Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 Arquitectura de Computadores

Memoria Caché

1. Motivación

En la mayoría de los computadores modernos, la CPU no se comunica directamente con la memoria principal, sino con una memoria más rápida y de menor tamaño denominada memoria caché. A continuación se explica por qué se utilizan las memorias cachés y como funcionan.

2. Jerarquía de Memoria

La memoria principal de un computador juega un rol fundamental, ya que será en ésta donde se almacenen tanto los datos como las instrucciones de los programas y por tanto la CPU deberá comunicarse continuamente con está para efectuar sus operaciones. Una memoria ideal sería una con una gran capacidad de almacenamiento, que nos permita almacenar muchos datos al mismo tiempo, y con una velocidad de acceso muy rápida, que permita que la CPU pueda obtener la información rápidamente, sin perder mucho tiempo de procesamiento. El problema es que dadas las tecnologías que existen en la actualidad, hay un trade-off entre la velocidad y la capacidad de una memoria: para un costo fijo, una memoria muy rápida puede tener solo una capacidad limitada de almacenamiento, y una memoria de mucha capacidad será de una velocidad reducida. Si ocupáramos solo la memoria más rápida, el costo de tener capacidades altas de almacenamiento sería demasiado; si ocupáramos solo la memoria de gran capacidad la velocidad sería demasiado baja.

La solución a este problema es utilizar una combinación de memorias de distinto tipo, en lo que se denomina la **jerarquía de memoria** de un computador. La jerarquía de memoria de un computador se basa en el uso de diversos niveles de memoria, cada uno de los cuales ocupa una tecnología distinta, aprovechando las ventajas de cada tecnología: para almacenar mucha información se utilizan tecnologías que tienen un costo bajo por byte, como los discos magnéticos; para acceder rápidamente a información se utilizan tecnologías electrónicas de rápido acceso, como SRAM (static RAM); para niveles intermedios se utilizan tecnologías con velocidades de acceso y costos intermedios, como DRAM (dynamic RAM).

La jerarquía de memoria comienza a partir de la CPU, la cual tendrá acceso al primer nivel de memoria, el cual será rápido, pero de poca capacidad. En el siguiente nivel, existirá una memoria de segundo nivel, la cual tendrá una velocidad menor, pero mayor capacidad. La memoria del primer nivel, entonces, será un subconjunto de la memoria del segundo nivel. En el tercer nivel se repite el proceso, está vez con una siguiente memoria cada vez más lenta pero de mayor capacidad. Los niveles pueden continuar mientras sea necesario, pero en la práctica los computadores modernos se basan en jerarquías de 3 niveles.

En la figura 1 se detallan las tecnologías, costos y velocidades de los tres niveles de memoria actualmente usados. El segundo nivel corresponde a la memoria principal («RAM») del computador, la que hasta ahora ha sido el único nivel de memoria que hemos considerado. El tercer nivel corresponde al disco duro, que a pesar de ser un dispositivo de I/O y comunicarse de manera distinta con la CPU, a nivel lógico se considera

como parte de la jerarquía de memoria, dado que es en el disco donde está contenida toda la información (programas y datos) del computador. Por último el primer nivel corresponde a una memoria de alta velocidad conocida como **memoria caché** de la cual hablaremos más adelante en este capítulo.

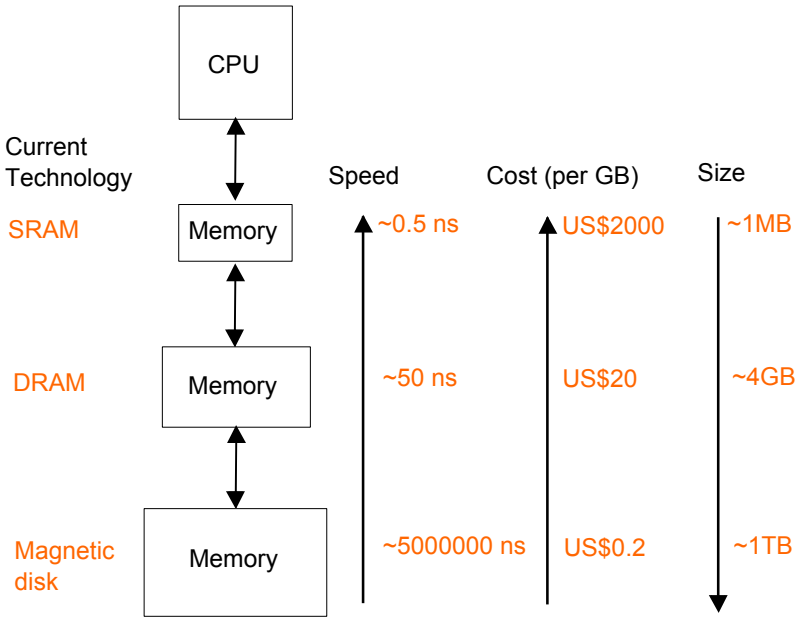

Figura 1: La jerarquía de memoria de los computadores actuales se basa en tres niveles ocupando tres tecnologías distintas.

Si cada vez que se quisiera acceder a un dato hubiese que bajar hasta el último nivel de la jerarquía, no se ganaría mucho, de hecho sería peor que si no hubiera jerarquía: el tiempo total de acceso al dato sería el tiempo acumulado de acceso a los tres niveles. La jerarquía de memoria es útil debido a lo que se conoce como el **principio de localidad**. Este principio se basa en dos tipos de localidad: **localidad temporal** y **localidad espacial**. La localidad temporal plantea que un dato recientemente obtenido de memoria, es muy probable que vuelva a ser usado en el corto plazo. La localidad espacial plantea que si un dato se necesita, es probable que datos ubicados en posiciones cercanas a este se van a necesitar en el corto plazo. De esta forma si la CPU necesita un dato y lo va a buscar al último nivel de la jerarquía, lo que conviene es que además de buscar ese dato, tome también datos contiguos y los vaya copiando en los niveles superiores de la jerarquía. Este conjunto de datos contiguos que se copia entre dos niveles de memoria se conoce como **bloque** o **línea**.

En el siguiente código se muestra una rutina de multiplicación en la arquitectura x86 que ejemplifica el principio de localidad temporal. En este caso, la variable **var2** va a ser accedida varias veces dentro del loop de multiplicación. Si la primera vez que accedemos dejamos una copia en el nivel más alto de la jerarquía, al acceder las siguientes veces será mucho más rápido, ya que la tendremos en el nivel más rápido de memoria.

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV CL, [var1]
0x01	while:	MOV AL,[res]
0x02		ADD AL,**[var2]**
0x03		MOV [res],AL
0x04		SUB CL,1
0x05		CMP CL,0
0x06		JNE while
	DATA:	
0x07	var1	3
0x08	**var2**	**2**
0x09	res	0

En el siguiente código se muestra una rutina que calcula el promedio de un arreglo en la arquitectura x86 que ejemplifica el principio de localidad espacial. En este caso, los distintos valores del arreglo van a ser accedidos todos de manera secuencial. Si al ir a buscar el primer valor del arreglo a un nivel bajo en la jerarquía traemos también el resto de los valores del arreglo (es decir traemos un bloque que contenga los demás valores), cuando se quiera acceder a los siguiente valores, estos estarán en el nivel alto de la jerarquía, y por tanto serán accedidos con mayor velocidad.

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV SI, 0
0x01		MOV AX, 0
0x02		MOV BX, arreglo
0x03		MOV CL, [n]
0x04	while:	CMP SI, CX
0x05		JGE end
0x06		MOV DX, **[BX + SI]**
0x07		ADD AL, DL
0x08		INC SI
0x09		JMP while
0x0A	end:	DIV CL
0x0B		MOV [prom], AL
	DATA:	
0x0C	**arreglo**	**6**
0x0D		**7**
0x0E		**4**
0x0F		**5**
0x10		**3**
0x11	n	5
0x12	prom	0

Cuando el procesador al ir a buscar un dato lo encuentra en el primer nivel de la jerarquía, se denomina un **hit**; si no lo encuentra, se denomina un **miss**. En caso de un miss, se debe acceder al siguiente nivel de la jerarquía a buscar el dato. El **hit rate** es la proporción de accesos a memoria en los cuales se encontró el dato en el primer nivel (o sea hubo un hit); el **miss rate** es la proporción de accesos a memoria en los cuales no se encontró el dato en el primer nivel (o sea hubo un miss), y es equivalente a 1 - hit rate.

La razón para definir la jerarquía de memoria era reducir los tiempos de accesos, por lo que es importante definir métricas para medir esto. El tiempo que le toma la CPU acceder a un dato encontrado en el primer nivel (un hit) se conoce como **hit time**, que incluye además del acceso determinar si hubo hit o miss. El tiempo que se necesita para reemplazar un bloque del nivel más alto por un bloque del siguiente nivel, en el caso de haber un miss, más el tiempo de acceso de la CPU al bloque agregado se conoce como **miss penalty**.

3. Memoria caché

3.1. Fundamentos de la memoria caché

La memoria que ocupa el primer nivel de la jerarquía en un computador se conoce como **memoria caché**. La palabra caché se utiliza no solo en este contexto sino también en cualquier situación en la cual se tiene un lugar de almacenamiento intermedio, el cual se utiliza aprovechando el principio de localidad (por ejemplo, el caché del browser, que almacena archivos y páginas web descargadas recientemente). En el contexto del computador la memoria caché será la memoria que se comunique directamente con la CPU: todo dato que pase por la CPU, pasará también por caché. De esta forma, en los diagramas de computador antes observados basta reemplazar la RAM por caché y el resto del funcionamiento del computador se mantiene igual.

Para entender el funcionamiento de la memoria caché, basta entender como funciona su comunicación con la CPU y con el siguiente nivel en la jerarquía (i.e. la memoria principal). Lo que ocurra más abajo en la jerarquía (e.g. comunicación entre la memoria y el disco) es irrelevante para la caché, dado que esto será transparente. De esta forma la caché sabe que en caso de no encontrar un dato solicitado, debe ir a pedirlo al siguiente nivel y luego almacenarlo.

A nivel de la CPU, lo único que conoce de la jerarquía de memoria es el espacio de direcciones de la memoria principal (segundo nivel de la jerarquía). De esta forma, una instrucción como `MOV AX, [120]` siempre se referirá a la dirección 120 de la memoria principal. Para poder lograr este nivel de transparencia la caché cuenta con un **controlador de caché**, encargado de gestionar la transferencia entre la caché y la memoria principal. Son dos funciones las principales del controlador:

- Mecanismo de acceso a los datos: Ante una solicitud de lectura de un dato de memoria, la caché debe saber ubicar una determinada dirección de la memoria principal (e.g. 120) entre sus datos almacenados. Es decir, para una dirección de memoria, la caché debe saber tanto si tiene ese dato almacenado como dónde lo tiene. En caso de no tener un cierto dato requerido, la caché debe encargarse de ir a buscar el dato a la memoria principal y guardarlo en una determinada posición, que después le permita ubicarlo si se solicita de nuevo.
- Políticas de escritura: Cuando la CPU escribe un dato en memoria (e.g. `MOV [120], AX`), la caché debe en algún momento actualizar ese valor en la memoria principal, para asegurar consistencia, y por tanto debe tener definida una política de escritura que indica cuando se realizará dicha actualización.

3.1.1. Mecanismo de acceso a los datos

Para poder almacenar bloques desde la memoria principal, la caché debe definir una **función de correspondencia** entre las direcciones de la memoria principal y su propia memoria. La función de correspondencia más simple se conoce como **directly mapped**. Con este mecanismo, cada **bloque** de la memoria principal tiene asociado un solo **bloque** en la memoria caché. La caché identificará cada bloque interno con un **índice** (equivalente a una dirección, pero para bloques). La fórmula de asociación entre un bloque de memoria y un índice de caché depende de los siguientes factores:

- Tamaño de cada bloque de caché. Por simplicidad siempre será una potencia de 2.
- Cantidad de bloques en caché. Por simplicidad siempre será una potencia de 2.
- Dirección del bloque en memoria. La dirección del bloque de memoria será la dirección de la primera palabra del bloque.

Para observar como se realiza la asociación entre memoria y caché, revisemos el siguiente ejemplo: se tiene una memoria principal de 32 bytes (32 palabras), una caché de 8 bytes, con 4 bloques cada uno de 2 bytes (2 palabras). El 5to bloque de memoria está asociado a la dirección 8 = 01000. Para obtener su asociación en caché hay que descomponer la dirección de la siguiente forma:

- Como la caché tiene bloques de 2 palabras = $2^{\mathbf{1}}$, se requiere **1** bit para determinar la posición de la palabra dentro del bloque. En este ejemplo entonces, el bit menos significativo de la dirección se usará para indicar la posición dentro del bloque, o sea 0100**0**, por lo que en este caso, la palabra asociada a la dirección 8 = 01000 estará almacenada en la palabra 0 del bloque.
- Como la caché tiene 4 bloques = $2^{\mathbf{2}}$, se requieren **2** bits para determinar el índice del bloque dentro de la caché. Se usarán los siguientes dos bits de la dirección para determinar el índice, o sea 01**00**0, por lo que en este caso, el bloque asociado a la dirección 8 = 01000 se almacenará en el bloque 00 de caché.

En el diagrama de la figura 2 se observa la memoria y caché del ejemplo y distintas asociaciones de bloques de memoria con bloques de caché. Se puede observar que un mismo bloque de caché será usado para varios bloques de memoria. El problema de esto, es que para que la CPU sepa que palabra de caché corresponde a que palabra de memoria, se requiere almacenar información adicional. Esta información adicional que se almacena se conoce como **tag**. El tag corresponderá al resto de los bits de la dirección del bloque, que no fueron ocupados para determinar el índice o palabra en caché. En el caso del ejemplo anterior, el tag para el bloque 8 = 01000 sería **01**000. Este valor se almacena en un lugar especial de la caché reservado para los tags.

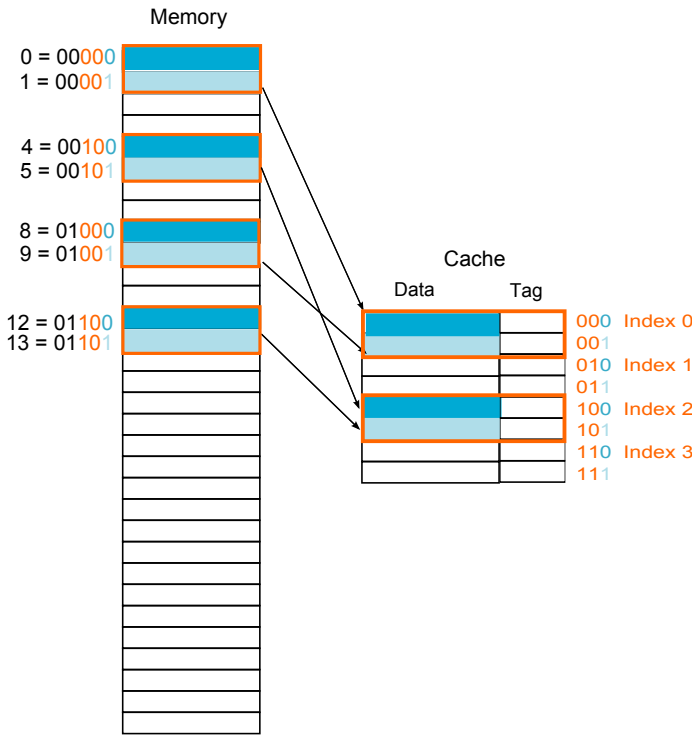

Figura 2: En una caché directly mapped, cada bloque de memoria tendrá una ubicación fija en la caché. Cada posición de caché, por su parte, podrá tener uno de varios bloques de memoria.

Un elemento adicional que se debe almacenar para cada palabra es un bit de validez o **valid bit**. Este bit indica si los valores almacenados en caché son válidos, lo que es importante al comenzar a llenar la caché, ya que cuando la caché está vacía, las palabras que tienen almacenadas no son válidas.

A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. La palabra asociada a la dirección 01100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 13 = 01101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	**Mem[12]**
	1	1	01	**Mem[13]**
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 11. La palabra asociada a la dirección 01110 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 15 = 01111 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	1	01	**Mem[14]**
	1	1	01	**Mem[15]**

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. Como este bloque ya está ocupado, hay que eliminar lo que se tenía guardado y reemplazarlo por el nuevo bloque traído de memoria. La palabra asociada a la dirección 00100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 5 = 00101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 00.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	00	**Mem[4]**
	1	1	00	**Mem[5]**
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

- El siguiente acceso es a la dirección 12 = 01100, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 10. Como este bloque ya está ocupado, hay que eliminar lo que se tenía guardado y reemplazarlo por el nuevo bloque traído de memoria. La palabra asociada a la dirección 01100 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 13 = 01101 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 01.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	**Mem[12]**
	1	1	01	**Mem[13]**
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

- El último acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria en el bloque de caché con índice 00. La palabra asociada a la dirección 00000 se guardará en la ubicación 0. Adicionalmente se guardará también el dato de la dirección 1 = 00001 que es parte del bloque de memoria, y se almacenará en el mismo bloque de caché, pero en la ubicación 1. El tag para ambas palabras es 00.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	00	**Mem[0]**
	1	1	00	**Mem[1]**
01	0	0		
	1	0		
10	0	1	01	Mem[12]
	1	1	01	Mem[13]
11	0	1	01	Mem[14]
	1	1	01	Mem[15]

3.2. Políticas de escritura

En el caso de que la CPU quiera escribir un dato en memoria además de preocuparse del acceso, debe encargarse de actualizar la memoria principal para que está mantenga consistencia en la información. Existen dos políticas usadas para realizar esta actualización

- Write-through: en esta política cada escritura en caché, se actualiza inmediatamente en memoria. La ventaja de esta política es que la memoria principal nunca tendrá datos inconsistentes. La desventaja es que cada escritura involucrará una espera considerable para la CPU ya que deberá esperar no tanto la escritura en caché como la escritura en memoria principal.

- Write-back: en esta política las escrituras se realizan en principio sólo en caché. El valor se actualizará en memoria sólo cuando el bloque que estaba en caché vaya a ser reemplazado. La ventaja de este mecanismo es que es mucho más eficiente, ya que las escrituras se realizan solo en los reemplazos y no siempre. La desventaja es que la memoria principal puede quedar con datos inconsistentes, lo que puede afectar si un dispositivo de I/O intenta acceder a esta mediante DMA. Para que esta política no genere errores de consistencia se deben agregar mecanismos que realicen la actualización ante un posible caso de inconsistencia, como en un acceso por DMA.

3.3. Mejoras al rendimiento de la memoria caché

3.3.1. Funciones de correspondencia

Una caché con función de correspondencia **directly mapped** presenta la desventaja de que al tener un mapeo fijo entre bloques de memoria y caché no se está aprovechando al máximo la disponibilidad de la caché. El problema ocurre por la posible contención que pueda ocurrir entre dos bloques de memoria mapeados a un mismo bloque de caché, los cuales aunque haya espacio disponible en otro bloque, estarán compitiendo por el espacio del mismo bloque. Este factor afecta directamente al hit rate de la caché, y por tanto es necesario definir funciones de correspondencia más eficientes.

Existen dos funciones de correspondencia además de directly mapped, que intentan solucionar el problema de la contención: **fully associative** y n-way associative:

Fully associative

En una memoria con función de correspondencia fully associative, cada bloque de memoria puede asociarse a cualquier bloque de la caché. De esta manera se aprovecha al máximo la disponibilidad de espacio en la caché y se evita el problema de la contención. El problema de esta función de correspondencia, es que ante un requerimiento de memoria de la CPU, ahora se vuelve más difícil encontrar la palabra en caché. En el caso de directly mapped, ocupando los bits de la dirección era posible llegar a la única posible ubicación de la palabra en caché. Una vez en esa posición se podía revisar el tag y con eso determinar si estaba o no la palabra. En el caso de fully associative, no es posible ir directamente a la posición dado que puede ser cualquiera. Para saber si una palabra está en caché es necesario buscar en todas las posiciones, y además es necesario guardar un tag más grande, que contenga toda la dirección menos los bits de ubicación de la palabra en el bloque.

Para que la detección de la palabra no sea tan lenta, se utilizan circuitos comparadores que funcionan en paralelo, lo que complejiza el hardware de la caché, haciéndola más cara. Además, a pesar de hacer la comparación en paralelo, el hecho de realizarla involucra un gasto de tiempo, por lo que está función, aun cuando aumenta el hit rate, aumenta también el hit time. De todas maneras la ganancia en velocidad por el aumento del hit rate supera lo que se pierde por aumentar el hit time. La principal desventaja de esta función es la necesidad del hardware complejo de comparación.

A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras con función de correspondencia fully associative para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, y como la caché es fully associative, el bloque asociado a esa palabra (Mem[12] y Mem[13]) se guardará en el primer bloque disponible de la caché (Índice 00). El tag para ambas palabras es 0110, es decir toda la dirección menos el bit de ubicación de la palabra. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	**Mem[12]**
	1	1	0110	**Mem[13]**
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como sí se encontró el valor en memoria, este acceso es un hit.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria (Mem[14] y Mem[15]) en el siguiente bloque de caché disponible (índice 01). El tag para ambas palabras es 0111. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	**Mem[14]**
	1	1	0111	**Mem[15]**
10	0	0		
	1	0		
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria (Mem[4] y Mem[5]) en el siguiente bloque de caché disponible. El tag para ambas palabras es 0010. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	**Mem[4]**
	1	1	0010	**Mem[5]**
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 12 = 01100. Como el bloque que se copió ya tenía ese valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como sí se encontró el valor en memoria, este acceso es un hit.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	Mem[4]
	1	1	0010	Mem[5]
11	0	0		
	1	0		

- El siguiente acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria (Mem[0] y Mem[1]) en el siguiente bloque de caché disponible. El tag para ambas palabras es 0000. Como no se encontró el valor en memoria, este acceso es un miss.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	1	0110	Mem[12]
	1	1	0110	Mem[13]
01	0	1	0111	Mem[14]
	1	1	0111	Mem[15]
10	0	1	0010	Mem[4]
	1	1	0010	Mem[5]
11	0	1	0000	**Mem[0]**
	1	1	0000	**Mem[1]**

El hit rate de esta secuencia de accesos fue de $\frac{2}{6}$ que es mejor que en el caso directly mapped donde había sido $\frac{1}{6}$.

N-way associative

Con directly mapped, la caché es simple, pero a costa de un bajo hit rate. Con fully associative, la caché es compleja, pero logra un alto hit rate. La función de correspondencia n-way associative busca ser un punto medio entre estos dos esquemas, tratando de simplificar el hardware de la caché, pero sin perder las ventajas de la fully associative respecto al hit rate. Para esto, la caché se divide lógicamente en **conjuntos** o sets, siendo cada uno un grupo de bloques. Cada bloque de memoria tendrá un mapeo directo a un solo conjunto, pero dentro del conjunto podrá ubicarse en cualquier bloque disponible (figura 3). De esta forma, al tener el mapeo directo al conjunto, se reducen las comparaciones al momento de buscar si está un dato o no en caché, y al tener libertad de elegir en que bloque del conjunto almacenar, se mantienen niveles de hit rate altos.

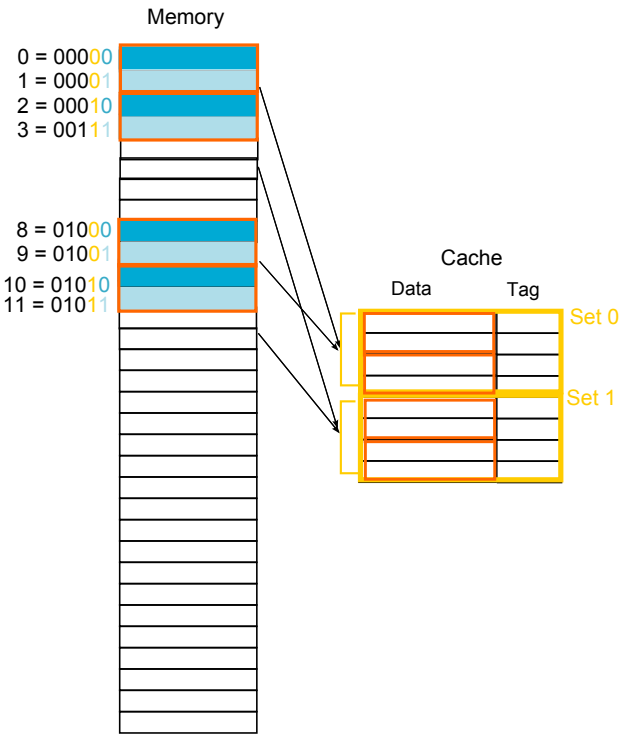

Figura 3: En una caché n-way associative, cada bloque de memoria estará asociado a un conjunto de bloques de la caché, y dentro del conjunto se podrá ubicar en cualquier lugar. En este caso, la caché es 2-way associative, es decir, tiene conjuntos de 2 bloques cada uno.

Dependiendo de cuantos bloques tenga cada conjunto, se tendrán distintos tipos de funciones n-way associative. Si se tienen 4 bloques por conjunto, se denominará 4-way associative; 2 bloques por conjunto, 2-way associative y así. A continuación se muestra un ejemplo del funcionamiento de una caché de 4 bloques de 2 palabras con función de correspondencia 2-way associative para un programa que accede a la siguiente secuencia de direcciones de memoria: 12, 13, 14, 4, 12, 0.

- En un comienzo la caché comienza vacía, y por tanto todos los bits de validez están en 0, indicando que los datos actuales no son válidos.

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

- El primer acceso es a la dirección 12 = 01100, y como la caché es 2-way associative, el bloque asociado a esa palabra (Mem[12] y Mem[13]) se guardará en el primer bloque disponible del conjunto 0. El tag para ambas palabras es 011. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	**Mem[12]**
		1	1	011	**Mem[13]**
	01	0	0		
		1	0		
1	10	0	0		
		1	0		
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 13 = 01100. Como en el paso anterior el bloque que se copió contenía ya este valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como sí se encontró el valor en memoria, este acceso es un hit.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0		
		1	0		
1	10	0	0		
		1	0		
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 14 = 01110, por lo tanto se almacenará el bloque de memoria (Mem[14] y Mem[15]) en el siguiente bloque de caché disponible del conjunto 1. El tag para ambas palabras es 011. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0		
		1	0		
1	10	0	1	011	**Mem[14]**
		1	1	011	**Mem[15]**
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 4 = 00100, por lo tanto se almacenará el bloque de memoria (Mem[4] y Mem[5]) en el siguiente bloque de caché disponible del conjunto 0. El tag para ambas palabras es 001. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0	001	**Mem[4]**
		1	0	001	**Mem[5]**
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 12 = 01100. Como el bloque que se copió ya tenía ese valor, no hay que ir a buscarlo a memoria y la CPU puede obtener el dato directamente de caché. Como sí se encontró el valor en memoria, este acceso es un hit.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	011	Mem[12]
		1	1	011	Mem[13]
	01	0	0	001	Mem[4]
		1	0	001	Mem[5]
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

- El siguiente acceso es a la dirección 0 = 00000, por lo tanto se almacenará el bloque de memoria (Mem[0] y Mem[1]) en el siguiente bloque de caché disponible del conjunto 0. Como el conjunto está lleno, se debe reemplazar un bloque, en este caso, se reemplazará el primero que ingresó. El tag para ambas palabras es 000. Como no se encontró el valor en memoria, este acceso es un miss.

Conjunto	Índice bloque	Ubicación palabra	Bit validez	Tag	Dato
0	00	0	1	000	**Mem[0]**
		1	1	000	**Mem[1]**
	01	0	0	001	Mem[4]
		1	0	001	Mem[5]
1	10	0	1	011	Mem[14]
		1	1	011	Mem[15]
	11	0	0		
		1	0		

El hit rate de esta secuencia de accesos fue de $\frac{2}{6}$ que es equivalente al ejemplo con fully associative.

3.3.2. Algoritmos de reemplazo

Para las funciones de correspondencia fully associative y n-way associative, hay que definir una política de que bloque reemplazar en caso de no haber más espacio, lo que se conoce como **algoritmos de reemplazo**. Existen diversos algoritmos de reemplazo, que en distintas circunstancias pueden representar una mejor alternativa:

First-in First-out (FIFO)

El algoritmo de reemplazo más simple se conoce como first-in first-out o FIFO, en el cual, el primer bloque que entró va a ser el primero en salir. Para implementar este algoritmo, se debe agregar información que indique el orden con que entraron los bloques para ir sacándolos acordemente.

A continuación se muestra un ejemplo del funcionamiento del algoritmo de reemplazo en una caché fully associative, llenada inicialmente con la secuencia: 12, 13, 14, 4, 12, 0 (el desglose de la secuencia está en el ejemplo de la sección anterior). Supongamos ahora que luego de esa secuencia viene el acceso a la secuencia 14, 7, 17:

- Lo primero es determinar el orden actual de los bloques de caché para saber a cuál le corresponde salir, lo que se observa a continuación:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0110	Mem[12]	1
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	2
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	3
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	4
	1	1	0000	Mem[1]	

- El primer acceso es al bloque 14-15, el cual ya está en caché por lo tanto no hay reemplazo. El acceso al bloque no afecta al orden FIFO:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0110	Mem[12]	1
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	2
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	3
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	4
	1	1	0000	Mem[1]	

- Como el primer bloque en entrar fue el de las direcciones 12 y 13, ese bloque se reemplazará por el bloque de las direcciones 6 y 7. Luego del reemplazo, se actualiza el orden:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0011	**Mem[6]**	4
	1	1	0011	**Mem[7]**	
01	0	1	0111	Mem[14]	1
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	2
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	3
	1	1	0000	Mem[1]	

- Como el primer bloque en entrar de los que están fue el de las direcciones 14 y 15, ese bloque se reemplazará por el bloque de las direcciones 16 y 17. Luego del reemplazo, se actualiza el orden:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden
00	0	1	0011	Mem[6]	3
	1	1	0011	Mem[7]	
01	0	1	1000	**Mem[16]**	4
	1	1	1000	**Mem[17]**	
10	0	1	0010	Mem[4]	1
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	2
	1	1	0000	Mem[1]	

Least-frequently used (LFU)

El algoritmo least-frequently used (LFU) reemplazará el bloque que ha sido accedido menos frecuentemente. La idea es que si un bloque ha sido accedido varias veces, es probable que vuelva a ser accedido (principio de localidad temporal). Para implementar este algoritmo, se debe almacenar un contador de accesos a cada bloque, de manera de saber que bloque ha sido accedido menos. En caso de haber dos bloques empatados en menor cantidad de accesos, se ocupará otro algoritmo para desempatar, generalmente FIFO.

A continuación se muestra un ejemplo del funcionamiento del algoritmo de reemplazo en una caché fully associative, llenada inicialmente con la secuencia: 12, 13, 14, 4, 12, 0 (el desglose de la secuencia está en el ejemplo de la sección anterior). Supongamos ahora que luego de esa secuencia viene el acceso a la secuencia 14, 7, 17:

- Lo primero es determinar el orden actual de los bloques y cuantos accesos han tenido, lo que se observa a continuación:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden	Nº Accesos
00	0	1	0110	Mem[12]	1	3
	1	1	0110	Mem[13]		
01	0	1	0111	Mem[14]	2	1
	1	1	0111	Mem[15]		
10	0	1	0010	Mem[4]	3	1
	1	1	0010	Mem[5]		
11	0	1	0000	Mem[0]	4	1
	1	1	0000	Mem[1]		

- El primer acceso es al bloque 14-15, el cual ya está en caché por lo tanto no hay reemplazo. El acceso al bloque modificará el contador de accesos:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden	Nº Accesos
00	0	1	0110	Mem[12]	1	3
	1	1	0110	Mem[13]		
01	0	1	0111	Mem[14]	2	2
	1	1	0111	Mem[15]		
10	0	1	0010	Mem[4]	3	1
	1	1	0010	Mem[5]		
11	0	1	0000	Mem[0]	4	1
	1	1	0000	Mem[1]		

- Como el bloque de las direcciones 12 y 13 ha sido accedido 3 veces, a pesar de ser el primero en entrar, no será reemplazado. El siguiente bloque en el orden FIFO (Mem[14] y Mem[15]) tiene 2 accesos, y por tanto tampoco será reemplazo. El bloque que se reemplazará será el 4-5 que sigue en la lista FIFO y tiene un solo acceso. Ese bloque se reemplazará por el bloque de las direcciones 6 y 7. Luego del reemplazo, se actualizan el orden y los accesos:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden	Nº Accesos
00	0	1	0110	Mem[12]	1	3
	1	1	0110	Mem[13]		
01	0	1	0111	Mem[14]	2	2
	1	1	0111	Mem[15]		
10	0	1	0011	**Mem[6]**	4	1
	1	1	0011	**Mem[7]**		
11	0	1	0000	Mem[0]	3	1
	1	1	0000	Mem[1]		

- Nuevamente, como el bloque de las direcciones 12 y 13 ha sido accedido 3 veces, y el 14-15 2 veces, a pesar de ser los primeros en entrar, no serán reemplazados tampoco en este caso. Se utilizará el siguiente bloque en el orden FIFO (Mem[0] y Mem[1]) y ese bloque se reemplazará por el bloque de las direcciones 16 y 17. Luego del reemplazo, se actualizan el orden y los accesos:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Orden	Nº Accesos
00	0	1	0110	Mem[12]	1	3
	1	1	0110	Mem[13]		
01	0	1	0111	Mem[14]	2	2
	1	1	0111	Mem[15]		
10	0	1	0011	Mem[6]	3	1
	1	1	0011	Mem[7]		
11	0	1	1000	**Mem[16]**	4	1
	1	1	1000	**Mem[17]**		

Least-recently Used (LRU)

Un tercer algoritmo de reemplazo se denomina least-recently used (LRU) el cual define el reemplazo usando el bloque que se usó hace más tiempo. Este algoritmo es similar a LFU en el sentido de que interesa si un bloque es accedido varias veces, pero la diferencia es que solo importa cuando fue el último acceso, y no la frecuencia acumulada. Para implementarlo en hardware, es necesario almacenar un «timestamp» de cuando se accedió al bloque, lo que lo hace complejo. A diferencia de en LFU, no habrá posibilidad de empate ya que los tiempos siempre serán distintos.

A continuación se muestra un ejemplo del funcionamiento del algoritmo de reemplazo en una caché fully associative, llenada inicialmente con la secuencia: 12, 13, 14, 4, 12, 0 (el desglose de la secuencia está en el ejemplo de la sección anterior). Supongamos ahora que luego de esa secuencia viene el acceso a la secuencia 14, 7, 17:

- Lo primero es determinar el orden actual de los bloques y el tiempo en que se accedió por última vez. Para simplificar este ejemplo, se utilizará como unidad de tiempo lo que haya transcurrido entre dos accesos de la secuencia:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Tiempo
00	0	1	0110	Mem[12]	2
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	3
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	4
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	1
	1	1	0000	Mem[1]	

- El primer acceso es un hit al bloque 14-15, y por tanto lo único que ocurre es que se actualiza el tiempo de acceso:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Tiempo
00	0	1	0110	Mem[12]	3
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	1
	1	1	0111	Mem[15]	
10	0	1	0010	Mem[4]	5
	1	1	0010	Mem[5]	
11	0	1	0000	Mem[0]	2
	1	1	0000	Mem[1]	

- Para agregar el bloque 6 y 7 se busca el bloque de caché accedido hace más tiempo, en este caso corresponde al bloque 4-5, el cual será entonces reemplazado. Luego del reemplazo, se actualizan el orden y el tiempo:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Tiempo
00	0	1	0110	Mem[12]	4
	1	1	0110	Mem[13]	
01	0	1	0111	Mem[14]	2
	1	1	0111	Mem[15]	
10	0	1	0011	**Mem[6]**	1
	1	1	0011	**Mem[7]**	
11	0	1	0000	Mem[0]	3
	1	1	0000	Mem[1]	

- Para agregar el bloque 16 y 17 se busca el bloque de caché accedido hace más tiempo, en este caso corresponde al bloque 12-13, el cual será entonces reemplazado. Luego del reemplazo, se actualizan el orden y el tiempo:

Índice bloque	Ubicación palabra	Bit validez	Tag	Dato	Tiempo
00	0	1	1000	**Mem[16]**	1
	1	1	1000	**Mem[17]**	
01	0	1	0111	Mem[14]	3
	1	1	0111	Mem[15]	
10	0	1	0011	Mem[6]	2
	1	1	0011	Mem[7]	
11	0	1	0000	Mem[0]	4
	1	1	0000	Mem[1]	

Random

Por último, otra alternativa es no ocupar ningún algoritmo «inteligente» y simplemente elegir al azar que bloque retirar. Lo interesante de este mecanismo es que se han hecho análisis que muestran que en la práctica ocupar un algoritmo random, entrega resultados no muy inferiores a los otros algoritmos.

3.3.3. Tipos de Cachés

En un computador con arquitectura Von Neumann, la memoria principal tendrá almacenado tanto las instrucciones como los datos de los programas. Debido a esto, el tipo de caché más simple, denominado **caché unified** también almacenará ambos (instrucciones y datos). El problema de esto es que a nivel del programa, las instrucciones y datos se guardan en porciones separadas de memoria, por lo que si los tamaños de bloque son pequeños, se están perdiendo las ventajas entregadas por el principio de localidad espacial. Para solucionar este problema, existe un tipo especial de caché denominado **caché split** el cual puede ser pensado como dos caché independientes uno para almacenar instrucciones y otro para datos.

El siguiente código corresponde a una rutina de multiplicación en el computador básico Von Neumann, que tiene palabras de 16 bits. Se asume que el program counter comienza desde la dirección 5. A continuación se muestra el resultado de ir almacenando los accesos a memoria en dos caché: una de tipo unified y otra split. La caché unified es de 8 palabras, bloques de tamaño 2 palabras y directly mapped; la caché split se divide en dos cachés de 4 palabras, bloques de tamaño 2 palabras y directly mapped.

Dirección	Label	Instrucción/Dato	
	DATA:		
0	i	0	
1	var1	2	
2	var2	2	
3	res	0	
	CODE:		
4	start:	MOV	A,(res)
5		ADD	A,(var2)
6		MOV	(res),A
7		MOV	A,(i)
8		ADD	A,1
9		MOV	(i),A
10		MOV	B,(var1)
11		CMP	A,B
12		JLT	start

1. Caché unified: la secuencia de accesos a memoria en el ejemplo de código es la siguiente:

   - Primer ciclo del loop: 4,3,5,2,6,3,7,0,8,9,0,10,1,11,12
   - Segundo ciclo del loop: 4,3,5,2,6,3,7,0,8,9,0,10,1,11,12

   A partir de esta secuencia, podemos observar la secuencia de estados para la caché unified:

Dir	Bloque 0	Bloque 1	Bloque 2	Bloque 3	
4			Mem[4]-Mem[5]		M
3		Mem[2]-Mem[3]	Mem[4]-Mem[5]		M
5		Mem[2]-Mem[3]	Mem[4]-Mem[5]		H
2		Mem[2]-Mem[3]	Mem[4]-Mem[5]		H
6		Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
3		Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
7		Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
0	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
8	Mem[8]-Mem[9]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
9	Mem[8]-Mem[9]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
0	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
10	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
1	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
11	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
12	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[12]-Mem[13]	Mem[6]-Mem[7]	M
4	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
3	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
5	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
2	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
6	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
3	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
7	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
5	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
0	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
8	Mem[8]-Mem[9]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
9	Mem[8]-Mem[9]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
0	Mem[0]-Mem[1]	Mem[2]-Mem[3]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
10	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	M
1	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
11	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[4]-Mem[5]	Mem[6]-Mem[7]	H
12	Mem[0]-Mem[1]	Mem[10]-Mem[11]	Mem[12]-Mem[13]	Mem[6]-Mem[7]	M

El hit rate es de $\frac{17}{30}$

2. Caché split: la secuencia de accesos a los datos es la siguiente:

   - Primer ciclo del loop: 3,2,3,0,0,1
   - Segundo ciclo del loop: 3,2,3,0,0,1

   La secuencia de accesos a las instrucciones es la siguiente:

   - Primer ciclo del loop: 4,5,6,7,8,9,10,11,12
   - Segundo ciclo del loop: 4,5,6,7,8,9,10,11,12

   A partir de esta secuencia, podemos observar la secuencia de estados para la caché split:

	Instrucciones	Instrucciones	Datos	Datos	
Dir	Bloque 0	Bloque 1	Bloque 0	Bloque 1	
4	Mem[4]-Mem[5]				M
3	Mem[4]-Mem[5]			Mem[2]-Mem[3]	M
5	Mem[4]-Mem[5]			Mem[2]-Mem[3]	H
2	Mem[4]-Mem[5]			Mem[2]-Mem[3]	H
6	Mem[4]-Mem[5]	Mem[6]-Mem[7]		Mem[2]-Mem[3]	M
3	Mem[4]-Mem[5]	Mem[6]-Mem[7]		Mem[2]-Mem[3]	H
7	Mem[4]-Mem[5]	Mem[6]-Mem[7]		Mem[2]-Mem[3]	H
0	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
8	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
9	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
0	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
10	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
1	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
11	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
12	Mem[12]-Mem[13]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
4	Mem[4]-Mem[5]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
3	Mem[4]-Mem[5]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
5	Mem[4]-Mem[5]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
2	Mem[4]-Mem[5]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
6	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
3	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
7	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
5	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
0	Mem[4]-Mem[5]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
8	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
9	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
0	Mem[8]-Mem[9]	Mem[6]-Mem[7]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
10	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M
1	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
11	Mem[8]-Mem[9]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	H
12	Mem[12]-Mem[13]	Mem[10]-Mem[11]	Mem[0]-Mem[1]	Mem[2]-Mem[3]	M

El hit rate es de $\frac{19}{30}$

3.3.4. Cachés multinivel

Otra manera de mejorar el desempeño de caché es crear una jerarquía de caché, es decir varios niveles cachés de tamaños sucesivamente mayores. En la práctica los computadores personales actuales utilizan 3 niveles de caché, llamados L1, L2 y L3. El nivel L1 corresponde a una caché de tipo split, los siguientes unified. Los niveles L1 y L2 están en el mismo chip que la CPU, la caché L3, que es la más grande, es externa a la CPU.

4. Anexo

4.1. Rendimiento de Caché

El rendimiento de una memoria caché ante la ejecución de un determinado programa depende tanto del miss rate que se tenga, como del miss penalty. Para calcular el rendimiento, es necesario considerar cuánto tiempo de ejecución se está perdiendo por las fallas de caché. En un sistema ideal, en el que no haya misses de caché, el tiempo que demore en ejecutar un programa se denomina **tiempo ideal de ejecución**, el cual se calcula con la siguiente ecuación:

$$TiempoIdeal = NumeroInst \times CiclosPorInst \times TiempoPorCiclo \tag{1}$$

De esta forma, si un programa tiene 100 instrucciones, cada instrucción utiliza 1 ciclo, y cada ciclo se ejecuta en 5 nanosegundos, el tiempo ideal de ejecución del programa es: $100 \times 1 \times 5 = 500$ nanosegundos.

En un sistema no ideal, en el cual existen misses, es necesario agregar el tiempo perdido en ir a buscar palabras a la memoria principal, para poder determinar el **tiempo real de ejecución**. La ecuación para calcularlo es la siguiente:

$$TiempoReal = (NumeroInst \times CiclosPorInst + CiclosMemoryStall) \times TiempoPorCiclo \tag{2}$$

Donde el número de ciclos de memory stall se refiere a los ciclos los cuales perdió la CPU esperando a que el controlador de caché obtuviera la palabra solicitada desde memoria principal. Esta cantidad se puede calcular mediante la siguiente ecuación:

$$NumeroCiclosMemoryStall = NumeroAccesosAMemoria \times MissRate \times MissPenalty \tag{3}$$

De esta forma, para el mismo programa anterior, si se consideran 200 accesos a memoria, un miss rate de 10 %, y un miss penalty de 5 ciclos, el número de ciclos perdidos en memory stall sería: $200 \times 0{,}1 \times 5 = 100$ ciclos.

El tiempo real para este ejemplo sería: $(100 \times 1 + 100) \times 5 = 1000$ nanosegundos.

Con esta información es posible calcular el rendimiento de la caché para un determinado programa, calculando la razón entre ambos tiempos:

$$Rendimiento = \frac{TiempoIdeal}{TiempoReal} \tag{4}$$

En el ejemplo el rendimiento sería: $\frac{500}{1000} = 0,5$, es decir se tiene un rendimiento del 50 %.

4.2. Reducción de misses

Para mejorar el rendimiento de un programa ejecutándose en un sistema con caché, hay dos alternativas: modificar la caché a nivel de hardware o a nivel de software. Las modificaciones a nivel de hardware pueden incidir tanto en una disminución del miss rate como del miss penalty. Las modificaciones en software, en cambio solo puede modificar el miss rate.

Existen distintos mecanismos para disminuir el miss rate realizando modificaciones de software, y en general se relacionan con modificar la forma en que se almacenan los datos, o la forma en que se accede a estos. Hay cuatro esquemas principales que pueden ser aplicados en diversos programas:

- Mezcla de Arreglos:

  Este esquema se basa en modificar la ubicación de los datos en memoria, para que, dado la forma en que van a ser accedidos en el programa, se aproveche la localidad espacial. En el siguiente ejemplo, se

tienen tres arreglos independientes que son accedidos de manera secuencial en un ciclo que calcula el promedio entre dos valores:

```
public static void independentArrays(int n)
{
  double[] i1 = new double[n];
  double[] i2 = new double[n];
  double[] promedio = new double[n];

  Random rand = new Random();

  for(int i=0; i<n; i++)
  {
    i1[i] = rand.NextDouble()*6+ 1;
    i2[i] = rand.NextDouble()*6+ 1;
  }

  long time = System.DateTime.Now.Ticks;
  for(int i=0; i<n; i++)
  {
    promedio[i] = (i1[i] + i2[i])/2;
  }

  double delta = (System.DateTime.Now.Ticks - time);
  Console.WriteLine(delta);

}
```

El problema de este caso está en que se debe ir saltando entre tres zonas separadas de caché para recuperar los tres valores del arreglo. Para solucionar esto, se puede crear una agrupación lógica de estos tres valores, usando un `struct` y crear un arreglo de estos elementos, de manera que ahora al acceder a los valores, estos estén almacenados de manera contigua y se aproveche mejor la localidad espacial:

```
struct Notas
{
    public double i1;
    public double i2;
    public double promedio;
}

public static void mergedArrays(int n)
{
  Notas[] notas = new Notas[n];
  Random rand = new Random();

  for(int i=0; i<n; i++)
  {
    notas[i].i1 = rand.NextDouble()*6+ 1;
          notas[i].i2 = rand.NextDouble()*6+ 1;
  }

      long time = System.DateTime.Now.Ticks;
      for (int i = 0; i < n; i++)
  {
    notas[i].promedio = (notas[i].i1 + notas[i].i2)/2;
  }
      double delta = (System.DateTime.Now.Ticks - time);
      Console.WriteLine(delta);
}
```

- Intercambio de Loops:

Este esquema modifica el orden en que se acceden a los datos, para aprovechar mejor la localidad espacial. En el ejemplo, se tiene una matriz almacenada por filas en memoria, la cual es recorrida primero por columnas y luego por filas. Al realizar esto, nuevamente ocurre el problema de ir saltando entre zonas separadas de la memoria (en este caso, se salta entre las filas).

```
 public static void matrizNoExchange(int m, int n)
{
  int[,] matriz = new int[m,n];

  long time = System.DateTime.Now.Ticks;
  for (int j = 0; j < n; j++)
  {
    for(int i=0; i<m; i++)
    {
      matriz[i,j] = 1;
    }
  }
  double delta = (System.DateTime.Now.Ticks - time);
      Console.WriteLine(delta);
}
```

Una solución sencilla es intercambiar los loops, e ir recorriendo primero por filas, y luego por columnas, lo que resulta en un avance secuencial a través de la memoria:

```
 public static void matrizLoopExchange(int m, int n)
{
    int[,] matriz = new int[m, n];

    long time = System.DateTime.Now.Ticks;
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            matriz[i, j] = 1;
        }
    }
    double delta = (System.DateTime.Now.Ticks - time);
    Console.WriteLine(delta);
}
```

- Fusión de Loops:

  Este tercer esquema, se basa en aprovechar mejor la localidad temporal, agrupando en el tiempo accesos repetidos a datos. Por ejemplo en el siguiente acceso, se está recorriendo de manera separada, primero para generar los valores de dos arreglos, y luego para realizar una operación entre ellos:

```
public static void noLoopFusion(int n)
{
    int[] array1 = new int[n];
    int[] array2 = new int[n];
    int[] array3 = new int[n];

    Random rand = new Random();
    long time = System.DateTime.Now.Ticks;
    for (int i = 0; i < n; i++)
    {
        array2[i] = rand.Next(n);
        array3[i] = rand.Next(n);
    }

    for (int i = 0; i < n; i++)
    {
        array1[i] = array2[i] * array3[i];
    }
    double delta = (System.DateTime.Now.Ticks - time);
    Console.WriteLine(delta);
}
```

  En este caso, si se juntan las operaciones en un solo recorrido de los arreglos, se está logrando que los valores de los dos arreglos que recién fueron accedidos, se utilicen de inmediato, y así se aproveche la localidad temporal.

```
public static void loopFusion(int n)
    {
        int[] array1 = new int[n];
        int[] array2 = new int[n];
        int[] array3 = new int[n];

        Random rand = new Random();
        long time = System.DateTime.Now.Ticks;
        for (int i = 0; i < n; i++)
        {
            array2[i] = rand.Next(n);
            array3[i] = rand.Next(n);
            array1[i] = array2[i] * array3[i];
        }
        double delta = (System.DateTime.Now.Ticks - time);
        Console.WriteLine(delta);
    }
```

- Agrupar en bloques:

  Un último esquema que se puede utilizar aprovecha el concepto de bloques en caché. Si sabemos que la caché va a ir a buscar bloques de un determinado tamaño, podemos modificar el programa para que trabaje siempre con conjuntos de datos de ese tamaño, y no mayor. En este ejemplo en que se multiplican dos matrices, se está continuamente accediendo a gran parte de los valores de las tres matrices:

```
public static void multiplyNonBlocking(int n)
    {
```

```
            int[,] matriz1 = new int[n, n];

            int[,] matriz2 = new int[n, n];

            int[,] matriz3 = new int[n, n];

            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    matriz1[i, j] = 0;
                    matriz2[i, j] = 2;
                    matriz3[i, j] = 3;
                }
            }

            long time = System.DateTime.Now.Ticks;
            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    int r = 0;
                    for (int k = 0; k < n; k++)
                    {
                        r =  matriz2[i, k] * matriz3[k, j];
                    }
                    matriz1[i, j] += r;
                }
            }
            double delta = (System.DateTime.Now.Ticks - time);
            Console.WriteLine(delta);
        }
```

Para aprovechar el concepto de bloque, el algoritmo de multiplicación se puede separar en submultiplicación de bloques de menor tamaño (parámetro B en el ejemplo) y así mejorar el rendimiento:

```
    public static void multiplyBlocking(int n, int B)
        {
            int[,] matriz1 = new int[n, n];

            int[,] matriz2 = new int[n, n];

            int[,] matriz3 = new int[n, n];

            for (int i = 0; i < n; i++)
            {
                for (int j = 0; j < n; j++)
                {
                    matriz1[i, j] = 0;
                    matriz2[i, j] = 2;
                    matriz3[i, j] = 3;
                }
            }

            long time = System.DateTime.Now.Ticks;
            for (int ii = 0; ii < n; ii+=B)
            {
                for (int jj = 0; jj < n; jj += B)
                {
                    for (int kk = 0; kk < n; kk += B)
                    {
                        for (int i = ii; i < Math.Min(ii + B - 1, n); i++)
                        {
                            for (int j = jj; j < Math.Min(jj + B - 1, n); j++)
                            {
                                int r = 0;
```

```
                            for (int k = kk; k < Math.Min(kk + B - 1, n); k++)
                            {
                                r = matriz2[i, k] * matriz3[k, j];
                            }
                            matriz1[i, j] = matriz1[i, j] + r;
                        }
                    }
                }
            }
        }
        double delta = (System.DateTime.Now.Ticks - time);
        Console.WriteLine(delta);
    }
```

5. Ejercicios

Dirección	Label	Instrucción/Dato	
	DATA:		
0	i	0	
1	n	3	
2	sum	0	
3	arreglo	1	
4		2	
5		3	
6		4	
7		5	
	CODE:		
8	start:	MOV	A,(i)
9		MOV	B,(n)
10		CMP	A,B
11		JGE	end
12		MOV	B,arreglo
13		ADD	B,A
14		MOV	A,(sum)
15		ADD	A,(B)
16		MOV	(sum),A
17		MOV	A,(i)
18		ADD	A,1
19		MOV	(i),A
20		JMP	start

Para el código anterior del computador básico Von Neumann (palabras de 16 bits), asumiendo que el Program Counter comienza desde la dirección 8, responda las siguientes preguntas:

- Suponga que se le agrega al computador básico una caché unified de tamaño 4 palabras, con bloques de 2 palabras y ocupando correspondencia fully associative con algoritmo de reemplazo LFU. Escriba los estados de la caché a medida que avanza el programa.
- Suponga que se le agrega al computador básico una caché unified de tamaño 16 palabras, con bloques de 4 palabras y ocupando correspondencia 2-way associative con algoritmo de reemplazo LRU en cada conjunto. Escriba los estados de la caché a medida que avanza el programa.

- Suponga que se le agrega al computador básico una caché unified de tamaño 8 palabras, con bloques de 2 palabras y ocupando correspondencia directly mapped. Escriba los estados de la caché a medida que avanza el programa.
- Suponga que se le agrega al computador básico una caché split, en que la parte de instrucciones y de datos son de tamaño 4 palabras, con bloques de 2 palabras y ocupando correspondencia directly mapped. Escriba los estados de la caché a medida que avanza el programa.

6. Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 5: Large and Fast: Exploiting Memory Hierarchy.