



**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 11 - Jerarquía de Memoria y Memoria Caché**

**Profesor: Germán Leandro Contreras Sagredo**

## Objetivos de la clase

- Conocer lo que es una jerarquía de memoria.
- Entender lo que es una memoria caché y conocer sus funciones de correspondencia, políticas de reemplazo, de escritura y categorizaciones.
- Realizar ejercicios que consoliden los conocimientos anteriores.

## Hasta ahora...

- Ya construimos un computador que tiene todo lo necesario para que pueda ser operado y programado.
- Además, añadimos los elementos necesarios para poder comunicarse con otros tipos de dispositivos en su interior.

Lo que veremos de ahora en adelante serán **mejoras y extensiones sobre la arquitectura existente**.

Comenzaremos viendo mejoras a los **accesos a memoria**.

## Accesos a memoria en el computador básico

Hagamos un pequeño ejercicio.

Tomemos la instrucción **ADD A, (dir)**.

Enumere los **pasos** que realiza el computador básico para ejecutar dicha instrucción.



“Los escucho”

## Accesos a memoria en el computador básico

### Pasos de la ejecución de **ADD A, (dir)**

1. Obtención de la instrucción de la memoria. *Fetch*
2. Decodificación de la instrucción en señales de control. *Decode*
3. Obtención del dato ubicado en dir de la memoria. *Memory*
4. Ejecución de la operación ADD en la ALU. *Execute*
5. Escritura del resultado en el registro A. *Write back*

## Accesos a memoria en el computador básico

### Pasos de la ejecución de **ADD A, (dir)** y su tiempo de ejecución

1. Obtención de la instrucción de la memoria. **50 ns**
2. Decodificación de la instrucción en señales de control. **0.5ns**
3. Obtención del dato ubicado en dir de la memoria. **50 ns**
4. Ejecución de la operación ADD en la ALU. **0.5ns**
5. Escritura del resultado en el registro A. **1 ns**

\* Valores inventados, son para ilustrar.

## Accesos a memoria en el computador básico

El problema de la instrucción anterior es que requiere de tres accesos a memoria, los que suelen ser más costosos en términos de tiempo.

Si bien podemos optimizar nuestro código de distintas formas, los accesos a memoria son inevitables.

Por lo tanto, implementaremos un esquema para agilizar los accesos a memoria basándonos en **principios que utilizamos como humanos al momento de programar**. Estos se conocen como **principios de localidad**.

## Principios de localidad

**Principio de localidad temporal:** “Es probable que un dato obtenido en memoria sea usado **posteriormente** en otra operación”.

```
.data
var1: .word 3
var2: .word 2
res: .word 0
.text
main:
    lw t0, var1
    lw t2, res
    while:
        lw t1, var2
        add t2, t2, t1
        addi t0, t0, -1
        beqz t0, end
    j while
end:
    la t0, res
    sw t2, 0(t0)
```

En este caso, realizamos una lectura de memoria sobre una variable de forma continua a través de iteraciones.

Si bien es posible guardarla como “constante” o, en este caso, hacer la lectura antes de la iteración, existen situaciones en las que no es posible.



## Principios de localidad

**Principio de localidad espacial:** “Es probable que datos **cercanos** al buscado también sean usados (**bloque de memoria**)”.

```
.data
arr: .word 1, 0, 6, 12, 1
len: .word 5
index: .word 0
avg: .word 0
.text
main:
    lw t0, index
    lw t1, len
    lw t2, avg
    la t3, arr
    while:
        beq t0, t1, end
        lw t4, 0(t3)
        add t2, t2, t4
        addi t0, t0, 1
        addi t3, t3, 4
        j while
    end:
        div t2, t2, t1
        la t5, avg
        sw t2, (t5)
```

En este caso, realizamos una lectura de memoria sobre un bloque continuo, en este caso, sobre los datos del arreglo arr.

Trivia: ¿En qué otro contexto tenemos accesos de memoria contiguos?

R: Instrucciones del programa.

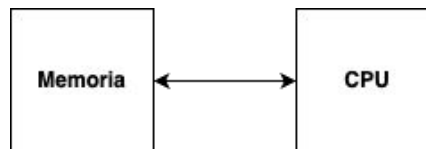
## Accesos a memoria en el computador básico

Podemos aprovechar estos principios modificando la forma en la que organizamos la memoria del computador. Esto nos permitirá **acelerar la ejecución de las instrucciones**.

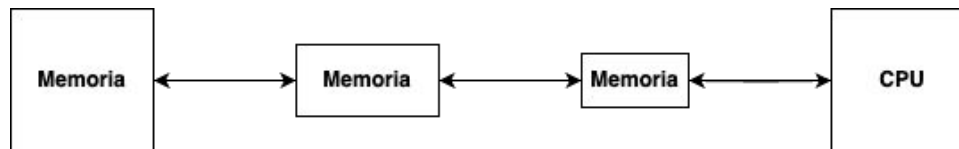
En particular, mejoraremos el rendimiento a través de una **jerarquía de memoria**.

## Jerarquía de memoria

Realizamos una transformación de este esquema:



A este esquema:



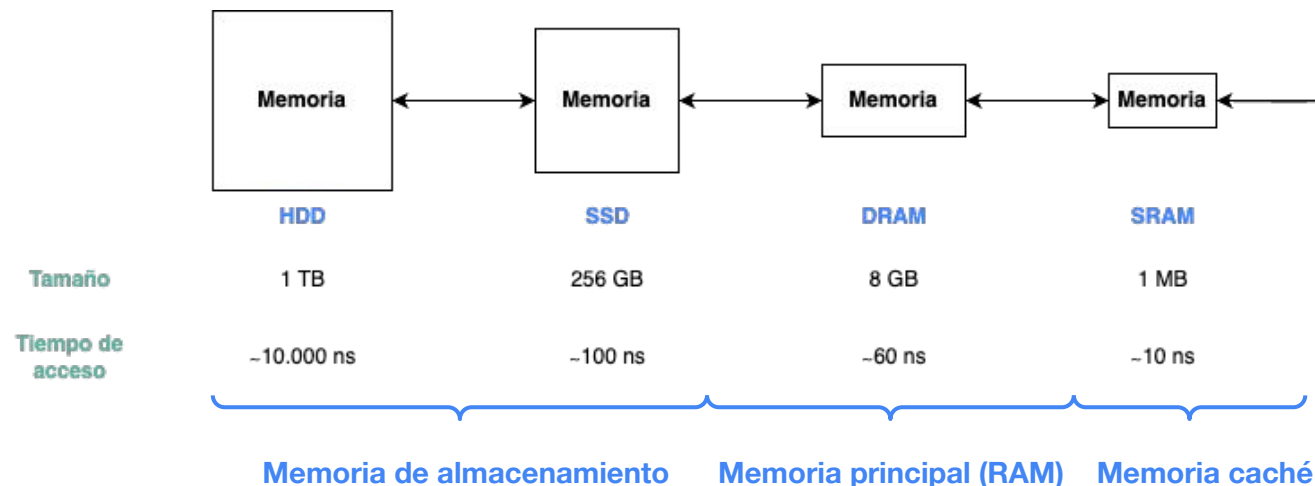
## Jerarquía de memoria

En una jerarquía de memoria, contamos con distintos niveles, cada uno con características distintas.

Los niveles más cercanos a la CPU en la jerarquía suelen poseer menor capacidad de almacenamiento, pero a su vez menores tiempos de lectura y escritura de datos. Esto aprovecha los principios de localidad y, asimismo, **optimiza costos** en unidades de memoria.

La memoria que se conecta directo con la CPU es lo que conocemos como **memoria caché**.

## Jerarquía de memoria - Ejemplo



La cantidad de niveles, velocidades y tamaños de cada memoria dependen netamente del computador construido.

## Jerarquía de memoria - Evaluación

Evaluaremos una jerarquía de memoria con las siguientes métricas:

- **Hit:** Búsqueda exitosa de un dato en la memoria caché.
- **Miss:** Búsqueda fallida de un dato en la memoria caché.
- **Hit-rate (HR):**  $\#Hits \div \#Accesos \text{ a memoria}$
- **Miss-rate (MR):**  $1 - HR$
- **Hit-time (HT):** Tiempo que toma buscar un dato en memoria caché, independiente de si se encuentra o no.
- **Miss-penalty (TP):** Tiempo que toma, luego de un *miss* en caché, copiar un bloque de memoria del siguiente nivel en la jerarquía en la caché y luego acceder al dato buscado desde esta.

## Jerarquía de memoria - Evaluación

De esta manera, a partir de la siguiente fórmula podemos obtener el tiempo de acceso promedio (TP) dentro de una jerarquía de memoria:

Tiempo promedio en caso de *hit*

Tiempo promedio en caso de *miss* \*

$$\begin{aligned} TP &= HR * HT + (1 - HR) * (HT + MP) \\ &= HT + (1 - HR) * MP \end{aligned}$$

Podemos agrupar términos y simplificar la fórmula, como se muestra aquí.

- \* En caso de un *miss*, es importante considerar el *hit-time* dado que la búsqueda de un dato en caché se realiza independiente de si se encuentra o no en ella.

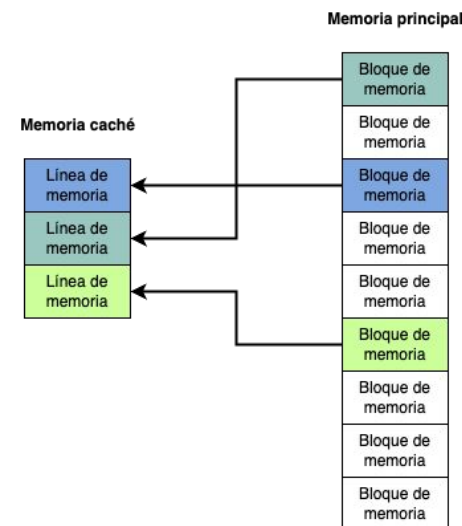
## Memoria caché

- Ocupa el **primer nivel** de la jerarquía de memoria.
- Antes era externa a la CPU, ahora está integrada dentro de ella.
- La CPU **no sabe** que la memoria caché existe. Un **controlador de caché** se encarga de su manejo.
- Se divide en **líneas de memoria** (división física), asociadas a **bloques de memoria** (división lógica) de la memoria principal. Esto aprovecha el **principio de localidad espacial**. Cada línea posee un **bit de validez** que indica si su contenido es válido o no.



## Memoria caché - Bloques y líneas de memoria

- Al acceder a un dato, si este no se encuentra en caché no se obtiene solo este desde la memoria principal, sino que el **bloque contiguo en el que se encuentra ubicado**.
- Las líneas de la caché deben ser del mismo tamaño que los bloques de la memoria principal.



Esquema de asignación de bloques de la memoria principal a líneas de la caché.

Trivia: ¿Podemos tener tantos bloques de memoria como líneas de caché?

R: No, ya que la memoria caché es más pequeña. Si tuvieran el mismo tamaño, se copiaría toda la memoria principal en la caché y la jerarquía carecería de sentido.

## Memoria caché - Controlador de caché

- La CPU **solo conoce** el tamaño de la memoria principal.
- El controlador de caché se encarga de la comunicación y coordinación. Posee dos funciones principales:
  - **Mecanismo de acceso a datos:** Forma en la que el controlador asocia bloques de memoria a línea de caché y cómo se actualizan con cada acceso a datos.
  - **Política de escritura:** Forma en la que el controlador actualiza un dato en la memoria principal si la CPU realiza una escritura.

## Memoria caché - Controlador de caché

- El mecanismo de acceso a datos de un controlador de caché se define, a su vez, a partir de dos elementos:
  - **Función de correspondencia:** Asociación entre una línea de caché y un bloque de memoria.
  - **Política de reemplazo:** Forma en la que se sobrescribe una línea de caché en caso de que no exista disponibilidad para una copia de datos, aprovechando el **principio de localidad temporal**.

## Funciones de correspondencia

La fórmula de asociación entre un bloque de memoria y una línea de la caché dependerá de los siguientes factores:

- Tamaño de línea de caché.
- Cantidad de líneas de caché.
- Cantidad de direcciones de la memoria principal (tamaño).

Por simplicidad, siempre serán potencias de 2.

Además, en cada función de correspondencia se utilizará un **tag**.


## Funciones de correspondencia - *Tag*

El *tag* corresponde a un identificador “único” que nos ayudará a determinar si el dato de una dirección de la memoria principal se encuentra efectivamente almacenado en la caché o no. Este es el dato que revisamos al realizar las búsquedas.

La composición del *tag* se obtiene a través de los bits de la dirección de memoria y su formato final dependerá del tipo de función de correspondencia a utilizar.

Con eso en mente, veremos a continuación tres tipos de función de correspondencia.

## Funciones de correspondencia - *Directly mapped*

- Función de correspondencia más simple. Cada bloque de la memoria principal se asocia **solo a una línea de la caché**.
- Cada línea de la caché poseerá un **índice**. Para obtener su valor a partir de la dirección de un bloque, se hace uso de la siguiente fórmula:  $\text{Núm. línea} = \text{Núm. bloque} \bmod \text{\#Líneas}$
- **Ejemplo:** 4 bloques y 2 líneas. 
  - Bloque 0  $\rightarrow 0 \bmod 2 = 0 \rightarrow$  Línea 0
  - Bloque 1  $\rightarrow 1 \bmod 2 = 1 \rightarrow$  Línea 1
  - Bloque 2  $\rightarrow 2 \bmod 2 = 0 \rightarrow$  Línea 0
  - Bloque 3  $\rightarrow 3 \bmod 2 = 1 \rightarrow$  Línea 1

## Funciones de correspondencia - *Directly mapped*

Para determinar la línea de caché en la que se *podría* encontrar una dirección de memoria, seguimos los siguientes pasos:

1. Expresamos la dirección de memoria en binario. Usamos los bits necesarios para direccionar **toda** la memoria principal.
2. Obtenemos la cantidad de bits necesaria para direccionar una palabra **dentro** de una línea. Usamos los bits menos significativos de la dirección para esta asociación.
3. Obtenemos la cantidad de bits necesaria para el **índice** de línea. Usamos los siguientes bits menos significativos para la asociación.
4. Usamos los bits restantes de la dirección como **tag**, el que nos permitirá ver si el dato buscado está efectivamente almacenado en su línea de caché.

## Funciones de correspondencia - *Directly mapped*

### Ejemplo práctico

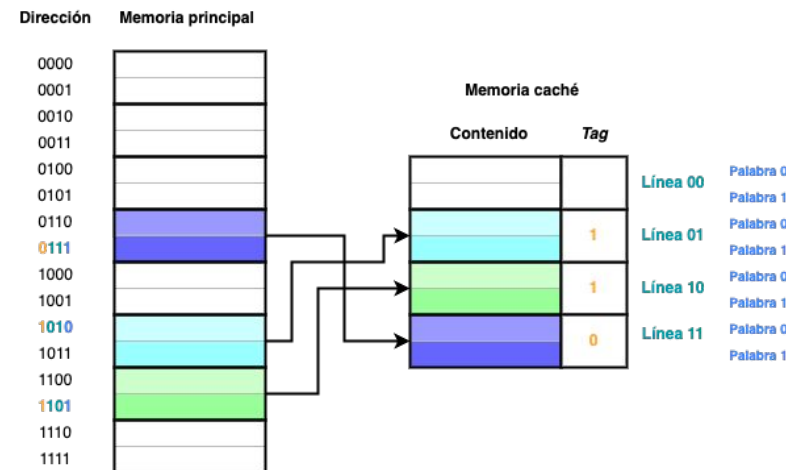
- Memoria principal de 16 bytes  $\rightarrow 2^4$  bytes  $\rightarrow$  **4 bits de direccionamiento**
- Memoria caché de 8 bytes y 4 líneas
  - $8 \div 4$  bytes/línea = 2 bytes/línea =  $2^1$  palabras por línea  
 $\rightarrow$  **1 bit para direccionar una palabra dentro de una línea**
  - 4 líneas =  $2^2$  líneas  $\rightarrow$  **2 bits de índice de línea**
  - 1 bit sin uso de la dirección  $\rightarrow$  **1 bit de tag**
- Ejemplos de direcciones de memoria
  - 7: **0111**      Palabra = 1    Línea = 11    Tag = 0
  - 10: **1010**      Palabra = 0    Línea = 01    Tag = 1
  - 13: **1101**      Palabra = 1    Línea = 10    Tag = 1



## Funciones de correspondencia - *Directly mapped*

### Ejemplo práctico

- Aunque se acceda a una sola dirección del bloque, se copia por completo en la línea de caché correspondiente.
- Dos bloques de memoria pueden asociarse a la misma línea, pero el *tag* nos permite diferenciarlas.
- Por construcción, **nunca existirán** dos bloques que pertenezcan a la misma línea y que posean el mismo *tag*.

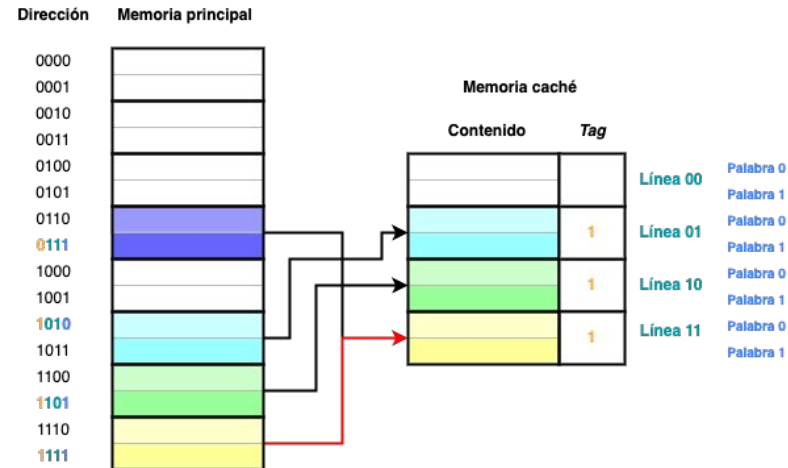


Asociación resultante del ejemplo anterior.

## Funciones de correspondencia - *Directly mapped*

### Ejemplo práctico

- Si se agrega un acceso adicional que apunta a una línea de caché ya ocupada, **su contenido será sobrescrito**. Por ejemplo, las direcciones 7 (0**111**) y 15 (1**111**).
- Se actualiza tanto el contenido como el *tag* para poder distinguir entre ambas direcciones.



Asociación resultante si hubiera un acceso adicional con una línea ocupada.

## Funciones de correspondencia - *Directly mapped*

### Ventajas

- Cómputo sencillo y directo.

### Desventajas

- Retención entre bloques y líneas.
- Desaprovechamiento del resto de la caché.

Veremos otros esquemas que resuelven esto **relajando la asociación fija entre un bloque de memoria y una línea de la caché.**

## Funciones de correspondencia - *Fully associative*

- En este esquema, cada bloque de memoria puede asociarse a **cualquier línea de la caché**.
- Aprovecha todo el espacio de la memoria y, por ende, **maximiza el *hit-rate***.
- Como se puede ocupar cualquier línea, la composición del *tag* para identificar si un dato se encuentra en caché se complejiza.

## Funciones de correspondencia - *Fully associative*

Para determinar la línea de caché en la que se *podría* encontrar una dirección de memoria, seguimos los siguientes pasos:

1. Expresamos la dirección de memoria en binario. Usamos los bits necesarios para direccionar **toda** la memoria principal.
2. Obtenemos la cantidad de bits necesaria para direccionar una palabra **dentro** de una línea. Usamos los bits menos significativos de la dirección para esta asociación.
3. Usamos los bits restantes de la dirección como **tag**. Dado que se puede usar *cualquier* línea, es necesario recorrer toda la caché y sus *tags* para ver si existe coincidencia y, así, verificar si hay *hit* o *miss*.

## Funciones de correspondencia - *Fully associative*

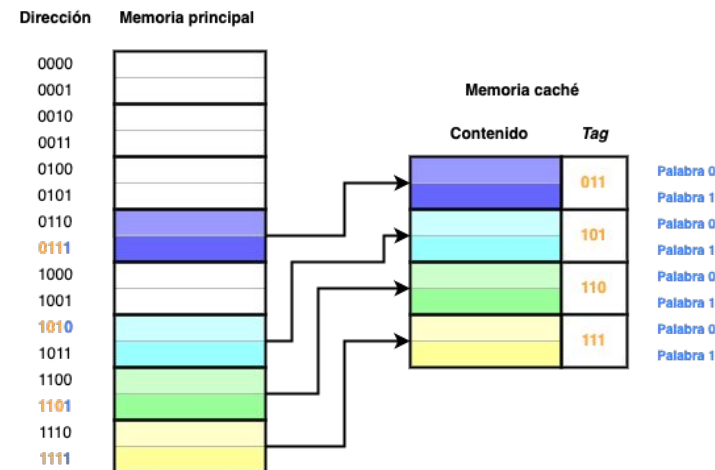
### Ejemplo práctico

- Memoria principal de 16 bytes  $\rightarrow 2^4$  bytes  $\rightarrow$  **4 bits de direccionamiento**
- Memoria caché de 8 bytes y 4 líneas
  - $8 \div 4$  bytes/línea = 2 bytes/línea =  $2^1$  palabras por línea  
 $\rightarrow$  **1 bit para direccionar una palabra dentro de una línea**
  - 3 bits sin uso de la dirección  $\rightarrow$  **3 bits de tag**
- Ejemplos de direcciones de memoria
  - 7: **0111**      **Palabra = 1**    **Tag = 011**
  - 10: **1010**      **Palabra = 0**    **Tag = 101**
  - 13: **1101**      **Palabra = 1**    **Tag = 110**
  - 15: **1111**      **Palabra = 1**    **Tag = 111**

## Funciones de correspondencia - *Fully associative*

### Ejemplo práctico

- Si accedemos a las direcciones en el orden 7, 10, 13 y 15, tendremos las asociaciones señaladas en la figura.
- Cuando se accede a una dirección que no está en caché, se copia el bloque a la **primera línea disponible según el bit de validez**.



Asociación resultante del ejemplo anterior.

## Funciones de correspondencia - *Fully associative*

### Ventajas

- Se ocupa toda la caché, aprovechándola al máximo.

### Desventajas

- El *hit-time* es significativamente mayor al tener que buscar coincidencia de *tag* en todas las líneas.
- El almacenamiento del *tag* crece sustancialmente.

¿Podemos mejorar esto de **alguna** forma?





## Funciones de correspondencia - *N-way associative*

- En este esquema, cada bloque de memoria puede asociarse a **un conjunto de líneas de la caché**.
- Divide a la caché en conjuntos de  $N$  líneas. **No son  $N$  conjuntos**.
- Lo mejor de los dos mundos:
  - Cada bloque posee un **mapeo directo** a un conjunto.
  - Dentro de un conjunto, un bloque se asocia a **cualquier línea**.



## Funciones de correspondencia - *N-way associative*

Para determinar la línea de caché en la que se *podría* encontrar una dirección de memoria, seguimos los siguientes pasos:

1. Expresamos la dirección de memoria en binario. Usamos los bits necesarios para direccionar **toda** la memoria principal.
2. Obtenemos la cantidad de bits necesaria para direccionar una palabra **dentro** de una línea. Usamos los bits menos significativos de la dirección para esta asociación.
3. Obtenemos la cantidad de bits necesaria para el **conjunto** de líneas. Usamos los siguientes bits menos significativos para la asociación.
4. Usamos los bits restantes de la dirección como **tag**, el que nos permitirá ver si el dato buscado está almacenado en una línea de caché en su conjunto.

## Funciones de correspondencia - *N-way associative*

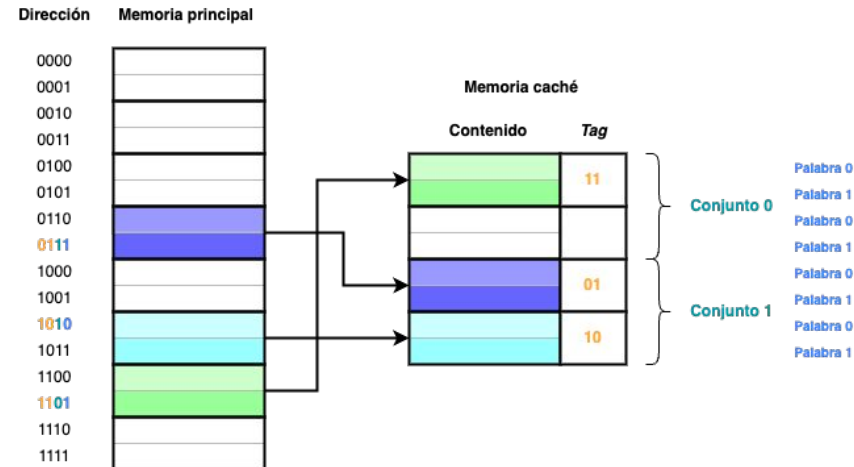
### Ejemplo práctico

- Memoria principal de 16 bytes  $\rightarrow 2^4$  bytes  $\rightarrow$  **4 bits de direccionamiento**
- Memoria caché de 8 bytes, 4 líneas y conjuntos de 2 líneas (*2-way associative*)
  - $8 \div 4$  bytes/línea = 2 bytes/línea =  $2^1$  palabras por línea  
 $\rightarrow$  **1 bit para direccionar una palabra dentro de una línea**
  - $4 \div 2$  líneas/conjunto = 2 líneas/conjunto =  $2^1$  líneas por conjunto  
 $\rightarrow$  **1 bit de índice de conjunto**
  - 2 bits sin uso de la dirección  $\rightarrow$  **2 bits de tag**
- Ejemplos de direcciones de memoria
  - 7:    **0111**      Palabra = 1      Conjunto = 1      Tag = 01
  - 10:   **1010**      Palabra = 0      Conjunto = 1      Tag = 10
  - 13:   **1101**      Palabra = 1      Conjunto = 0      Tag = 11

## Funciones de correspondencia - *N-way associative*

### Ejemplo práctico

- Si accedemos a las direcciones en el orden 7, 10 y 13, tendremos las asociaciones señaladas en la figura.
- Cuando se accede a una dirección que no está en caché, se copia el bloque a la **primera línea disponible dentro de un conjunto según el bit de validez.**



Asociación resultante del ejemplo anterior.

## Funciones de correspondencia - *N-way associative*

### Ventajas

- *Hit-time* menor respecto a la función *fully associative*.
- Mayor uso de caché respecto a *directly mapped*.
- Ocupa menos bits de *tag*.

### Desventajas

- *Hit-time* mayor respecto a *directly mapped*.
- Menor uso de caché respecto a *fully associative*.

Si sopesamos respecto a las funciones anteriores, de todas formas es la más equilibrada.

## Funciones de correspondencia - Obtención del *tag* e índices

¿Por qué obtenemos el *offset*, índices y *tags* de derecha a izquierda?

Tomemos una memoria de 16 bytes y una caché de 8 bytes y 4 líneas.

Tenemos  $2^4$  direcciones de memoria distintas y  $2^1$  direcciones por bloque. De esta forma:

$$\begin{aligned} 2^4 \text{ direcciones} \div 2^1 \text{ direcciones/bloque} \\ = 2^3 \text{ bloques} \end{aligned}$$

Esto es equivalente a realizar un *shift right* sobre la dirección de memoria:

$b_3b_2b_1b_0 \rightarrow$   $b_3b_2b_1$ : Dirección de un bloque

$b_0$ : Dirección de una palabra en un bloque

## Funciones de correspondencia - Obtención del *tag* e índices

Ahora, lo que hacemos sobre los bits restantes depende del tipo de función de correspondencia que utilizamos.

**Fully associative:** Como todo bloque puede ser asignado a cualquier línea de la caché, tenemos:  $2^3 \text{ bloques} \div 2^0 \text{ líneas} = 2^3 \text{ bloques/línea}$   
→ Requerimos 3 bits para **identificar un bloque por línea**, por lo que  **$b_3b_2b_1$  es el *tag* requerido**. No se realizan *shift rights* adicionales.

## Funciones de correspondencia - Obtención del *tag* e índices

**Directly mapped:** Como todo bloque puede ser asignado solo a una línea de memoria, tenemos:  $2^3 \text{ bloques} \div 2^2 \text{ líneas} = 2^1 \text{ bloques/línea}$   
Esta división corresponde a dos *shift right*:

$b_3b_2b_1 \rightarrow b_2b_1$ : Dirección de la línea a la que pertenece el bloque.  
 $b_3$ : *Tag* resultante que identifica un bloque en una línea.

Esto significa que solo existen dos bloques asignables por línea, por lo que basta un bit para identificarlos. Esta construcción nos asegura que nunca existan dos bloques con mismo *tag* e índice de línea.



## Funciones de correspondencia - Obtención del *tag* e índices

***N-way associative***: Como un bloque puede ser asignado a cualquier línea dentro de un conjunto, si tenemos 2 conjuntos, por ejemplo:

$$2^3 \text{ bloques} \div 2^1 \text{ conjuntos} = 2^2 \text{ bloques/conjunto}$$

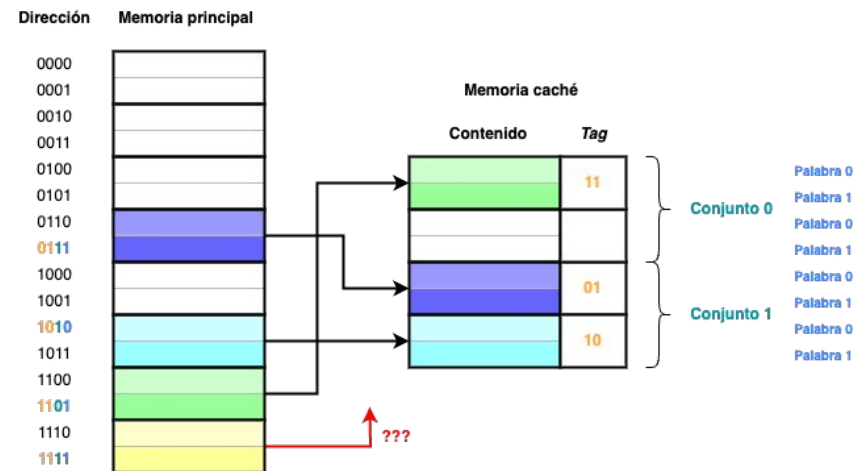
Esta división corresponde a un solo *shift right*:

$b_3 b_2 b_1 \rightarrow$   **$b_1$ : Dirección del conjunto al que pertenece el bloque.**  
 **$b_3 b_2$ : Tag resultante que identifica un bloque en un conjunto.**

Esto implica que solo hay cuatro bloques asignables por línea en un conjunto, requiriendo 2 bits para identificarlos. Esto asegura que nunca existan dos bloques con mismo *tag* e índice de conjunto.

## Funciones de correspondencia - Reemplazo de líneas

- Por conveniencia, en las funciones asociativas hicimos caso omiso de las situaciones en las que la memoria caché o un conjunto de líneas de esta están llenas.
- En estos casos, es necesario realizar un reemplazo, pero ¿cómo lo hacemos?



Si ahora tratamos de acceder a la dirección 15, esta apunta al conjunto 1 que se encuentra lleno. ¿Dónde sobrescribimos el bloque?

## Políticas de reemplazo

- **Bélády:** Se saca el bloque que se utilizará más lejos en el futuro. **Ideal no alcanzable en la práctica.**
- **First-In First-Out (FIFO):** El primer bloque en entrar es el primero en salir. Simple de implementar, pero ingenuo.
- **Least Frequently Used (LFU):** El bloque con menos accesos se saca. Mejor que FIFO, pero más complejo de implementar por su contador de accesos.
- **Least Recently Used (LRU):** El bloque con mayor tiempo sin accesos se saca. Complejo por requerir una medida de “tiempo” en bits, pero el mejor en la práctica.
- **Random:** Rápido, con rendimiento inferior a LFU y LRU; pero mejor que FIFO.

## Políticas de escritura

Hasta ahora nos hemos enfocado en acceso a datos, pero un programa también puede **escribir datos en memoria**.

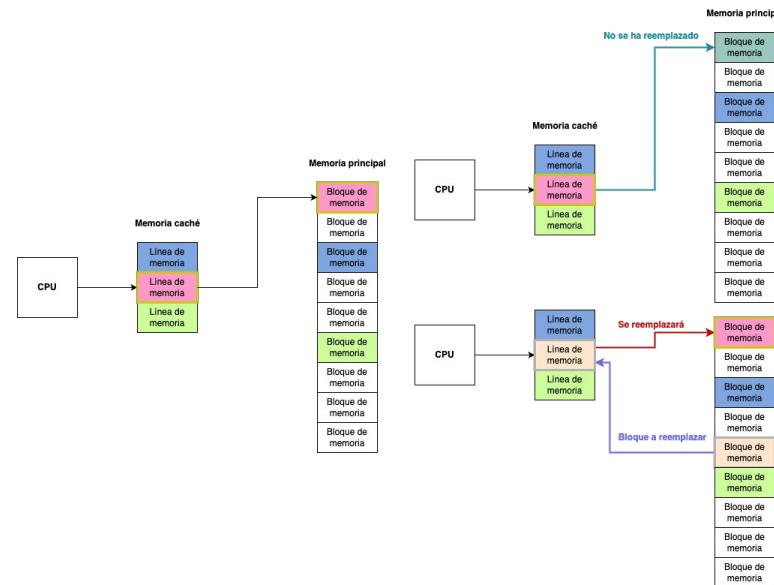
¿Cómo se hace cargo el controlador de caché en este caso? Puede hacer uso de **dos políticas de escritura**:

- **Write-through:** El bloque modificado se escribe inmediatamente en la memoria principal.
- **Write-back:** El bloque modificado se escribe en la memoria principal **solo cuando su línea en caché será sustituida**.

## Políticas de escritura

- *Write-through* es menos conveniente por el tiempo adicional que implica cada instrucción de escritura.
- *Write-back* mejora los tiempos, pero otros dispositivos (DMA) o núcleos **pueden acceder a estas direcciones desde la memoria principal**, lo que genera un problema de **sincronización**.\*

\* Más adelante veremos cómo abordarlo.



Ejemplo de *write-through* vs. *write-back*.

## Memoria caché - Tipos de memoria

Hasta ahora, no hemos hecho la distinción entre las arquitecturas Harvard y Von Neumann. ¿Qué relevancia tiene esto?

- Si utilizamos Von Neumann, tenemos en una misma memoria los datos e instrucciones de nuestros programas.
- ¿Es la distribución de accesos a memoria similar entre los datos e instrucciones?

...**No**...

## Memoria caché - Tipos de memoria

De este hecho surge la necesidad de tener dos tipos de memoria caché:

- **Caché unified:** Almacena datos e instrucciones. Cuenta con la desventaja de que probablemente su *hit-rate* sea inferior al mezclar dos distribuciones de acceso distintas.
- **Caché split:** Caché con división interna para datos e instrucciones. Puede tener mejor *hit-rate*, pero su *hardware* es más complejo y, por ende, posee mayor *overhead* de acceso.

## Memoria caché - Tipos de memoria

Pregunta capciosa: ¿Cuáles son las ventajas de tener una memoria caché *split* en vez de una *unified* conectada a la memoria de datos del computador básico?

**Ninguna.** La caché *split* solo tiene sentido bajo una arquitectura Von Neumann con memoria compartida entre datos e instrucciones. En la memoria de datos solo tiene sentido usar una memoria *unified* por este motivo.





## Memoria caché - Caché multinivel

Así como aplicamos la idea de jerarquía a la memoria, podemos hacer lo mismo con la memoria caché: definir una jerarquía de caché con varios niveles, cuyos tamaños son sucesivamente mayores.

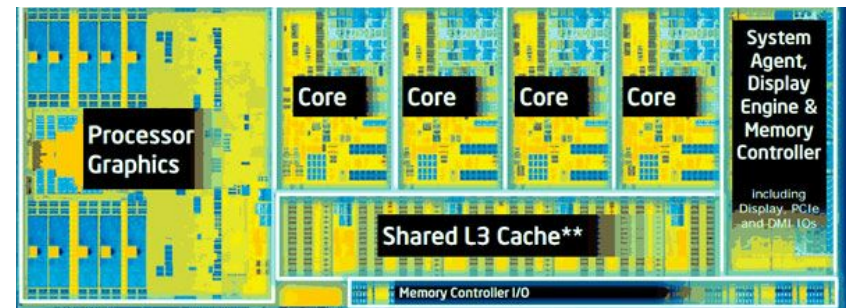
En la práctica, suelen existir tres niveles: L1, L2, L3.

En general, para balancear el rendimiento, las cachés L1 suelen ser de tipo *split* por ser la sección más crítica del rendimiento, mientras que las L2 y L3 suelen ser *unified* para facilitar la sincronización de su contenido con otros dispositivos y núcleos en el sistema.

## Memoria caché - Caché multinivel

### Ejemplo real: Intel Haswell Core i5/i7

- Chip con 4 núcleos (*cores*).
- Cada núcleo posee cachés L1 y L2.
- Todos los núcleos usan una caché compartida L3.



Composición de un chip Intel Haswell.

## Memoria caché - Ejercicio en clases

Suponga que tiene una memoria principal de 16 bytes y una memoria caché de 8 bytes y 4 líneas. Además, asuma que tiene un programa que accede, en este orden, a las direcciones de la memoria principal: 0, 1, 5, 7, 10, 13, 4, 6.

Obtenga el estado final de la memoria caché (en una tabla) y el *hit-rate* para cada una de las siguientes funciones de correspondencia: *directly mapped*, *fully associative*, *2-way associative*. Puede asumir una política de reemplazo LRU, en caso de necesitarla.

## Memoria caché - Ejercicio en clases

Antes de partir, algunos elementos que tendremos en común en todos los casos:

- Memoria de 16 bytes  $\rightarrow 2^4$  direcciones, **4 bits de dirección.**
- Caché de 8 bytes y 4 líneas  $\rightarrow 2^1$  palabras por línea, **1 bit para direccionar una palabra en una línea (*offset*).**

Veremos, para cada caso, cómo se va llenando la caché.

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Bits de *offset* de palabra: 1 bit.**

**Bits de índice de línea: 4 líneas  $\rightarrow 2^2$  líneas  $\rightarrow$  2 bits.**

**Bits de *tag*: 4 - 2 (bits índice de línea) - 1 (*offset*) = 1 bit.**

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

Veremos la forma en la que se va llenando el contenido de la caché a través de la tabla.

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	0	-	-
01	0 1	0	-	-
10	0 1	0	-	-
11	0 1	0	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

Accesos: 0, 1, 5, 7, 10, 13, 4, 6

Dirección actual: 0 = 0000

Offset palabra: 0

Índice de línea: 00

Tag: 0

¡Miss! HR: 0/1

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	0 → 1	- → 0	Mem[0] Mem[1]
01	0 1	0	-	-
10	0 1	0	-	-
11	0 1	0	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 1 = 0001

**Offset palabra:** 1

**Índice de línea:** 00

**Tag:** 0

**¡Hit!** HR: 1/2

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	0	-	-
10	0 1	0	-	-
11	0 1	0	-	-

Contenido de la caché



## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 5 = 0101

**Offset palabra:** 1

**Índice de línea:** 10

**Tag:** 0

**¡Miss! HR:** 1/3

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	0	-	-
10	0 1	0 → 1	- → 0	Mem[4] Mem[5]
11	0 1	0	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 7 = 0111

**Offset palabra:** 1

**Índice de línea:** 11

**Tag:** 0

**¡Miss!** HR: 1/4

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	0	-	-
10	0 1	1	0	Mem[4] Mem[5]
11	0 1	0 → 1	- → 0	Mem[6] Mem[7]

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, **10**, 13, 4, 6

**Dirección actual:** 10 = **1010**

**Offset palabra:** 0

**Índice de línea:** 01

**Tag:** 1

**¡Miss!** HR: 1/5

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	0 → 1	- → 1	Mem[10] Mem[11]
10	0 1	1	0	Mem[4] Mem[5]
11	0 1	1	0	Mem[6] Mem[7]

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, **13**, 4, 6

**Dirección actual:** 13 = **1101**

**Offset palabra:** 1

**Índice de línea:** 10

**Tag:** 1

**¡Miss!** HR: 1/6

Se reemplaza la línea 10 con el nuevo acceso

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	1	1	Mem[10] Mem[11]
10	0 1	1	0 → 1	Mem[12] Mem[13]
11	0 1	1	0	Mem[6] Mem[7]

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 4 = 0100

**Offset palabra:** 0

**Índice de línea:** 10

**Tag:** 0

**¡Miss!** HR: 1/7

Se reemplaza la línea 10 con el nuevo acceso

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	1	1	Mem[10] Mem[11]
10	0 1	1	1 → 0	Mem[4] Mem[5]
11	0 1	1	0	Mem[6] Mem[7]

Contenido de la caché

## Memoria caché - Ejercicio en clases

### *Directly mapped*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, **6**

**Dirección actual:** 6 = **0110**

**Offset palabra:** 0

**Índice de línea:** 10

**Tag:** 0

Índice línea	Offset palabra	Bit validez	Tag	Datos
00	0 1	1	0	Mem[0] Mem[1]
01	0 1	1	1	Mem[10] Mem[11]
10	0 1	1	0	Mem[4] Mem[5]
11	0 1	1	0	Mem[6] Mem[7]

Contenido de la caché

**¡Hit!** HR: 2/8 → HR final igual a 0.25 o 25%

## Memoria caché - Ejercicio en clases

***Fully associative***

**Bits de *offset* de palabra: 1 bit.**

**Bits de *tag*:  $4 - 1$  (*offset*) = 3 bits.**

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

Veremos la forma en la que se va llenando el contenido de la caché a través de la tabla.

Incluimos el tiempo de acceso para saber qué línea reemplazar.

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	0	-	-	-
01	0 1	0	-	-	-
10	0 1	0	-	-	-
11	0 1	0	-	-	-

Contenido de la caché



## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 0 = 0000

**Offset palabra:** 0

**Tag:** 000

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	0 → 1	- → 000	Mem[0] Mem[1]	- → 0
01	0 1	0	-	-	-
10	0 1	0	-	-	-
11	0 1	0	-	-	-

Contenido de la caché

**¡Miss!** HR: 0/1

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 1 = 0001

**Offset palabra:** 1

**Tag:** 000

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	000	Mem[0] Mem[1]	0
01	0 1	0	-	-	-
10	0 1	0	-	-	-
11	0 1	0	-	-	-

Contenido de la caché

¡Hit! HR: 1/2

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, **5**, 7, 10, 13, 4, 6

**Dirección actual:** 5 = **0101**

**Offset palabra:** **1**

**Tag:** **010**

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	000	Mem[0] Mem[1]	0 → 1
01	0 1	0 → 1	- → 010	Mem[4] Mem[5]	- → 0
10	0 1	0	-	-	-
11	0 1	0	-	-	-

Contenido de la caché

**¡Miss!** HR: 1/3

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 7 = 0111

**Offset palabra:** 1

**Tag:** 011

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	000	Mem[0] Mem[1]	1 → 2
01	0 1	1	010	Mem[4] Mem[5]	0 → 1
10	0 1	0 → 1	- → 011	Mem[6] Mem[7]	- → 0
11	0 1	0	-	-	-

Contenido de la caché

**¡Miss!** HR: 1/4

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, **10**, 13, 4, 6

**Dirección actual:** 10 = **1010**

**Offset palabra:** 0

**Tag:** **101**

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	000	Mem[0] Mem[1]	2 → 3
01	0 1	1	010	Mem[4] Mem[5]	1 → 2
10	0 1	1	011	Mem[6] Mem[7]	0 → 1
11	0 1	0 → 1	- → 101	Mem[10] Mem[11]	- → 0

Contenido de la caché

**¡Miss!** HR: 1/5

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, **13**, 4, 6

**Dirección actual:** 13 = **1101**

**Offset palabra:** 1

**Tag:** **110**

Se reemplaza la línea 00 por ser accedida hace más tiempo

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	000 → 110	Mem[12] Mem[13]	3 → 0
01	0 1	1	010	Mem[4] Mem[5]	2 → 3
10	0 1	1	011	Mem[6] Mem[7]	1 → 2
11	0 1	1	101	Mem[10] Mem[11]	0 → 1

Contenido de la caché

**¡Miss!** HR: 1/6

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 4 = 0100

**Offset palabra:** 0

**Tag:** 010

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	110	Mem[12] Mem[13]	0 → 1
01	0 1	1	010	Mem[4] Mem[5]	3 → 0
10	0 1	1	011	Mem[6] Mem[7]	2 → 3
11	0 1	1	101	Mem[10] Mem[11]	1 → 2

Contenido de la caché

¡Hit! HR: 2/7

## Memoria caché - Ejercicio en clases

### *Fully associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, **6**

**Dirección actual:** 6 = **0110**

**Offset palabra:** 0

**Tag:** **011**

Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
00	0 1	1	110	Mem[12] Mem[13]	1 → 2
01	0 1	1	010	Mem[4] Mem[5]	0 → 1
10	0 1	1	011	Mem[6] Mem[7]	3 → 0
11	0 1	1	101	Mem[10] Mem[11]	2 → 3

Contenido de la caché

**¡Hit!** HR: 3/8 → HR final igual a 0.375 o 37.5%



## Memoria caché - Ejercicio en clases

### ***2-way associative***

**Bits de *offset* de palabra: 1 bit.**

**Bits de índice de conjunto:**

4 líneas  $\div$  2 líneas/conjunto  $\rightarrow 2^1$  conjuntos = **1 bit.**

**Bits de *tag*:  $4 - 1$  (bit índice de conjunto) - 1 (*offset*) = 2 bits.**

## Memoria caché - Ejercicio en clases

### *2-way associative*

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

Veremos la forma en la que se va llenando el contenido de la caché a través de la tabla.

Incluimos el tiempo de acceso para saber qué línea reemplazar y, a su vez, el índice de conjunto.

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	0	-	-	-
	01	0 1	0	-	-	-
1	10	0 1	0	-	-	-
	11	0 1	0	-	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

Accesos: 0, 1, 5, 7, 10, 13, 4, 6

Dirección actual: 0 = 0000

Offset palabra: 0

Índice de conjunto: 0

Tag: 00

¡Miss! HR: 0/1

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	0 → 1	- → 00	Mem[0] Mem[1]	- → 0
	01	0 1	0	-	-	-
1	10	0 1	0	-	-	-
	11	0 1	0	-	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 1 = 0001

**Offset palabra:** 1

**Índice de conjunto:** 0

**Tag:** 00

**¡Hit!** HR: 1/2

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	00	Mem[0] Mem[1]	0
	01	0 1	0	-	-	-
1	10	0 1	0	-	-	-
	11	0 1	0	-	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

Accesos: 0, 1, 5, 7, 10, 13, 4, 6

Dirección actual: 5 = 0101

Offset palabra: 1

Índice de conjunto: 0

Tag: 01

¡Miss! HR: 1/3

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	00	Mem[0] Mem[1]	0 → 1
	01	0 1	0 → 1	- → 01	Mem[4] Mem[5]	- → 0
1	10	0 1	0	-	-	-
	11	0 1	0	-	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 7 = 0111

**Offset palabra:** 1

**Índice de conjunto:** 1

**Tag:** 01

**¡Miss! HR:** 1/4

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	00	Mem[0] Mem[1]	1
	01	0 1	1	01	Mem[4] Mem[5]	0
1	10	0 1	0 → 1	- → 01	Mem[6] Mem[7]	- → 0
	11	0 1	0	-	-	-

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

Accesos: 0, 1, 5, 7, **10**, 13, 4, 6

Dirección actual: 10 = **1010**

Offset palabra: 0

Índice de conjunto: 1

Tag: **10**

**¡Miss!** HR: 1/5

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	00	Mem[0] Mem[1]	1
	01	0 1	1	01	Mem[4] Mem[5]	0
1	10	0 1	1	01	Mem[6] Mem[7]	0 → 1
	11	0 1	0 → 1	- → 10	Mem[10] Mem[11]	- → 0

Contenido de la caché

# Memoria caché - Ejercicio en clases

## 2-way associative

Accesos: 0, 1, 5, 7, 10, **13**, 4, 6

Dirección actual: 13 = **1101**

Offset palabra: **1**

Índice de conjunto: **0**

Tag: **11**

**¡Miss!** HR: 1/6

Se reemplaza la línea 00 por ser accedida hace más tiempo en el conjunto

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	00 → 11	Mem[12] Mem[13]	1 → 0
	01	0 1	1	01	Mem[4] Mem[5]	0 → 1
1	10	0 1	1	01	Mem[6] Mem[7]	1
	11	0 1	1	10	Mem[10] Mem[11]	0

Contenido de la caché



## Memoria caché - Ejercicio en clases

### 2-way associative

**Accesos:** 0, 1, 5, 7, 10, 13, 4, 6

**Dirección actual:** 4 = 0100

**Offset palabra:** 0

**Índice de conjunto:** 0

**Tag:** 01

¡Hit! HR: 2/7

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	11	Mem[12] Mem[13]	0 → 1
	01	0 1	1	01	Mem[4] Mem[5]	1 → 0
1	10	0 1	1	01	Mem[6] Mem[7]	1
	11	0 1	1	10	Mem[10] Mem[11]	0

Contenido de la caché

## Memoria caché - Ejercicio en clases

### 2-way associative

**Accesos:** 0, 1, 5, 7, 10, 13, 4, **6**

**Dirección actual:** 6 = **0110**

**Offset palabra:** 0

**Índice de conjunto:** 1

**Tag:** 01

Índice conjunto	Índice línea	Offset palabra	Bit validez	Tag	Datos	Tiempo Acceso
0	00	0 1	1	11	Mem[12] Mem[13]	0
	01	0 1	1	01	Mem[4] Mem[5]	1
1	10	0 1	1	01	Mem[6] Mem[7]	1 → 0
	11	0 1	1	10	Mem[10] Mem[11]	0 → 1

Contenido de la caché

**¡Hit!** HR:  $3/8 \rightarrow$  HR final igual a 0.375 o 37.5%

# Memoria caché



## Ejercicios

Ahora, veremos algunos ejercicios.

Estos se basan en preguntas de tareas y pruebas de semestres anteriores, por lo que nos servirán de preparación para las evaluaciones.



## Ejercicios

Un computador tiene una memoria caché de 16KB, con líneas de 32 bytes que almacenan 8 palabras y un tiempo de acceso de 10ns. La memoria caché está conectada a la memoria principal mediante un bus capaz de transferir 8 bytes en 120ns. ¿Cuál es el *hit-rate* que debe tener la memoria caché para tener un tiempo de acceso promedio de 20ns?

## Ejercicios

Un computador de 64 bits tiene una memoria caché de 32KB, con 1024 líneas de 32 palabras. ¿Cuánto espacio de esta caché es usado por información distinta de los datos?

## Ejercicios

¿Cómo es el rendimiento de una memoria caché, si el patrón de accesos a memoria distribuye de manera uniforme sobre todas las posibles direcciones? Ejemplifique el o los posibles casos.

## Ejercicios

El algoritmo de reemplazo MRU (*Most Recently Used*), a diferencia de LRU, descarta primero los elementos que han sido ocupados más recientemente. ¿En qué casos podría ser útil el uso de este esquema?



## Antes de terminar

¿Dudas?

¿Consultas?

¿Inquietudes?

¿Comentarios?





**DCC**  
DEPARTAMENTO DE CIENCIA  
DE LA COMPUTACIÓN

**IIC2343**

# **Arquitectura de Computadores**

**Clase 11 - Jerarquía de Memoria y Memoria Caché**

**Profesor: Germán Leandro Contreras Sagredo**

## Anexo - Resolución de ejercicios

### ¡Importante!

Estos ejercicios pueden tener más de un desarrollo correcto. Las respuestas a continuación no son más que soluciones que **no excluyen** otras alternativas igual de correctas.



## Ejercicios - Respuesta

Un computador tiene una memoria caché de 16KB, con líneas de 32 bytes que almacenan 8 palabras y un tiempo de acceso de 10ns. La memoria caché está conectada a la memoria principal mediante un bus capaz de transferir 8 bytes en 120ns. ¿Cuál es el *hit-rate* que debe tener la memoria caché para tener un tiempo de acceso promedio de 20ns?

Respuesta en la siguiente diapositiva.

## Ejercicios - Respuesta

Nos guiamos por la siguiente fórmula:

$$TP = HR * HT + (1 - HR) * (HT + MP)$$

Donde:

- $TP$  : Tiempo promedio.
- $HR$  : *Hit-rate*.
- $HT$  : *Hit-time*.
- $MP$  : *Miss penalty*.

Ahora, agrupando términos, tenemos que:

$$TP = HT + (1 - HR) * MP$$

Como el tiempo de acceso es 10ns,  $HT = 10ns$ , y como la transferencia del bus es de 120ns por 8 bytes,  $MP = \frac{32}{8} * 120ns = 480ns$ . Finalmente, como queremos que  $TP = 20ns$ :

$$20ns = 10ns + (1 - HR) * 480$$

$$HR = \frac{470}{480} \approx 0,98$$

**Nota:** Recordar que el tiempo de acceso al tener un *miss* incluye el tiempo de acceso en la *caché*, ya que se busca el dato de todas formas, pero no se encuentra.

## Ejercicios - Respuesta

Un computador de 64 bits tiene una memoria caché de 32KB, con 1024 líneas de 32 palabras. ¿Cuánto espacio de esta caché es usado por información distinta de los datos?

Primero, notemos que esto depende del tipo de función de asociación que tengamos. Para poder ejemplificar bien, asumamos que tenemos una función *4-way associative*.

- Al tener  $2^5$  palabras por línea, necesitamos 5 bits para identificar una específica dentro de una línea en especial (*offset*).
- Al ser *4-way associative*, tendremos conjuntos de 4 líneas. Si tenemos  $2^{10}$  líneas, entonces tendremos  $2^8$  conjuntos (es decir, necesitamos 8 bits para identificar el conjunto).
- Finalmente, usamos 13 bits para posicionar (5 bits *offset* + 8 bits conjunto), por lo que el resto lo utilizamos para el *tag*: 51 bits.

Lo que almacenaremos en la tabla serán finalmente los 51 bits del *tag* para cada línea (pues todas las palabras en la línea lo comparten), además de un bit de validez (para ver si el dato fue escrito efectivamente desde la memoria principal). Finalmente, tendremos un espacio total de  $52 * 1024 \text{ bits} = 6.5 \text{ KB}$ .

Veamos qué habría pasado si hubiera sido *fully associative*: solo utilizaríamos los 5 bits de posicionamiento dentro de una línea (ya que todo bloque de memoria puede ser asignado a cualquier línea de la *cache*). Entonces, ahora nuestro *tag* tendrá 59 bits y, por ende, incluyendo el bit de validez ocupará un espacio de  $60 * 1024 \text{ bits} = 7.5 \text{ KB}$ .

Puede realizar el mismo análisis para otros tipos de asociación (*directly mapped*, *2-way associative*, etc.).

**\* Esto, además, no considera la información adicional que se guarda dependiendo de la política de reemplazo (como LRU que requiere bits para determinar la línea de caché accedida hace más tiempo).**

## Ejercicios - Respuesta

¿Cómo es el rendimiento de una memoria caché, si el patrón de accesos a memoria distribuye de manera uniforme sobre todas las posibles direcciones? Ejemplifique el o los posibles casos.

Antes de ejemplificar, notemos que una distribución uniforme implica que se accede a cada posición en memoria en la misma proporción, sin importar el orden en el que se haga. Sabiendo esto, podemos ver dos casos generales. Para ejemplificar mejor, asumiremos una memoria *caché* con líneas de 8 palabras:

- **Caso 1:** Se accede a los datos de memoria en forma secuencial:  $0, 1, 2, 3, \dots, N$ . En este caso, al partir en el acceso 0, tendremos un *miss*, pero guardaremos en la *caché* los datos desde la posición 0 a la 7 en la primera línea, por lo que en los accesos  $1, 2, \dots, 7$  solo tendremos *hits*. Luego, tendremos lo mismo: Tratamos de acceder a la posición 8 con un *miss*, guardamos en la siguiente línea los datos desde la posición 8 a la 15, y en los accesos tendremos  $9, 10, \dots, 15$  solo tendremos *hits*. Esto se repetirá constantemente hasta el último dato, por lo que tendremos un *hit-rate* muy alto.
- **Caso 2:** Se accede a los datos de memoria de la siguiente forma:  $0, 8, 16, \dots, N - 7, 1, 9, 17, \dots, N - 6, \dots, N$ . Al partir en el acceso 0, tendremos un *miss*, y guardamos en la *caché* los datos desde la posición 0 a la 7 en la primera línea. Luego, en el acceso 8, tendremos otro *miss*, almacenando los datos desde la posición 8 a la 15, y así sucesivamente. Si no contáramos con el espacio suficiente en la *caché* para toda la memoria accedida, y siguiéramos una política de reemplazo LRU, las primeras líneas utilizadas serían sobreescritas, y al llegar al acceso 1, tendríamos un *miss* nuevamente por la sobreescritura. Esto, finalmente, conlleva a un *hit-rate* prácticamente nulo.

Bajo estos ejemplos, llegamos a la conclusión de que no es posible catalogar el rendimiento dada la información del enunciado, ya que puede ser muy alto o muy bajo, dependiendo del caso.

## Ejercicios - Respuesta

El algoritmo de reemplazo MRU (*Most Recently Used*), a diferencia de LRU, descarta primero los elementos que han sido ocupados más recientemente. ¿En qué casos podría ser útil el uso de este esquema?

Un caso específico en el que puede ser útil, es en la iteración constante de un arreglo. Supongamos que el arreglo completo no cabe en la *caché*, así que una vez que se nos agote el espacio, reemplazamos la última posición iterada por la siguiente a acceder. Hacemos esto hasta llegar al final, y luego, al reiniciar, tendremos en memoria una porción considerable del mismo, dándonos un buen *hit-rate* (que concluirá al llegar a la posición donde se realizó la primera reescritura).

Notar que este caso convendría siempre y cuando la memoria *caché* pueda contener más de la mitad del arreglo (en otro caso, el *miss rate* sería mayor).