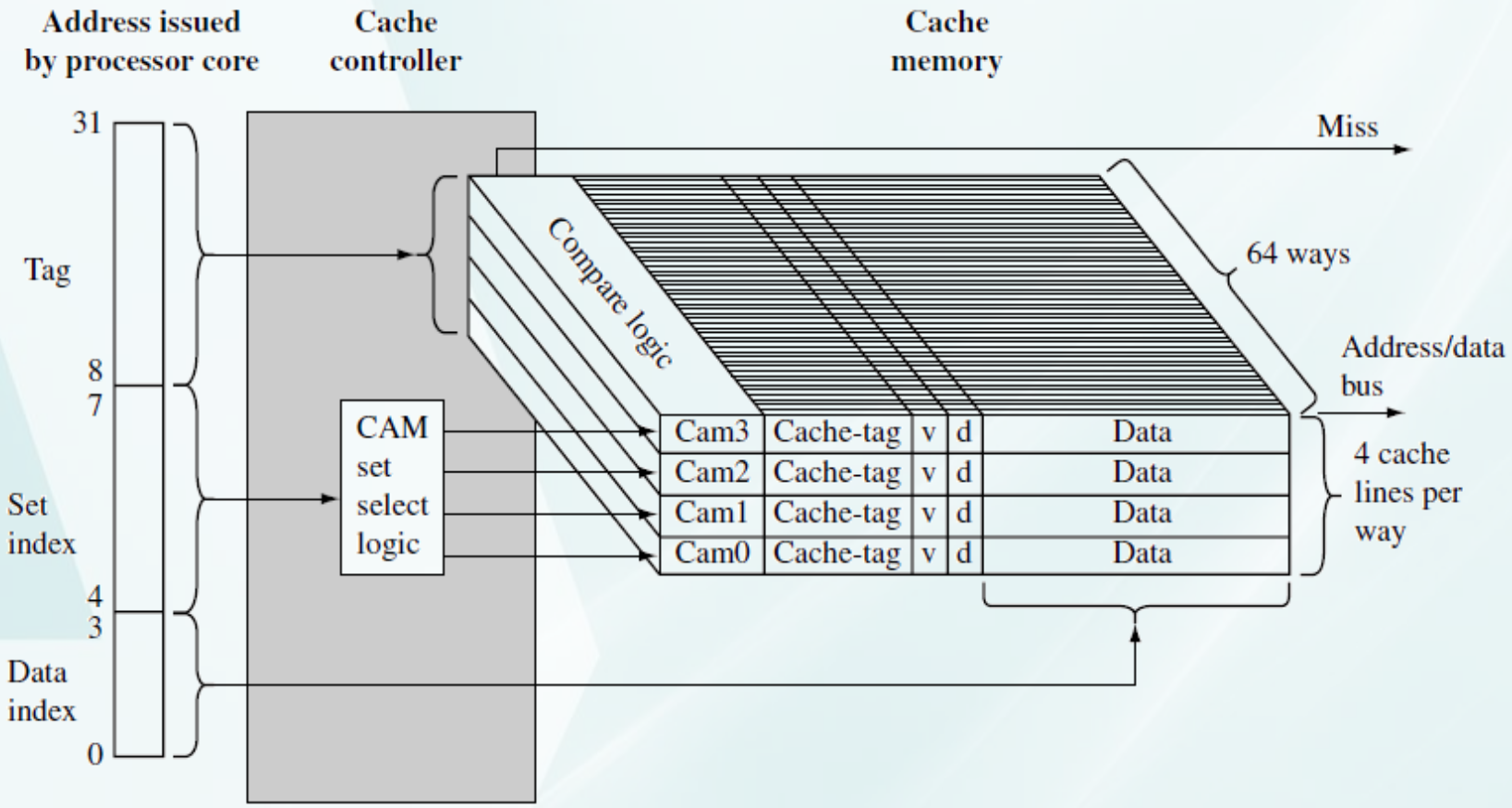
- **Hyper-associative** cache (typically N is 16,32,64,128,… sets):



  - Its an implementation of a N-way set associative with a huge number of small sets;
  - Common in super scalar implementations;
  - Increased number of sets to increase hit rate in applications with several small data work sets.

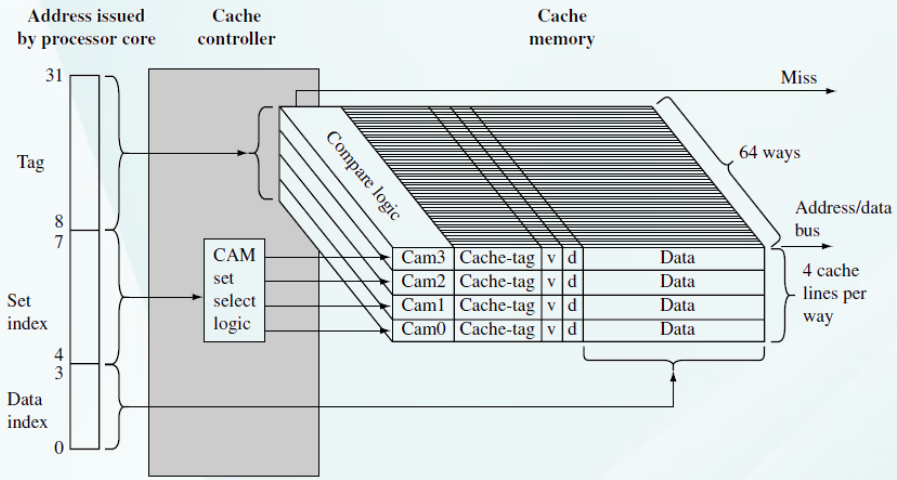
Types of Cache

- **Fully associative** cache:



Types of Cache

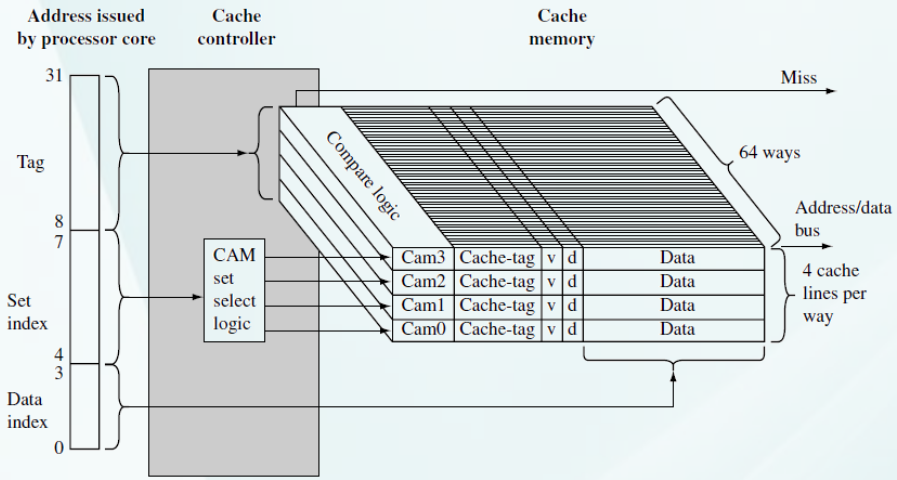
- **Fully associative** cache:



  - As the **associativity** of a cache goes up, the probability of **trashing** goes down.
    - In theory the ideal goal would be have any main memory location mapped to any cache line
    - **But**, in practice as the associativity increases, so does the complexity of the hardware that supports it.
    - A method to implement this includes a CAM (Content Addressable Memory)

Types of Cache

- **Fully associative** cache:



  - CAM:
    - Uses a set of comparators to compare the input tag address with a cache tag stored in each valid cache line.
    - The cache controller uses the address tag as the input to the CAM and the output select the way containing the valid cache line.
    - Reverse memory: user provides the content, it gives the address where it is stored.

Cache Policy

- Three policies that determine the operation of a cache:
  - **Write Policy**
    - Writeback or Writethrough
  - **Replacement Policy**
  - **Allocation Policy**

Cache Policy
Write Policy

- When the processor core writes to memory, the cache controller has two alternatives for its write policy:
  - **Writethrough**:
    - When there is a cache hit on write, the cache controller writes to both cache and main memory, ensuring that the cache and main memory stay coherent at all times;
    - Because of the write to main memory this policy is slower than a writeback policy.
  - **Writeback**:
    - The cache controller writes to valid cache data memory and not to main memory;
    - Consequently, valid cache lines and main memory may contain different data;
    - The dirty bits in cache lines status information block are always changing;
      - When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true.

Cache Policy
Replacement Policy

- On a cache miss, the CC must select a cache line from the available set in cache memory to store the new information from main memory:
  - The cache line selected for replacement is know as a ***victim***.

- The strategy implemented in a cache controller to select the next victim is called its **replacement policy**.
  - It selects a cache line from the available associative member set (way);

  - The **set index selects the set of cache lines available in the ways** and the **replacement policy selects the specific cache line from the set to replace.**

Cache Policy
Replacement Policy

- Two replacement policies examples used in ARM cached cores:
  - Round-robin replacement:
    - It simply selects the next cache line in a set to replace.
    - The selection algorithm uses a sequential, incrementing victim counter that increments each time the CC allocates a cache line.
    - **Greater predictability** (!)
  - Pseudorandom replacement:
    - Randomly selects the next cache line in a set to replace.
    - The selection algorithm uses a non-sequential incrementing victim counter.

Cache Policy
Replacement Policy

- Another replacement policy:
  - Least Recently Used (LRU):
    - This policy keeps track of cache line use and selects the cache line that has been unused for the longest time as the next victim.

Cache Policy
Allocation Policy – Cache miss

- ARM cores use one of two strategies to allocate a cache line after the occurrence of a **cache miss**:
  - Read allocate:
    - Allocates a cache line only during a read from main memory;
    - If the victim cache line contains valid data, then it is written to main memory before the cache line is filled with new data.
  - Read-write allocate:
    - Allocates a cache line for either a read or write to memory;
    - Any load or store operation made to main memory, which is not in cache memory, allocates a cache line.

Cachegen Tool
Overview

- It's a software developed by
  - Peter Yiannacouras for partial fulfillment of BASc at the University of Toronto
  - For Linux

- Simple cache generator, able to generate synthesizable **Verilog** code for **associative**, **direct mapped** and **2-way set associative** caches.

- Generates only "one word per line" cache sets.

- Useful for educational purposes!

- Lets try it ;)

Cachegen Tool

- Go to Linux;
- Extract the tarball;
- Run the following commands:

```
embeddedgate@ubuntu: ~/cachegen
embeddedgate@ubuntu:~$ cd cachegen/
embeddedgate@ubuntu:~/cachegen$ ls
design  genall.sh  main.c  Makefile  README.txt
embeddedgate@ubuntu:~/cachegen$ make
gcc -g -o cachegen main.c -lm
embeddedgate@ubuntu:~/cachegen$ ls
cachegen  design  genall.sh  main.c  Makefile  README.txt
embeddedgate@ubuntu:~/cachegen$ ./cachegen

CAM GENERATION
Usage: cachegen -c <depth> <width> <output location>
ASSOCIATIVE CACHE GENERATION
Usage: cachegen -a [-hei] <cache depth> <address width> <data width> <output loc
ation>
DIRECT MAPPED CACHE GENERATION
Usage: cachegen -d [-hei] <cache depth> <address width> <data width> <output loc
ation>
TWO WAY SET ASSOCIATIVE CACHE GENERATION
Usage: cachegen -t [-heiw] <cache depth> <address width> <data width> <output lo
cation>

        OPTIONS:
        -h Cache Hit - outputs a signal that indicates when a cache hit occurs
        -e Expand Read - Adds an extra cycle to the read operation improving fma
x
        -i Intercept - passes read write signals to slave only when necessary
        -w Expand Write - Adds an extra cycle to the write operation improving f
max
embeddedgate@ubuntu:~/cachegen$
```

Cachegen Tool

- Cache depth?
- Cache width?
- Data width?

```
embeddedgate@ubuntu: ~/cachegen
embeddedgate@ubuntu:~$ cd cachegen/
embeddedgate@ubuntu:~/cachegen$ ls
design  genall.sh  main.c  Makefile  README.txt
embeddedgate@ubuntu:~/cachegen$ make
gcc -g -o cachegen main.c -lm
embeddedgate@ubuntu:~/cachegen$ ls
cachegen  design  genall.sh  main.c  Makefile  README.txt
embeddedgate@ubuntu:~/cachegen$ ./cachegen

CAM GENERATION
Usage: cachegen -c <depth> <width> <output location>
ASSOCIATIVE CACHE GENERATION
Usage: cachegen -a [-hei] <cache depth> <address width> <data width> <output loc
ation>
DIRECT MAPPED CACHE GENERATION
Usage: cachegen -d [-hei] <cache depth> <address width> <data width> <output loc
ation>
TWO WAY SET ASSOCIATIVE CACHE GENERATION
Usage: cachegen -t [-heiw] <cache depth> <address width> <data width> <output lo
cation>

        OPTIONS:
        -h Cache Hit - outputs a signal that indicates when a cache hit occurs
        -e Expand Read - Adds an extra cycle to the read operation improving fma
x
        -i Intercept - passes read write signals to slave only when necessary
        -w Expand Write - Adds an extra cycle to the write operation improving f
max
embeddedgate@ubuntu:~/cachegen$
```

Cachegen Tool

- **Cache depth:**
  - Number of cache lines present in the cache set and tag memory. In other words is the number of **cache lines**.
- **Address width:**
  - Number of bits available to address memory. It's a consequence of the architecture and relates with the memory size.
- **Data width:**
  - Number of bits per cache line. In other words is the **line size**.

Cachegen Tool

- **Metrics examples:**
  - Cache depth: 32
  - Address width (address space): 8
  - Data width (line size): 16

```
embeddedgate@ubuntu: ~/cachegen
embeddedgate@ubuntu:~/cachegen$ ./cachegen -a -h 32 8 16 ./associative
Creating directory ./associative
Generating 32x16 associative cache ... done
Generating 32:5 encoder ... done
Generating 32x8 cam ... done
embeddedgate@ubuntu:~/cachegen$ ./cachegen -d -h 32 8 16 ./direct
Creating directory ./direct
Generating 32x16 direct-mapped cache ... done
embeddedgate@ubuntu:~/cachegen$ ./cachegen -t -h 32 8 16 ./setassociative
Creating directory ./setassociative
Generating 32x16 two way set associative cache ... done
embeddedgate@ubuntu:~/cachegen$ ls
associative  design  genall.sh  Makefile    setassociative
cachegen     direct  main.c     README.txt
embeddedgate@ubuntu:~/cachegen$
```

Cache Implementation

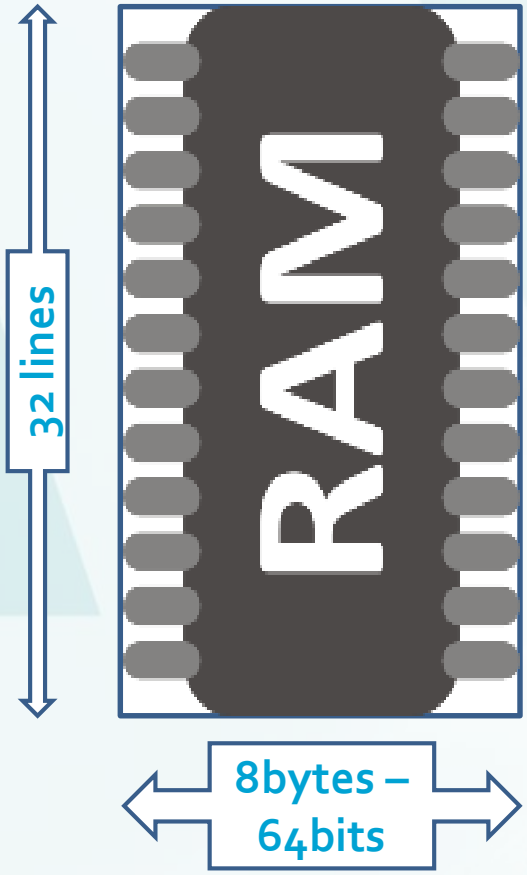
Direct Mapped & 2-way Set Associative

- What about if Cachegen doesn't fit your requirements? (performance, words per line, latency, etc)

- Lets create a cache from scratch!
  - Simplest design: **Direct mapped Instruction Cache**
    - What's the main difference between **Instruction** and **Data** caches?
  - More complex design: **2-way Set Associative Instruction Cache**

Cache Implementation

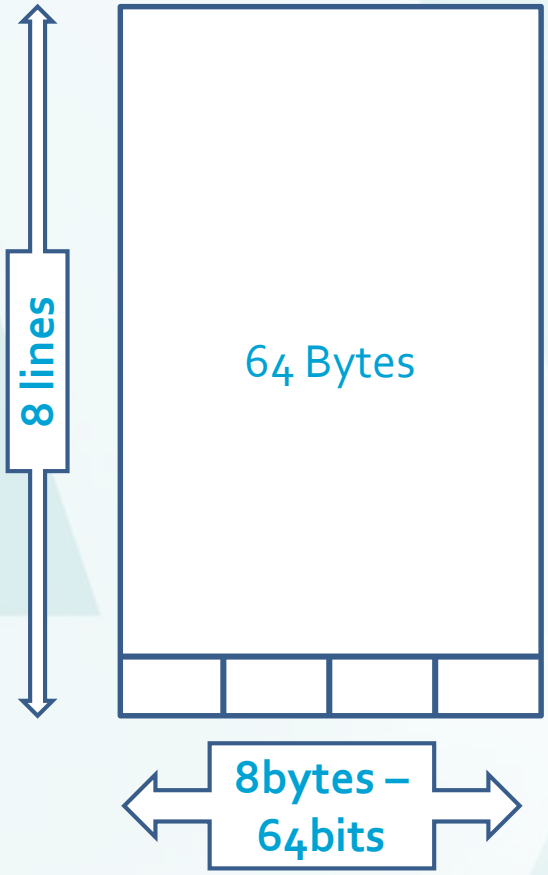
- **Direct-mapped** cache:
  - Metrics:
    - Off-chip Memory:



  - Requires 5 bit to address the lines
  - 256 bytes memory

Cache Implementation

- **Direct-mapped** cache:
  - Metrics:
    - Cache Set:

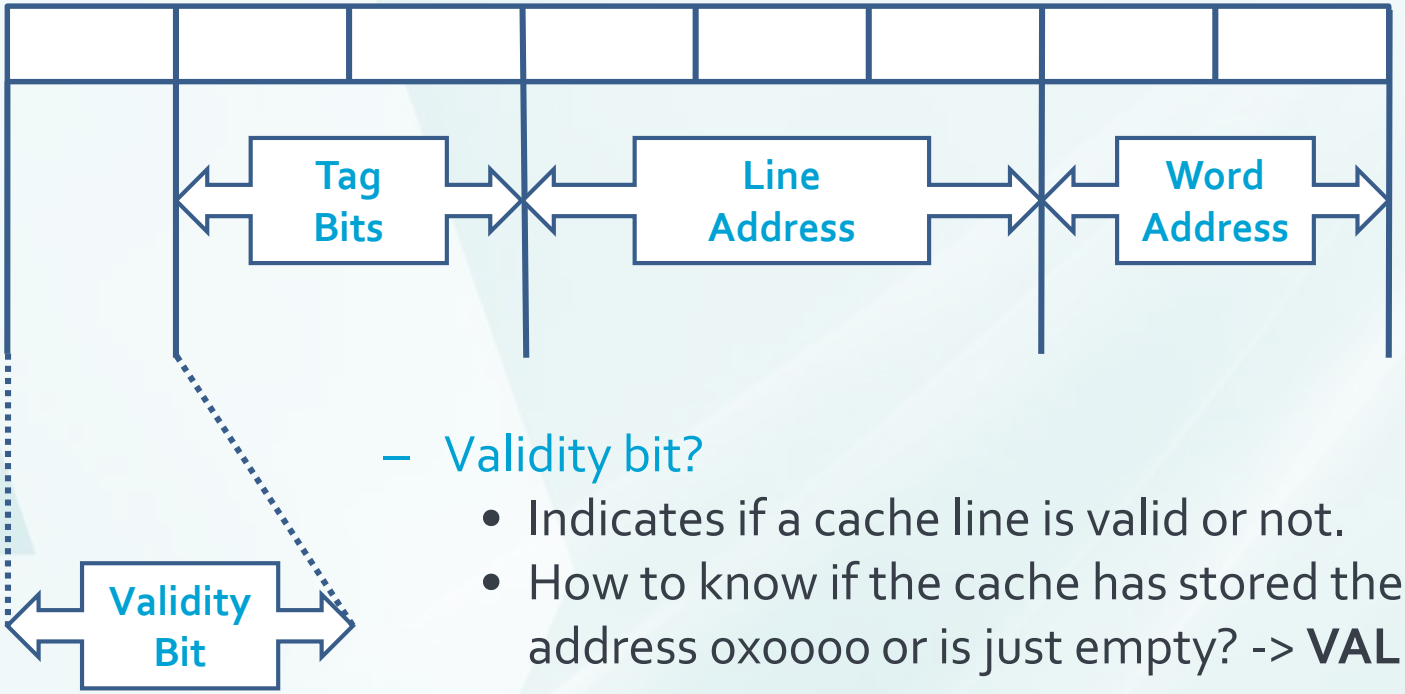


$$\frac{64Bytes}{4 \times 16Bits}$$

WORD

Cache Implementation

- **Direct-mapped** cache:
  - Metrics:
    - Address division:



  - Validity bit?
    - Indicates if a cache line is valid or not.
    - How to know if the cache has stored the address oxoooo or is just empty? -> **VAL bit**

Cache Implementation

- **Dive into ISE:**
  - Create a new project;
  - Add the following sources:
    - memory_hierarchy.v
    - Cache_2way_Associative.v
    - Cache_2way_Associative.v
    - offchip_memory.v
    - rom.v
  - Open the ***memory_hierarchy.v*** and analyze it.
    - Change the type of cache by commenting or no the following define:
      - `define CACHE_2_WAY_ASSOCIATIVE
    - For now keep it comment to try the **direct-mapped** cache
  - You have another source with a simple testbench: cache_test.v
    - Don't forget to change the top module in Implementation tab

Cache Implementation

- **2-way Set Associative** cache:
  - **How to adapt the previous direct-mapped cache into a 2-way set associative cache?**
  - It's simple: a N-way set associative is basically a direct mapped cache with replicated storage resources:
    - N Cache sets
    - N Tag memories
    - N Cache hit comparators
    - …
  - However, there are certain questions that must be answer.
    - Where to store the data coming from memory?
    - What's the criteria to select the set?
      - The one which is empty :)
    - And if both addressed lines in all the sets have content?
      - Replacement mechanism must be employed!

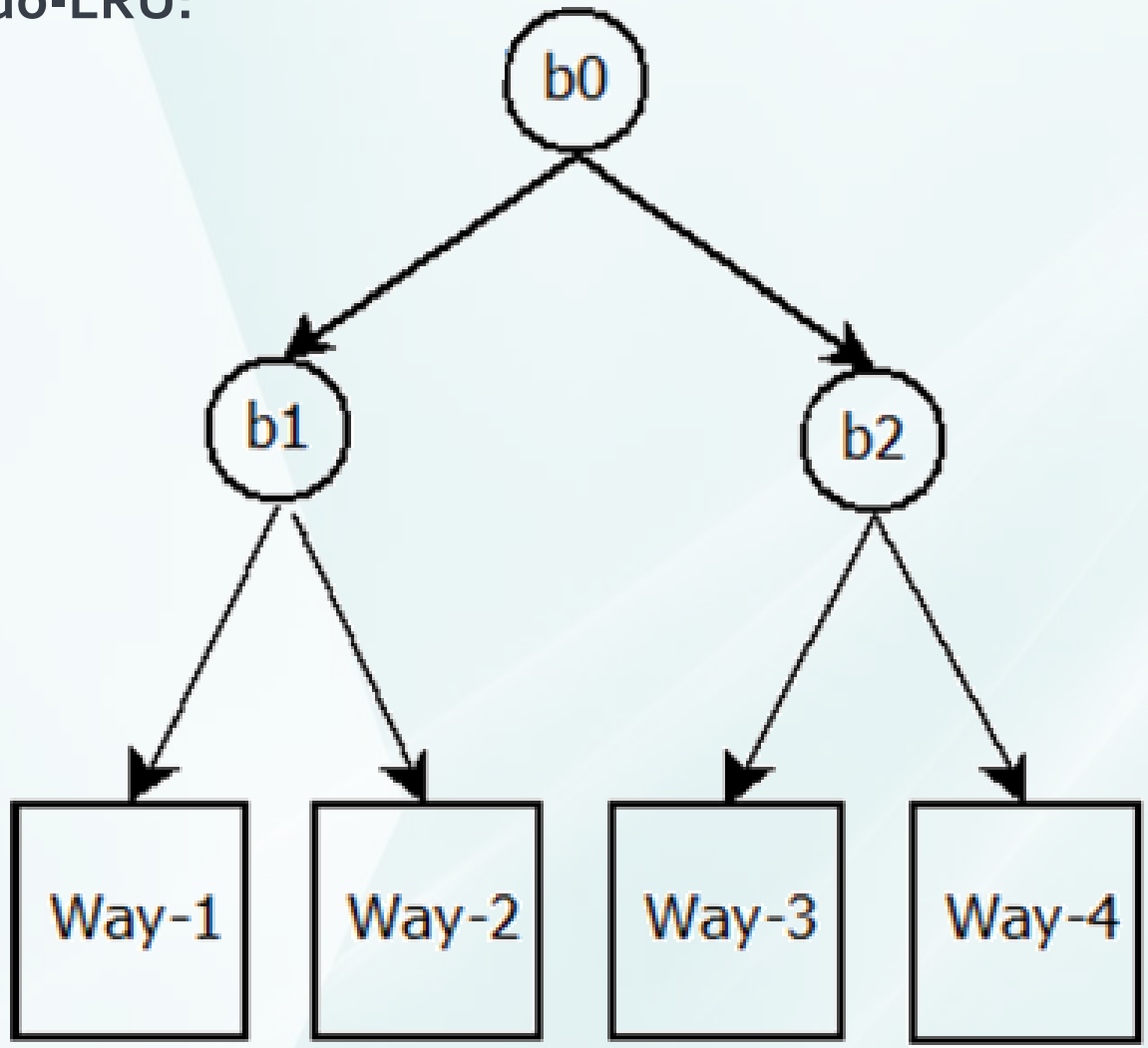
Cache Implementation

- **2-way Set Associative** cache:
  - Replacement mechanism implemented: a derivation of LRU, **pseudo-LRU**
    - One of the **simplest** and **most used**;
    - Easy to implement;
    - Light implementation of the LRU algorithm;
    - It has a great efficiency;
    - Requires few hardware.

- **Pseudo-LRU:**
  - Uses a binary tree to decide which way should be replaced
  - Each binary branch (b0, b1, …) "points the way" to be replaced
    - **–0** means "**go left"**
    - **–1** means **"go right"**
  - And "pointer bits" are updated on every access to point to opposite way of the access

Cache Implementation

- **Pseudo-LRU:**



Cache Implementation

- Try it by yourselves (**TPC**):
  - Create a new Project on ISE
  - Start from the 2-way Set Associative instruction cache and adapt to a **DATA cache**!

Referências

- [1] S. Andrew, S. Dominic, W. Chris, "ARM System Developer's Guide," *Morgan Kaufman Pusblishers*, 2008.

The End