5. Jerarquía de Memorias

Introducción

Lo que vamos a ver aquí es que en cuanto a la performance de una computadora, lo que más influye negativamente a la misma es la **memoria**. Ésto se debe a que es difícil tener memorias grandes y rápidas; normalmente las computadoras tienen memorias pequeñas y lentas en comparación a su uso.

Es por esto que, para optimizar el uso de las memorias existentes y “simular” una memoria rápida y grande, se tiene lo que se llama **jerarquía de memoria**.

Conceptos básicos

Tenemos los siguientes conceptos que nos servirán para entender la lógica y estructura básica de una jerarquía de memorias.

Principio de Localidad

El principio de localidad aplicado a programas, nos dice que solo una parte pequeña de la memoria del programa se accede con frecuencia en un momento dado.

Existen dos tipos de localidades:

- Temporal: es la tendencia que tienen los programas de acceder a un elemento de datos/instrucción que ya se accedió anteriormente. Si se accedió a un elemento, es probable que vuelva a accederse de nuevo en el futuro.

  ▼ Ejemplo

  Los ciclos/bucles hacen que se accedan a datos o instrucciones repetidas veces. Ésto demuestra una gran localidad temporal, ya que se se tiende a volver a usar esos valores que se consultan repetidas veces.

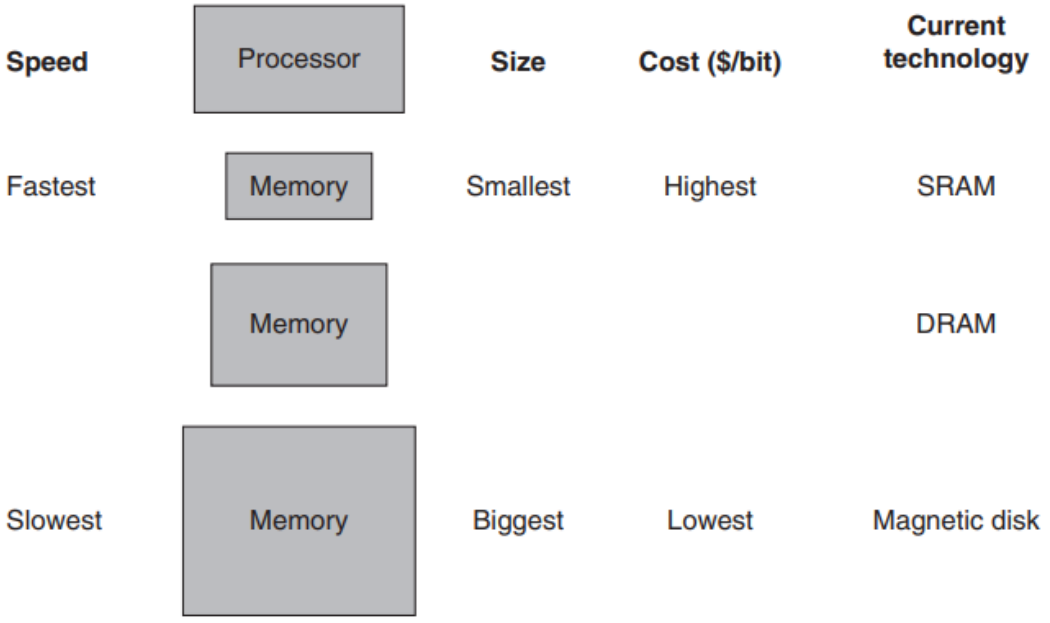
- Espacial: es la tendencia que tienen los programas de acceder a elementos de datos/instrucciones que entre sí, son muy cercanos en relación a su dirección de memoria.

  ▼ Ejemplo

  Si accedemos al elemento de una matriz, es probable que se intente acceder a otros elementos adyacentes en la misma matriz, con memoria cercana al primer elemento que se accedió.

Básicamente, la idea que debemos rescatar de esto es que los programas no acceden a todos sus datos y código de manera uniforme en un momento dado; se prioriza lo que se accede con más frecuencia, aprovechando la localidad para mejorar el rendimiento.

💪 **¿Por qué este principio es importante?**
Veamos la estructura básica de una jerarquía de memorias:



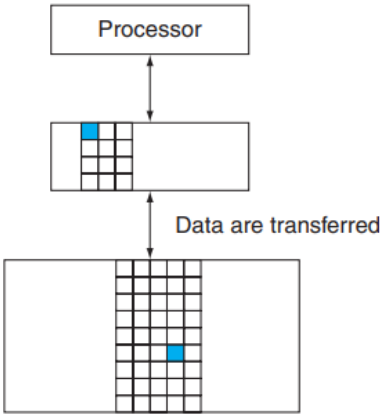
En este diagrama, la memoria de una computadora se ordena en niveles, donde cada nivel tiene memorias con velocidades y capacidades de almacenamiento diferentes. En particular, la memoria en el nivel más rápido es la más **pequeña**, **costosa** y **cercana al procesador**.

Ésta estructura nos muestra que la jerarquía de memoria está diseñada para aprovechar la localidad en los programas. Los datos e instrucciones más accedidos, se van a almacenar en los niveles más rápidos y cercanos al procesador; todos los demás datos, se almacenan en los niveles más bajos.

⚠️ Una jerarquía de memorias tiene muchos niveles, pero es importante entender el funcionamiento de dos niveles centrales, que son los que forman parte de la transferencia de bloques que veremos ahora. Éstos dos niveles en los que nos enfocaremos serán el más alto (más cercano al procesador) y uno de bajo nivel.

Jerarquía de memoria con dos niveles

Siguiendo la idea de que tenemos dos niveles centrales, analicemos más como sería la estructura de cada uno de ellos:



- **Nivel alto**: compuesta por memoria cache/SRAM. Nivel mas rápido y pequeño. Almacena lo que se llama **bloques** de información, que contienen datos e instrucciones accedidos con mucha frecuencia (o que tienen muchas chances de ser accedidos en un futuro).

  Cuando se realiza una petición de información, y se busca ese dato buscado en el nivel alto y se encuentra, se llama a la petición un **hit.** Un hit dura un ciclo de reloj.

- **Nivel bajo**: compuesta por memoria DRAM/principal. Nivel que almacena **bloques** también, pero contiene mayor cantidad de los mismos que un nivel

alto. Las peticiones de información serán de menor velocidad (lógico por lo que vimos ya) y si una petición no se encuentra en el nivel alto pero si en el nivel bajo, se llama a la petición un **miss**. Un miss dura 100 ciclos de reloj

Existen otros conceptos importantes como:

1. **hit rate**: es una proporción de los accesos a memoria que tuvieron éxito al buscar la información solicitada en el nivel más alto. Hit rate: hits/accesses (ejemplo, tuvimos 1000 accesos, y sólo en 950 de ellos hubo hits, entonces hit rate: 950/1000 = 0.95 de las veces le pegó).
2. **miss rate**: es la proporción de los accesos a memoria que buscaron información en el nivel más alto pero esa información no fue encontrada, por lo que se buscó en el nivel más bajo. Se puede calcular fácilmente haciendo **1 - hit rate**. En el ejemplo anterior, sería 1-0.95 = 0.05
3. **hit time**: calcula cuánto tiempo se necesita para acceder a un dato que está en el nivel más alto de memoria si la petición es un **hit**.
4. **miss penalty**: tiempo que se necesita para manejar un **miss**. Es decir, si se produce un **miss** ya que una información no se encontró en el nivel más alto, nos dirá el tiempo necesario para: acceder al nivel más bajo en busca de la información pedida + copiarlo en el nivel alto para su posterior acceso y entrega al procesador. (por eso tenemos el “Data is transferred” en la figura de arriba)

> 💡 Estos conceptos son importantes porque nos ayudan a analizar el desempeño de una jerarquía de memorias.

Tecnologías usadas en jerarquía de memorias

Hoy en día hay cuatro tecnologías usadas en jerarquía de memorias. Entre ellas:

1. **SRAM o caché**

   Comprende un circuito integrado que representa un arreglo de memoria con un sólo puerto de acceso para lectura o escritura. Es la tecnología que usan los niveles más cercanos al procesador.

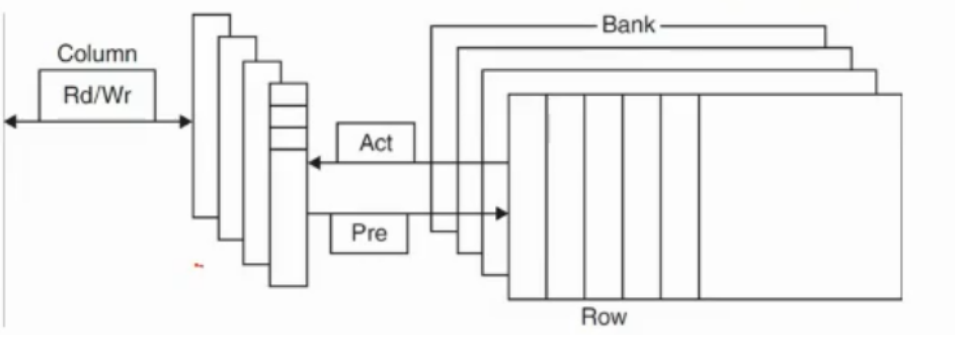
   Éste tipo de tecnología tiene un costo altísimo; la información puede ser guardada durante bastante tiempo con una necesidad mínima de energía, no requiere refrescos.

La transferencias de datos entre la caché/SRAM al procesador es a través de palabras de 32 o 64 bits.

2. **DRAM o memoria principal**

   Almacena la información en celdas como cargas. Es más económico que las SRAM pero necesitan de refrescos para mantener la información a lo largo del tiempo.

   Las cargas forman un arreglo rectangular de bits, donde la información se accede por filas.



La address que se desea leer se pasa a múltiples bancos dentro de la DRAM de forma paralela para buscar el dato que se quiere.

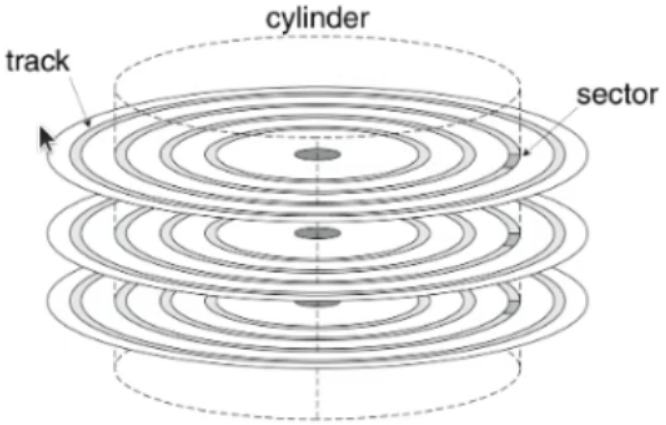
3. **FLASH**

   Usada comúnmente en dispositivos móviles y pendrives, la información que guarda este tipo de tecnología es sólo para lectura. Es un tipo de almacenamiento no volátil, capaz de mantener la información aún cuando no tiene energía. Son pequeñas y muy económicas.

   Algo importante de este tipo de memorias es que su capacidad de escritura es finita; se desgasta mucho con el tiempo.

4. **MEMORIA DISCO**

   No volátil, compuesto de varios discos, tracks y sectores a lo largo de un cilindro. Gracias a un cabezal, accedemos a los sectores deseados para escribir/leer información.



Los sectores tienen la data + información del sector + códigos correctores de errores.

Memoria Caché / SRAM

Como vimos, las cachés son aquellas memorias más cercanas la procesador. Veamos cómo funcionan las mismas.

Mapeo Directo (o 1 vía)

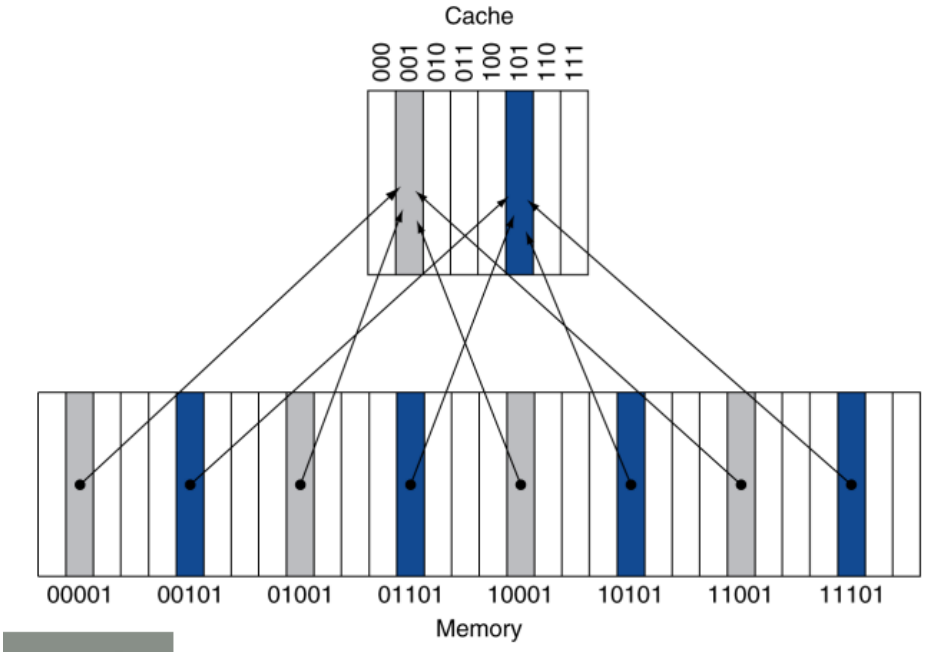
Para determinar a qué ubicación en la caché se asigna una dirección de memoria en particular, se utiliza una función de mapeo. En este caso, la función de mapeo utiliza la operación de módulo.

▶️ **(Dirección del Bloque) modulo (Número de bloques en la caché)**

- El valor de **Dirección del Bloque** es la dirección de memoria que tiene el bloque en la memoria principal.
- El valor de **Número de bloques en la caché** esta en potencias de dos

Como se usa potencia de dos para el número de bloques en la caché, el cálculo se simplifica a simplemente observar bits menos significativos de la dirección del bloque en la memoria principal.

Ejemplo: Tenemos la siguiente estructura, donde la caché tiene un total de 8 bloques ($8 = 2^3$), por ende, para saber qué dirección de la memoria principal se mapeará a qué dirección de la caché, bastará con tan solo observar los tres bits menos significativos de la dirección a mapear.



- Si tenemos la dirección de bloque 00001 (en binario) y queremos asignarle una ubicación en la caché, agarramos los 3 bits menos significativos: 001 (en binario).
- Luego, calculamos el valor decimal de estos 3 bits, que en este caso es 1.
- Ahora, la dirección de bloque 00001 se asigna a la ubicación 1 en la caché.

Tag y Valid Bit

¿Pero cómo sabemos si un dato/palabra ya está en un bloque de caché o no?

Lo que se hace es que se agregan un set de **tags** a la caché. Éstas tags contienen la información necesaria para identificar si una palabra contenida en la caché corresponde a la palabra requerida.

La tag esta comprendida por los bits más significativos de la dirección de memoria (es decir, los bits que no se usan para indexar los bloques de la caché).

En el ejemplo anterior, las tags serían los dos primeros bits de las direcciones en la memoria principal, que nos ayudarán a buscar la palabra deseada dentro del bloque de la caché.

¿Y si tenemos un bloque de caché sin data?

A veces tenemos bloques en la caché que no tienen información válida. Por ejemplo, cuando un procesador se enciende, la caché no tiene data relevante o está vacía directamente. Por esto, se usa un **valid bit** en cada entrada de caché que se establece en 1 cuando la entrada tiene datos válidos (0 c.c.).

Valid bit: 1 = present, 0 = not present
Initially 0

Entonces, al inicio de un programa, la memoria caché si es vacía, al tener los bits de validez en cero, no genera problemas si se hace una petición a la misma de un dato en específico. Es decir, se producirá un miss y se agregará este dato a la caché y su bit de validez pasará a ser 1.

Ejemplo:

Estado inicial de la caché

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

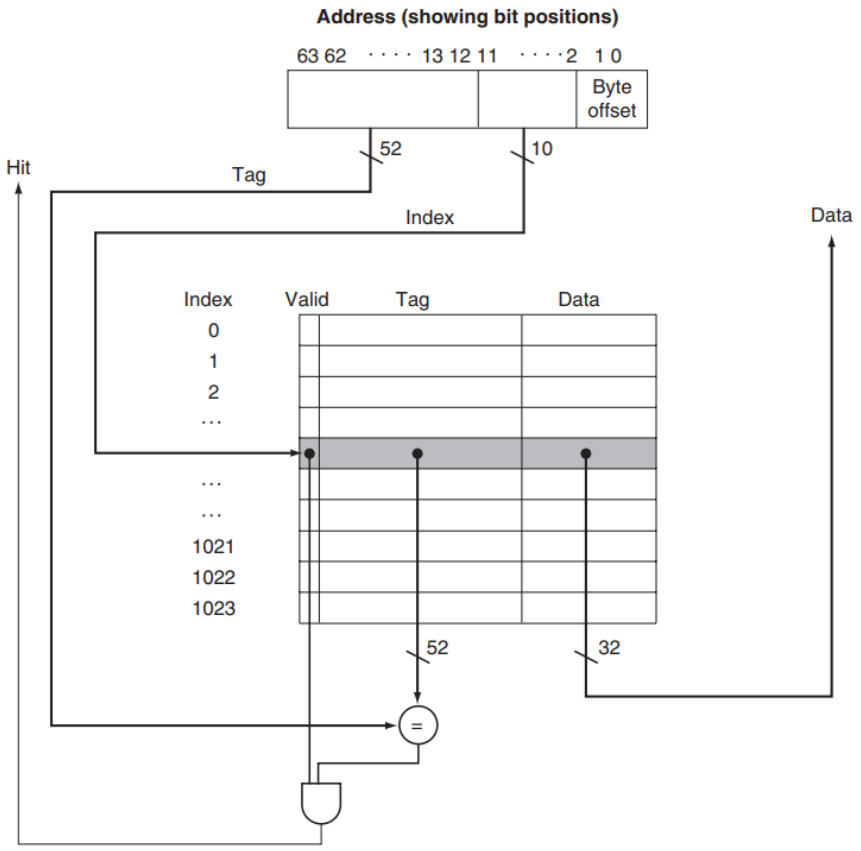
Se solicita un dato que genera un miss, ya que la caché está vacía

Word addr	Binary addr	Hit/miss	Cache block
22	10 110	Miss	110

Se agrega este dato a la caché

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem[10110]
111	N		

Cómo luce una address en un contexto de una caché más grande:



Ejemplo de address para una caché de 1024 palabras. Notar que el índice es de 10 bits porque 2^{10} = 1024, dejando además, 64-10 = 52 bits para la tag.

Caché Misses

Cuando ocurre un hit al buscar un tipo de dato, la CPU procede de forma normal sin problemas, la diferencia ocurre cuando tenemos un miss.

En nuestro procesador tenemos dos componentes de memoria que son cachés (instruction memory, data memory), por lo tanto se pueden producir misses de instrucciones o de datos. Ambos tipos de misses no se diferencian tanto en cómo responde el procesador ante ellos.

Básicamente, como hemos visto, ante un miss lo que hace el procesador es entrar en stalls. Éste tiempo de stall se invierte para buscar el valor que no está en cache en los otros niveles de jerarquía (como por ej memoria principal). Una vez el dato se encuentra, se resetea el proceso de fetch en caso de instrucciones, o se completa el acceso al dato deseado en caso de un dato.

Proceso de escritura en cachés

Manejo de inconsistencias en escrituras

Puede pasar que algún dato sólo se escriba en la memoria caché y no en la memoria principal (por ejemplo, ante uso de instrucción STUR). Para evitar estas inconsistencias, se tienen las siguientes formas de manejar los writes en memoria.

1. Write through: se escribe el dato en caché y en memoria principal al mismo tiempo. Produce un rendimiento deficiente ya que en cada escritura debería esperarse a que otra instrucción deje de escribir en memoria principal!
2. Write buffer: se tiene un buffer de escritura que almacena temporalmente los datos mientras esperan ser escritos en la memoria principal. O sea, al escribir en caché, se escribe en el bufer también, entonces se deja que se puedan ejecutar otras instrucciones entre medio que escriban en memoria principal (para no ralentizar). Luego, lo que esta en el bufer pasa a escribirse a memoria principal, liberando las entradas en el bufer.

   Solo se producen stalls si el bufer se llena, ya que ahí se debería esperar a que el mismo tenga un espacio disponible.
3. Write back: ante una escritura, escribir sólo en caché. El bloque que se haya actualizado en caché se lo diferenciará de los demás como dirty, haciendo alusión de que los datos en ese bloque no están incluidos en la memoria principal. Entonces, cuando se quiera reemplazar ese bloque de caché por otro, se actualizará la memoria principal con la información que se escribió en la cache, así una vez hecho el reemplazo, no se pierde información.

> **Chat GPT:**
>
> Aquí está la secuencia de eventos:
>
> 1. Cuando se realiza una escritura en un bloque de memoria que ya está en la caché y se utiliza un esquema de escritura posterior, la caché se actualiza con el nuevo valor. Esto significa que la versión en la caché del bloque se modifica para reflejar los cambios.
> 2. La versión modificada del bloque en la caché puede diferir de la versión en la memoria principal, ya que la memoria principal no se actualiza de inmediato con la

> nueva información. Los datos en la caché se vuelven "sucios" en el sentido de que difieren de la memoria principal.
>
> 3. La caché seguirá utilizando la versión modificada del bloque hasta que ese bloque se elimine o se reemplace por otro bloque en la caché. El reemplazo de un bloque puede ocurrir por varias razones, como la política de reemplazo de la caché o la necesidad de liberar espacio para nuevos datos. Cuando se realiza el reemplazo, la versión modificada del bloque se escribe en la memoria principal para que los datos en la memoria principal y en la caché estén nuevamente en concordancia.

Manejo de misses en escritura | bldo no se entiende nada esta unidad

Cuando se produce una escritura que resulta en un **miss** (ej: en un store, que surge cuando se quiere escribir en un bloque que no está en la caché) se sigue la política de “Write Allocation” que nos dice cómo actúa la caché dependiendo de la configuración de la misma:

- Caso write-through:
  - Allocate on miss: al tener un miss, se reserva un espacio en la caché y se busca el bloque faltante en la memoria principal, para luego escribirlo en la caché. Luego, el bloque que se trajo para escribirlo en cache, se reescribe la porción modificada en la principal.
  - Write around: lo mismo que allocate on miss pero no se trae el bloque desde la principal a la caché.
- Caso write-back
  - Generalmente se usa allocate on miss, es decir, se trae el bloque a la caché en caso de no estar para escribirse.

Desempeño de Caché

Vamos a ver como medir y analizar el **desempeño** de las cachés y dos enfoques para mejorar el mismo.

Dado un programa, en su ejecución se deben tener en cuenta dos cosas:

- Hit time de la caché
- stalls de la memoria: que vienen de los caché misses donde:

$$\text{Memory stall cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$+ \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Ejemplo

Si tenemos:

- miss rate de instrucciones en cache: 2% de instrucciones no se encuentran en caché
- miss rate de datos en cache: 4% de datos no se encuentran en caché
- Miss penalty: 100 ciclos se stallea si se produce miss
- Base CPI (ideal cache): 2 ciclos por instrucción (de ahí viene CPI), o sea es lo que necesita minimamente para ejecutar cada instrucción cuando la caché funciona joya.
- Load & Store son el 36% de instrucciones

Entonces calculamos el número de ciclos de reloj adicionales que necesitaríamos debido a los posibles fallos

Miss cycles por instrucción:

1. misses en data: 0.04 (miss rate de datos) * 100 (miss penalty por fallo) * 0.36 (porcentaje de instrucciones de carga/almacenamiento) = 1.44 ciclos por fallo en datos
2. misses instrucciones: 0.02 * 100 = 2 ciclos por fallo en instrucciones

Cálculo definitivo de desempeño en CPU

- CPI actual: 2 (CPI base) + 2 (ciclos por fallo en instrucciones) + 1.44 (ciclos por fallo en datos) = 5.44

En definitiva, para comparar con el CPI que se tenia anteriormente que era 2, tenemos que

CPU ideal: 5.44/2 = 2.72, por ende, el CPI actual es 2.72 veces mas lento que el anterior.

Análisis de acceso a datos promedio (AMAT)

Puede pasar que el procesador sea más rápido pero el sistema de memoria no. El AMAT se centra en medir el tiempo promedio de acceso a la memoria.

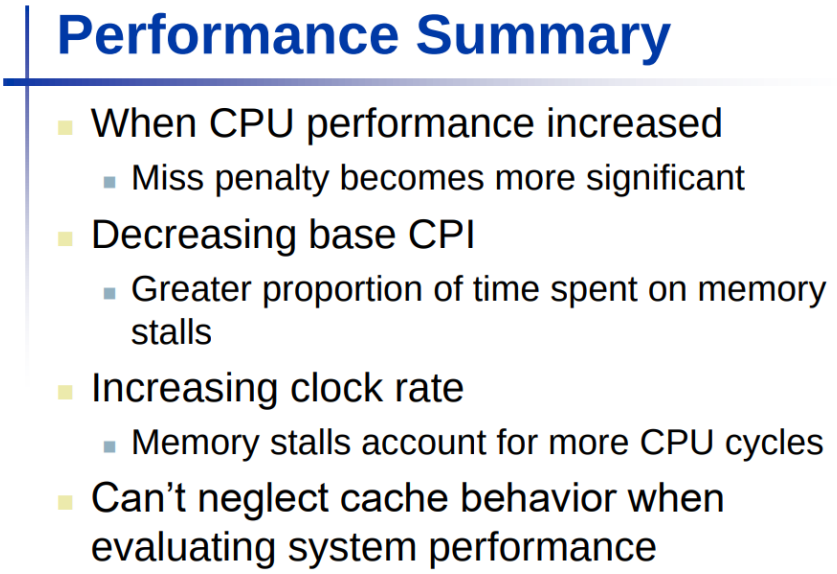
Para ello, teniendo en cuenta los hits y misses en la caché, calculamos:

AMAT = hit time + miss rate * miss penalty.

Ejemplo: en un CPU con un clock de 1ns, un hit time = 1 ciclo, un miss penalty = 20 ciclos, y un miss rate de instrucciones de 5%:

AMAT = 1 + 0.05*20 = 2ns = 2 ciclos por instrucción

Resumen



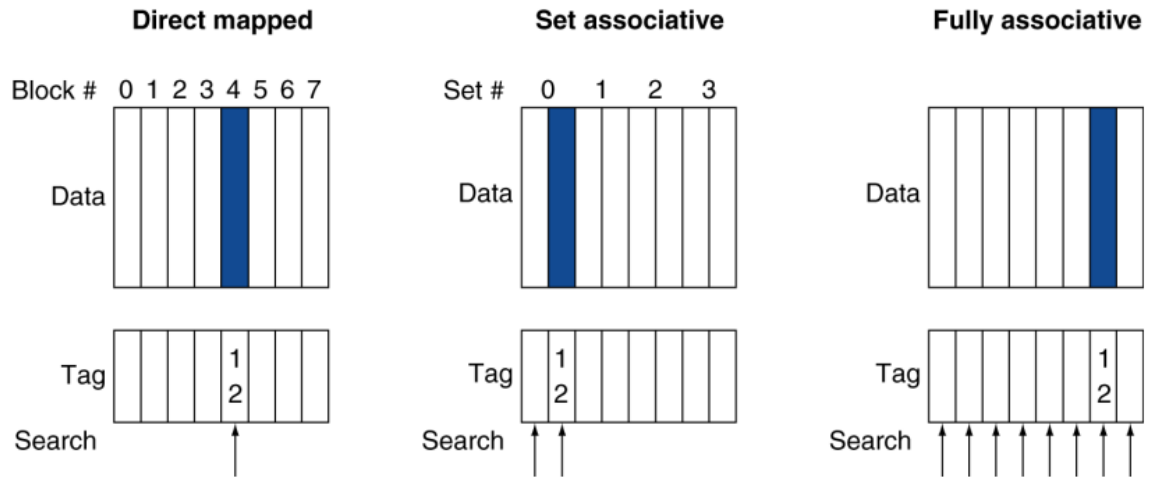
Set Associative Caché

En la estructura de la caché que veníamos viendo, el mapeo de bloques se hacía de forma directa. Ahora, veremos otro tipo de esquema que es el **fully associative**: un

bloque de memoria puede ser asociado a cualquier entrada en la caché. Para encontrar un bloque en una cache **fully associative**, todas las entradas de la caché deben ser analizadas porque un bloque puede localizarse en cualquiera de ellas.

En este tipo de esquemas, tenemos un **conjunto fijo de ubicaciones disponibles** para cada bloque de memoria. Es decir, un bloque se busca entre las ubicaciones dentro de ese conjunto fijo.

Definiremos n-vías a la cantidad n de ubicaciones dentro de un conjunto de ubicaciones disponibles. En el esquema visto de direct-mapped es 1 vía.



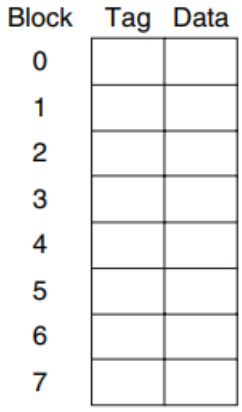
direct-mapped: 1 vía, el del medio es 2 vías, y el tercero es igual cantidad de vías como de bloques

Para saber el conjunto que contiene a un bloque de memoria en específico se calcula:

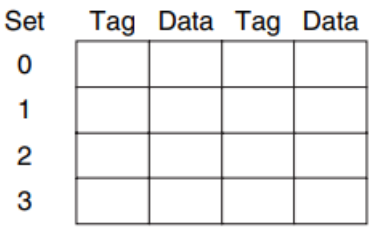
$$\text{(Block number) modulo (Number of } \textit{sets} \text{ in the cache)}$$

Dado que un bloque de memoria puede ubicarse en cualquiera de los elementos dentro de un conjunto en una caché set-associative, es necesario buscar entre todas las etiquetas (tags) de los elementos en ese conjunto para encontrar una coincidencia.

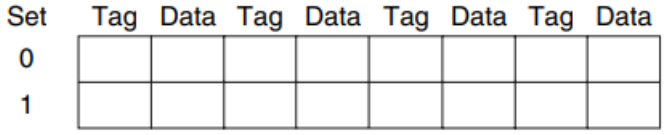
One-way set associative (direct mapped)



Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)