

# Conceptos básicos de cachés

# Temario

- Introducción.
- Cachés de mapeo directo.
- Overhead de un caché.
- Manejo de escrituras.
- Organización de la memoria.
- Cachés asociativos n-way.
- Cachés totalmente asociativos.

# Introducción

- La memoria caché es el nivel de memoria situada entre el procesador y la memoria principal.
- Se comenzaron a usar a fines de los años 60s.
- Todas la computadoras modernas incluyen cachés.



# Tipos de memorias cachés

- **Cachés de datos.** Guardan los últimos datos referenciados.
- **Cachés de instrucciones.** Guardan las últimas instrucciones ejecutadas.
- **Cachés unificados.** Guardan los últimos datos e instrucciones utilizadas.
- **Cachés de trazas (trace caches).** Guardan secuencias de instrucciones para ejecutar que no son necesariamente adyacentes.

# Ejemplo

- El tamaño del bloque es 1 palabra (4 bytes).
- Las peticiones de la CPU son de 1 palabra.
- La memoria caché ya tiene los siguientes datos:
- Se pide  $X_n$
- Se produce una falla.
- Se trae  $X_n$  de la memoria.
- Se inserta en el caché.

|           |
|-----------|
| $X_4$     |
| $X_1$     |
| $X_{n-2}$ |
|           |
| $X_{n-1}$ |
| $X_2$     |
| $X_n$     |
| $X_3$     |



## 2 preguntas 2

1. ¿Cómo se sabe si un dato está en el caché?
  2. Si está, ¿Dónde se encuentra?
- Estrategias:
    - a) Caché de mapeo directo (direct mapped cache).
    - b) Caché asociativo total (fully associative cache).
    - c) Caché asociativo por conjunto (n-way set associative cache).

# Caché de mapeo directo

- A cada bloque en la memoria se le asigna un bloque (línea) en el caché.
- Problema: dada una dirección de un bloque en la memoria hay que encontrar en que bloque del caché se va a guardar.
- Hay 2 soluciones (al menos):
  1. Usar fórmulas.
  2. Usar la dirección del bloque en la memoria.



# Método 1

- Datos de entrada:
  - $a$  – la dirección del item en la memoria.
  - $k$  – el tamaño de bloque en bytes.
  - $n$  – el número de bloques que tiene el caché.
- Se usan la siguientes fórmulas:
- $d = a \text{ div } k$
- $b = d \text{ mod } n$
- Conclusión: el item con dirección  $a$  en la memoria se guarda en el bloque  $b$  del caché.



## Método 2

- La dirección del ítem en la memoria,  $a$ , se convierte a binario y se divide en tres partes.
  1. El offset. Ocupa los bits más bajos. Indica en qué byte dentro del bloque se almacena el ítem. Su tamaño es  $\log_2(k)$ , donde  $k$  es el tamaño del bloque en bytes.
  2. El índice. Ocupa los bits intermedios. En mapeo directo indica en qué bloque del caché se va a guardar el ítem. Su tamaño es  $\log_2(n)$ , donde  $n$  es el número de bloques del caché.

## Método 2

3. La etiqueta. Ocupa los bits altos. Indica de que dirección en la memoria viene el item (esto se verá a continuación). Su tamaño es lo que no se usa para el offset ni para el índice.



# Ejemplo 1

- Dados los siguientes datos:
  - $k = 1$  (el caché tiene bloques de 1 byte).
  - $n = 8$  (el caché tiene 8 bloques).
- ¿Qué bloque dentro del caché le toca a un dato con dirección 57 (i.e.  $a = 57$ )?
- Suponer direcciones de 8 bits.
- Respuesta:



# Ejemplo 1

- Método 1.
- Datos:  $a = 57$ ,  $k = 1$ ,  $n = 8$ .
  - $d = 57 \text{ div } 1 = 57$
  - $b = 57 \text{ mod } 8 = 1$
  - Le toca el bloque 1 en el caché.

# Ejemplo 1

- Método 2.
- Datos:  $a = 57 = 00111001$ ,  $k = 1$ ,  $n = 8$ .
  - Tamaño del offset =  $\log_2(k) = \log_2(1) = 0$
  - Tamaño del index =  $\log_2(n) = \log_2(8) = 3$
  - Tamaño de la etiqueta =  $8 - 3 = 5$
- Respuesta: la dirección se separa en:
  - $a = 00111-001$
  - Index = 001
  - Etiqueta = 00111
  - Le toca el bloque 1 en el caché.

## Ejemplo 2

- Dados los siguientes datos:
  - $k = 4$  (el caché tiene bloques de 4 bytes).
  - $n = 8$  (el caché tiene 8 bloques).
- ¿Qué bloque dentro del caché le toca a un dato con dirección 57 (i.e.  $a = 57$ )?
- Suponer direcciones de 8 bits.



## Ejemplo 2

- Método 1.
- Datos:  $a = 57$ ,  $k = 4$ ,  $n = 8$ ,
- Respuesta:
  - $d = 57 \text{ div } 4 = 14$
  - $b = 14 \text{ mod } 8 = 6$
  - Le toca el bloque 6 en el caché.

## Ejemplo 2

- Método 2.
- Datos  $a = 57 = 00111001$ ,  $k = 4$ ,  $n = 8$ .
  - Tamaño del offset =  $\log_2(k) = \log_2(4) = 2$
  - Tamaño del index =  $\log_2(n) = \log_2(8) = 3$
  - Tamaño de la etiqueta =  $8 - 3 - 2 = 3$

## Ejemplo 2

- Respuesta: la dirección se separa en:
  - $a = 001-110-01$
  - Offset = 01
  - Index = 110
  - Etiqueta = 001
  - Le toca el bloque 6 en el caché.



# Etiqueta y bit válido

- Problema: a varias direcciones en la memoria les corresponde un mismo bloque en el caché.
- Se necesita saber si el item en el caché es o no el item buscado.
- A cada bloque del caché se le agrega una **etiqueta**.
- La etiqueta tiene la información para identificar si el item en el caché es el dato buscado.
- La etiqueta tiene los bits altos de la dirección del item (la parte que no se usa para el offset ni para el índice).

# Etiqueta y bit válido

- Además se necesita saber si el bloque tiene información válida o no.
- Cada bloque tiene un bit llamado **bit válido**.



# Ejemplo

- Direcciones de 8 bits.
- Tamaño de la memoria caché: 8 bloques.
- Tamaño del bloque: 1 byte.
- Offset =  $\log_2(1) = 0$  bits.
- Index =  $\log_2(8) = 3$  bits.
- Etiqueta =  $8 - 3 - 0 = 5$  bits.
- Inicialmente la memoria caché está vacía (para todas las entradas bit válido = falso).



# Ejemplo

- Suponer que se reciben las siguientes peticiones de direcciones: 22, 26, 22, 26, 16, 3, 16 y 18.
- Simular el comportamiento del caché y determinar la tasa de éxitos.

# Estado inicial

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

# Se pide la dirección $22_{10}$

- $22_{10} = 10110_2 = 10-110$ .
- Se busca en el bloque 110 y se produce una falla.
- Se carga el dato en el bloque 110. Tasa de éxito: 0/1.

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

| Index | V | Tag        | Data                    |
|-------|---|------------|-------------------------|
| 000   | N |            |                         |
| 001   | N |            |                         |
| 010   | N |            |                         |
| 011   | N |            |                         |
| 100   | N |            |                         |
| 101   | N |            |                         |
| 110   | Y | $10_{two}$ | Memory( $10110_{two}$ ) |
| 111   | N |            |                         |



# Se pide la dirección $26_{10}$

- $26_{10} = 11010_2 = 11-010$ .
- Se busca en el bloque 010 y se produce una falla.
- Se carga el dato en el bloque 010. Tasa de éxito 0/2.

| Index | V | Tag               | Data                           |
|-------|---|-------------------|--------------------------------|
| 000   | N |                   |                                |
| 001   | N |                   |                                |
| 010   | N |                   |                                |
| 011   | N |                   |                                |
| 100   | N |                   |                                |
| 101   | N |                   |                                |
| 110   | Y | $10_{\text{two}}$ | Memory( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                |

| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | N |                   |                                 |
| 001   | N |                   |                                 |
| 010   | Y | $11_{\text{two}}$ | Memory ( $11010_{\text{two}}$ ) |
| 011   | N |                   |                                 |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |

# Se pide la dirección $22_{10}$

- $22_{10} = 10110_2 = 10-110$ .
- Se busca en el bloque 110.
- Se compara la etiqueta con 10 (la parte alta de la dirección).
- Se produce un éxito. Tasa de éxito: 1/3.

| Index | V | Tag        | Data                     |
|-------|---|------------|--------------------------|
| 000   | N |            |                          |
| 001   | N |            |                          |
| 010   | Y | $11_{two}$ | Memory ( $11010_{two}$ ) |
| 011   | N |            |                          |
| 100   | N |            |                          |
| 101   | N |            |                          |
| 110   | Y | $10_{two}$ | Memory ( $10110_{two}$ ) |
| 111   | N |            |                          |

# Se pide la dirección $26_{10}$

- $26_{10} = 11010_2 = 11-010$ .
- Se busca en el bloque 010.
- Se compara la etiqueta con 11 (la parte alta de la dirección).
- Se produce un éxito. Tasa de éxito: 2/4.

| Index | V | Tag        | Data                     |
|-------|---|------------|--------------------------|
| 000   | N |            |                          |
| 001   | N |            |                          |
| 010   | Y | $11_{two}$ | Memory ( $11010_{two}$ ) |
| 011   | N |            |                          |
| 100   | N |            |                          |
| 101   | N |            |                          |
| 110   | Y | $10_{two}$ | Memory ( $10110_{two}$ ) |
| 111   | N |            |                          |



# Se pide la dirección $16_{10}$

- $16_{10} = 10000_2 = 10-000$ .
- Se busca en el bloque 000 y se produce una falla.
- Se carga el dato en el bloque 000. Tasa de éxito 2/5.

| Index | V | Tag        | Data                     |
|-------|---|------------|--------------------------|
| 000   | N |            |                          |
| 001   | N |            |                          |
| 010   | Y | $11_{two}$ | Memory ( $11010_{two}$ ) |
| 011   | N |            |                          |
| 100   | N |            |                          |
| 101   | N |            |                          |
| 110   | Y | $10_{two}$ | Memory ( $10110_{two}$ ) |
| 111   | N |            |                          |

| Index | V | Tag        | Data                     |
|-------|---|------------|--------------------------|
| 000   | Y | $10_{two}$ | Memory ( $10000_{two}$ ) |
| 001   | N |            |                          |
| 010   | Y | $11_{two}$ | Memory ( $11010_{two}$ ) |
| 011   | N |            |                          |
| 100   | N |            |                          |
| 101   | N |            |                          |
| 110   | Y | $10_{two}$ | Memory ( $10110_{two}$ ) |
| 111   | N |            |                          |

# Se pide la dirección $3_{10}$

- $3_{10} = 00011_2 = 00-011$ .
- Se busca en el bloque 011 y se produce una falla.
- Se carga el dato en el bloque 011. Tasa de éxito 2/6.

| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | Y | $10_{\text{two}}$ | Memory ( $10000_{\text{two}}$ ) |
| 001   | N |                   |                                 |
| 010   | Y | $11_{\text{two}}$ | Memory ( $11010_{\text{two}}$ ) |
| 011   | N |                   |                                 |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |

| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | Y | $10_{\text{two}}$ | Memory ( $10000_{\text{two}}$ ) |
| 001   | N |                   |                                 |
| 010   | Y | $11_{\text{two}}$ | Memory ( $11010_{\text{two}}$ ) |
| 011   | Y | $00_{\text{two}}$ | Memory ( $00011_{\text{two}}$ ) |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |

# Se pide la dirección $16_{10}$

- $16_{10} = 10000_2 = 10-000$ .
- Se busca en el bloque 000.
- Se compara la etiqueta con 10 (la parte alta de la dirección).
- Se produce un éxito. Tasa de éxito: 3/7.

| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | Y | $10_{\text{two}}$ | Memory ( $10000_{\text{two}}$ ) |
| 001   | N |                   |                                 |
| 010   | Y | $11_{\text{two}}$ | Memory ( $11010_{\text{two}}$ ) |
| 011   | Y | $00_{\text{two}}$ | Memory ( $00011_{\text{two}}$ ) |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |



# Se pide la dirección $18_{10}$

- $18_{10} = 10010_2 = 10-010$ .
- Se busca en el bloque 010 y se produce una falla.
- Se carga el dato en el bloque 010. Tasa de éxito 3/8.

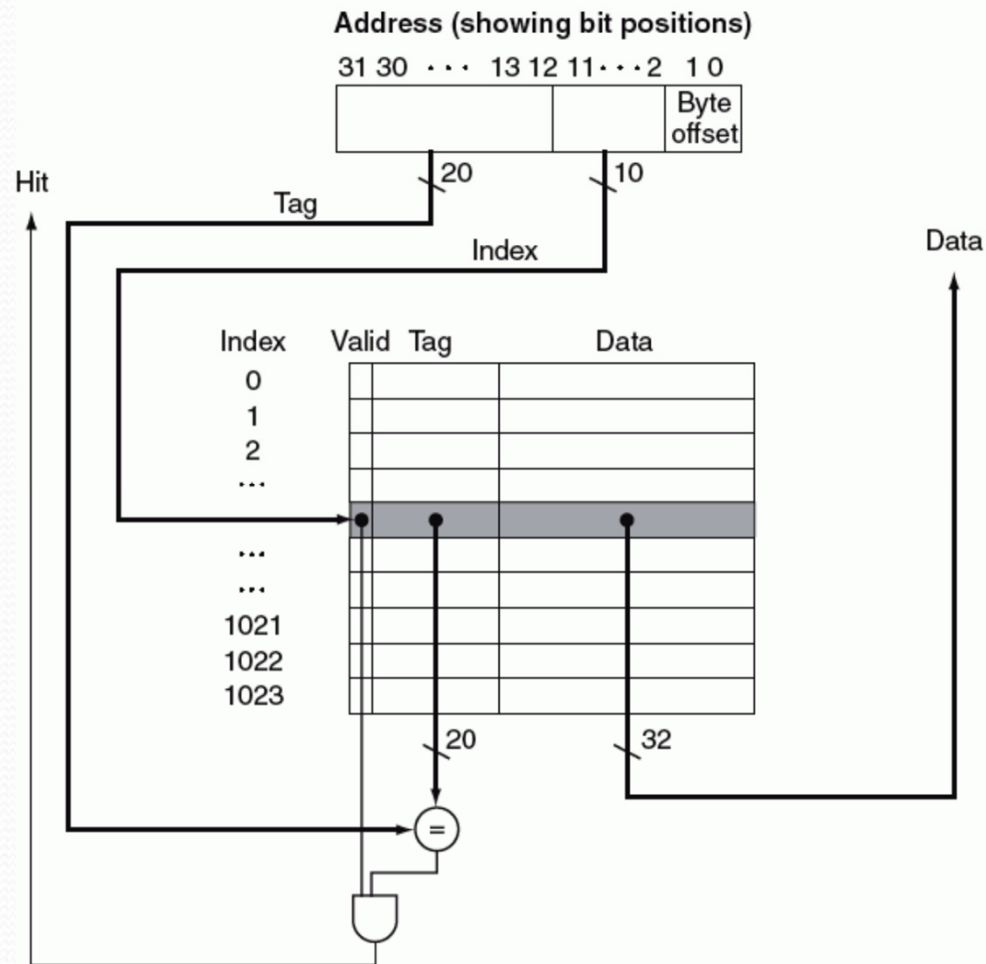
| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | Y | $10_{\text{two}}$ | Memory ( $10000_{\text{two}}$ ) |
| 001   | N |                   |                                 |
| 010   | Y | $11_{\text{two}}$ | Memory ( $11010_{\text{two}}$ ) |
| 011   | Y | $00_{\text{two}}$ | Memory ( $00011_{\text{two}}$ ) |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |

| Index | V | Tag               | Data                            |
|-------|---|-------------------|---------------------------------|
| 000   | Y | $10_{\text{two}}$ | Memory ( $10000_{\text{two}}$ ) |
| 001   | N |                   |                                 |
| 010   | Y | $10_{\text{two}}$ | Memory ( $10010_{\text{two}}$ ) |
| 011   | Y | $00_{\text{two}}$ | Memory ( $00011_{\text{two}}$ ) |
| 100   | N |                   |                                 |
| 101   | N |                   |                                 |
| 110   | Y | $10_{\text{two}}$ | Memory ( $10110_{\text{two}}$ ) |
| 111   | N |                   |                                 |

# Nomenclatura

- El nombre del caché indica el espacio disponible para guardar datos.
- No se toma en cuenta el espacio extra dedicado a las etiquetas y a los bits válidos.
- Un caché de 8KB tiene 8 kilobytes para guardar datos.
- Un cache de 64KB tiene 64 kilobytes para guardar datos.

# Caché de 4KB de mapeo directo





# Explicación

- $1K = 1024 = 2^{10}$  palabras.
- Direcciones de 32 bits.
- La dirección se divide en:
  - Índice del cache (bits 11:2) selecciona el bloque.
  - Etiqueta (bits 31:12) se compara con la etiqueta del bloque del caché.
- Éxito = válido AND (etiqueta == dirección[31:12])

# Overhead de un caché

- Overhead es el espacio extra requerido para guardar los items en un caché.
- En un caché de mapeo directo, el overhead incluye las etiquetas y los bits válidos.
- Hay al menos dos métodos para calcular el overhead de un caché:



# Overhead de un caché

- Método 1:
- Calcular el número de bits que ocupan los datos en el caché y llamarle a este número  $a$ .
- Calcular el número de bits que ocupa todo el caché (incluyendo etiquetas y bits válidos) y llamarle a este número  $b$ .
- $\text{Overhead} = ((b / a) - 1) \times 100$ .



# Overhead de un caché

- Método 2:
- Obtener el tamaño de bloque en bits y llamarle a este número  $a$ .
- Calcular  $b$  = tamaño de etiqueta en bits + 1 (por el bit válido).
- $\text{Overhead} = (b / a) \times 100$ .

# Ejemplo

- ¿Cuál es el overhead de un caché de 16 KB de datos en bloques de 4 palabras? Las direcciones son de 32 bits.



# Ejemplo método 1

- Calcular el número de bits que ocupan los datos en el caché y llamarle a este número  $a$ .
- El caché es de 16KB.
- $a = 16 \times 1024 \times 8 = 131072$  bits.
- Calcular el número de bits que ocupa todo el caché (incluyendo etiquetas y bits válidos) y llamarle a este número  $b$ .
- $b = \text{tamaño total del bloque en bits (datos + etiqueta + bit válido)} \times \text{número de bloques}$ .



# Ejemplo método 1

- Tamaño del bloque de datos = 4 palabras.
- Tamaño del bloque de datos en bits =  $4 \times 4 \times 8 = 128$  bits.
- Tamaño de etiqueta = Tamaño de la dirección – (tamaño del index + tamaño del offset).
- Tamaño del index =  $\log_2$  (número de bloques).
- Número de bloques = tamaño del cache / tamaño del bloque de datos.

# Ejemplo método 1

- Número de bloques =  $(16 \times 1024) / (4 \times 4) = 1024$  bloques.
- Tamaño del index =  $\log_2 (1024) = 10$  bits.
- Tamaño del offset =  $\log_2$  (tamaño de bloque en bytes).
- Tamaño del offset =  $\log_2 (4 \times 4) = 4$  bits.
- Tamaño de la etiqueta =  $32 - (10 + 4) = 18$  bits.
- Tamaño total del bloque =  $128 + 18 + 1 = 147$  bits.



# Ejemplo método 1

- El número de bits que ocupa todo el caché (incluyendo etiquetas y bits válidos) es:
- $b = 147 \times 1024 = 150528$  bits.
- Son 1024 bloques y cada bloque ocupa en total 147 bits.
- $\text{Overhead} = (150528 / 131072) - 1 \times 100.$
- $\text{Overhead} = 14.84$
- Conclusión: el caché ocupa un espacio extra de casi 15% dedicado a las etiquetas y bits válidos.



## Ejemplo método 2

- Calcular el tamaño de bloque en bits y llamar a este número a.
- Calcular el tamaño de la etiqueta, sumarle uno por el bit válido, y llamar a este número b.
- $a = 128$  bits.
- $b = 19$  bits.
- $\text{Overhead} = (19 / 128) \times 100 = 14.84$ .
- Conclusión: el overhead es de casi el 15%.

# Manejo de escrituras

- Una escritura a la memoria es el resultado de una instrucción sw.
- Hay dos escenarios:
  - Escritura con éxito. El dato está en el caché (y en la memoria).
  - Escritura con falla. El dato está en la memoria y no en el caché.



# Escritura con éxito

- Si el dato está en el caché surge la pregunta si el dato se debe escribir en el caché y en la memoria o solo en el caché.
- Si solo se escribe en el caché, se dice que el caché y la memoria son **inconsistentes**.
- Hay dos políticas de escritura con éxito: write-through y write-back.



# Escritura con falla

- Si el dato no está en el caché, surge la pregunta si después de escribir el dato en la memoria hay que subirlo o no al caché.
- Hay dos políticas de escritura con falla: write allocate y no write allocate.

# Write-through

- Escribir cada vez en el caché y en la memoria.
- Problema: escribir en la memoria es lento.
- Una escritura puede tomar 100 ciclos de reloj.
- Según SPECInt2000, 10% de las instrucciones son escrituras.
- Si el CPI es 1, gastar 100 ciclos en cada escritura aumenta el CPI a  $1 + 100 * 0.1 = 11$ .
- Escribir en el caché y en la memoria reduce el rendimiento 11 veces.



# Write-through

- Una solución es usar un buffer de escritura.
- El buffer de escritura es una cola que guarda datos que esperan ser escritos en la memoria principal.
- El programa escribe en el caché y en el buffer y continúa ejecutando instrucciones.
- Si el buffer está lleno, el siguiente store se detiene.
- Un procesador superescalar con scheduling dinámico puede continuar ejecutando alguna otra instrucción.



# Write-through

- Ningún buffer es suficiente si el procesador produce escrituras más rápido de lo que la memoria puede aceptarlas.

# Write-back

- Escribir solo en el caché y copiar el dato a la memoria cuando el bloque en el caché se debe reemplazar.
- El bloque en el caché tiene un bit llamado **bit sucio**.
- Si el bit sucio está prendido, hay una inconsistencia entre el caché y la memoria. Antes de ser reemplazado, el bloque se debe escribir en la memoria.
- Si el bit sucio está apagado, el bloque no fue modificado y puede ser reemplazado sin peligro.



# Comparación – write-through

- Ventajas:
- Las fallas de lectura nunca producen escrituras en la memoria principal.
- Fácil de implementar.
- La memoria principal y el caché siempre son consistentes.
- Desventajas:
- La escritura es más lenta.
- Cada escritura necesita acceso a la memoria principal.
- Requiere más ancho de banda.



# Comparación – write-back

- Ventajas:
- Las escrituras ocurren a la velocidad del caché.
- Múltiples escrituras dentro de un bloque requieren una sola escritura en la memoria principal.
- Requiere menos ancho de banda.
- Desventajas:
- Más difícil de implementar.
- La memoria principal no siempre es consistente con el caché.

# Comparación – write-back

- Las lecturas que resultan en un reemplazo pueden resultar en escrituras de bloques sucios a la memoria principal.

# Conclusión

- No hay un claro ganador entre write-through y write-back.



# Políticas de escritura con falla

- Write allocate: el bloque se carga en el caché y luego se aplica la política de escritura con éxito (write-through o write-back).
- No write allocate: el bloque se escribe (actualiza) en la memoria y **no** se carga en el caché.

# Combinaciones

- En teoría, cualquier política de escritura con éxito (write-back o write-through) se puede combinar con cualquier política de escritura con falla (write allocate o no write allocate).
- En la práctica, lo más común es usar una de las siguientes dos combinaciones:
- Write-through con no write allocate.
- Write-back con write allocate.



# Write-through con no write allocate

- Si hay éxito, escribe en el caché y en la memoria principal.
- Si hay falla, actualiza el bloque en la memoria sin cargar el bloque en el caché.
- Las siguientes escrituras al mismo bloque actualizan el bloque en la memoria.
- Se ahorra tiempo al no cargar el bloque en el caché al producirse una falla.



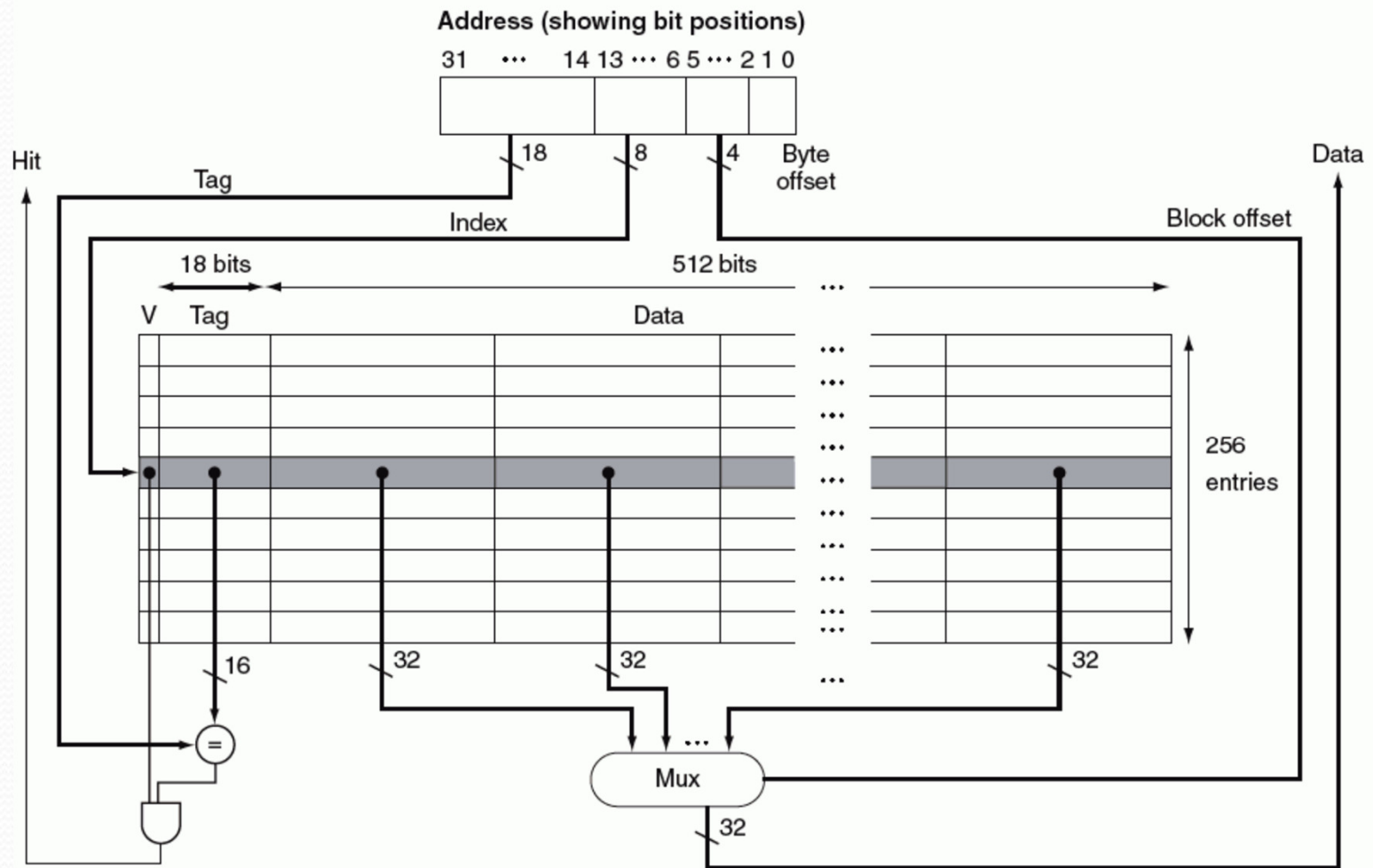
# Write-back con write allocate

- Si hay éxito, escribe en el caché y prende el bit sucio del bloque. La memoria no se actualiza.
- Si hay falla, actualiza el bloque en el memoria y lo carga en el caché.
- Las siguientes escrituras al mismo bloque van a producir éxitos y se ahorran accesos a la memoria.

# FastMATH

- Intrinsity FastMATH es un procesador con un pipeline de 12 etapas y arquitectura MIPS.
- FastMATH tiene un caché dividido (split cache): dos cachés independientes que operan en paralelo, uno para instrucciones y otro para datos.
- La capacidad del caché es de 32 KB en total (16 KB por cada caché).
- Cada caché tiene 256 bloques con 16 palabras (64 bytes) por bloque.

# FastMATH





# FastMATH (Explicación)

- La palabra correcta se selecciona con un MUX controlado por los bits 2 a 5 de la dirección.
- Para escribir, FastMATH ofrece write-through y write-back, dejando al sistema operativo decidir que estrategia usar para una aplicación.
- Además, tiene un buffer de escritura de una entrada.

# FastMATH

- Tasas de falla en SPEC2000.

| Instruction miss rate | Data miss rate | Effective combined miss rate |
|-----------------------|----------------|------------------------------|
| 0.4%                  | 11.4%          | 3.2%                         |

- Por lo general, un caché unificado tiene tasa de falla ligeramente menor que un caché dividido: 3.18% vs. 3.24%.
- Un caché dividido tiene más ancho de banda porque puede acceder instrucciones y datos a la vez.



# Organización de la memoria

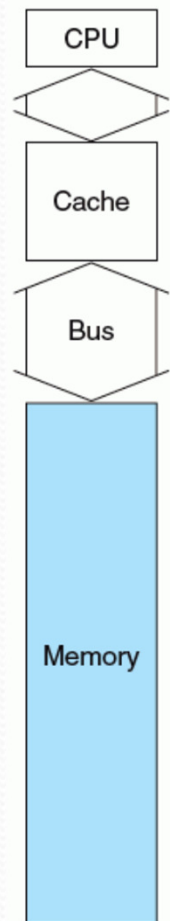
- La memoria se puede organizar para reducir el castigo por falla.
- Hay 3 formas de organizar la memoria:
  1. **Memoria delgada.** El tamaño de bloque del caché, el ancho del bus que conecta el caché con la memoria y el ancho de la memoria es típicamente de 1 palabra.
  2. **Memoria ancha.** El tamaño de bloque del caché, el ancho del bus que conecta el caché con la memoria y el ancho de la memoria es típicamente de 4 palabras o más.



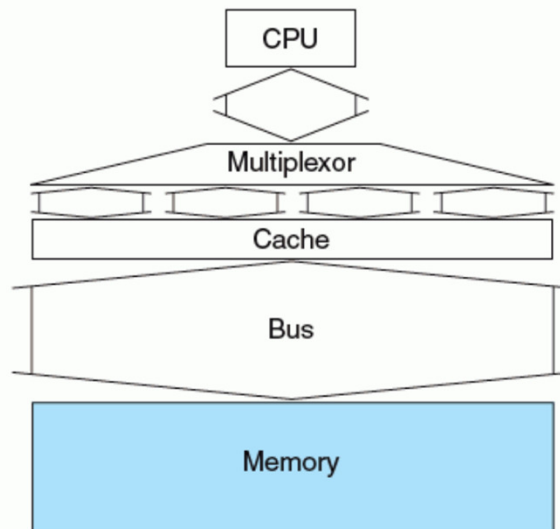
# Organización de la memoria

3. **Memoria entrelazada (interleaved).** El tamaño de bloque del caché y el ancho del bus que conecta el caché con la memoria es típicamente de 1 palabra. La memoria está organizada en bancos que se pueden leer en paralelo.

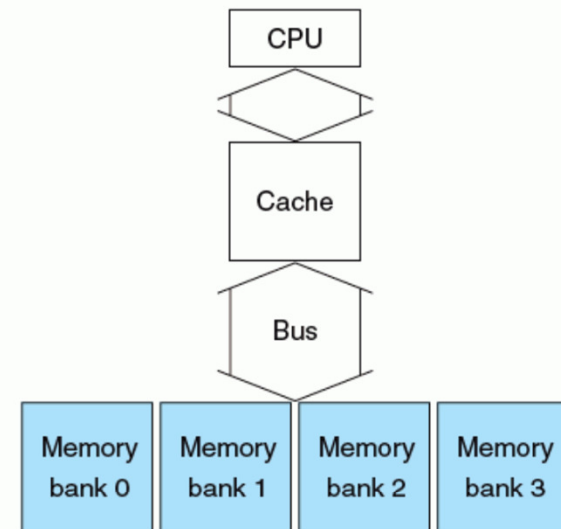
# Organización de la memoria



a. One-word-wide memory organization



b. Wide memory organization



c. Interleaved memory organization

# Ejemplo

- Suponer los siguientes tiempos en ciclos de reloj:
  - 1 ciclo para enviar la dirección.
  - 15 ciclos por cada acceso a memoria.
  - 1 ciclo para enviar una palabra.
- Suponer un tamaño de bloque de 4 palabras.



# Memoria delgada (1 palabra)

- Ancho del bus y de la memoria es de 1 palabra.
- El acceso a la memoria es secuencial.
- Castigo por falla:  $1 + 4 \times 15 + 4 \times 1 = 65$  ciclos de reloj del bus.
- Número de bytes transferidos por ciclo de reloj por cada falla:  $(4 \times 4) / 65 = 0.25$

# Memoria ancha (4 palabras)

- Ancho del bus y de la memoria es de 4 palabras.
- El acceso a la memoria es en paralelo.
- Castigo por falla:  $1 + 1 \times 15 + 1 \times 1 = 17$  ciclos de reloj del bus.
- Número de bytes transferidos por ciclo de reloj por cada falla:  $(4 \times 4) / 17 = 0.94$



# Memoria entrelazada

- Ancho del bus es de 1 palabra.
- Ancho de la memoria es de 4 palabras repartidas en 4 bancos de memoria independientes.
- El acceso a la memoria es en paralelo.
- La transferencia es secuencial.
- Castigo por falla:  $1 + 1 \times 15 + 1 \times 4 = 20$  ciclos de reloj.
- Número de bytes transferidos por ciclo de reloj del bus por cada falla:  $(4 \times 4) / 20 = 0.8$



# Organización de la memoria

- La organización entrelazada es la preferida:
  - El castigo por falla es aceptable.
  - Un bus y memoria de 4 palabras es mas caro que un bus y 4 bancos de 1 palabra cada uno.
  - Cada banco puede escribir en forma independiente, cuadruplicando el ancho de banda al escribir y provocando menos detenciones (stalls) en un caché write-through.

# Organización del caché

- Hasta ahora solo se ha visto la estrategia de mapeo directo.
- Un bloque solo puede ir en un lugar en el caché.
- Otras organizaciones pueden reducir la tasa de fallas:
  - Fully associative cache. El bloque de memoria puede ir en cualquier línea del caché.
  - Set associative cache. El bloque de memoria puede ir en cualquier línea dentro del conjunto que le toque.



# Fully associative cache

- Un bloque en memoria puede estar asociado con cualquier línea en el caché.
- Para encontrar un item se debe buscar en todo el caché.
- Para ser práctica, la búsqueda se hace en paralelo asociando un comparador con cada línea del caché.
- Los comparadores incrementan el costo del hardware.
- Son prácticos solo para cachés pequeños.



# Set associative cache

- El caché está dividido en  $m$  conjuntos.
- Cada conjunto consta de  $n$  bloques.
- Se le llama  $n$ -way set associative.
- Un bloque en la memoria está asociado con un solo conjunto.
- Dentro del conjunto, el bloque puede ir en cualquier bloque.
- Un item se busca en todos los bloques del conjunto asociado con este item.

# Set associative cache

- Las fórmulas para calcular el bloque en el caché son *casi* las mismas que en mapeo directo.
- La diferencia es que se usa el número de conjuntos y no el número de bloques.



# Set associative cache

- $d = a \text{ div } k$
- $d$  es la dirección de bloque
- $a$  es la dirección del item en la memoria
- $k$  es el tamaño del bloque en bytes
- $b = d \text{ mod } n$
- $b$  es el número de bloque
- $n$  es el número de **conjuntos** que tiene el caché



# Set associative cache

- La división de la dirección en binario funciona *casi* igual que en mapeo directo:
- Offset. La posición del byte dentro del bloque. Ocupa los bits bajos de la dirección. Tamaño =  $\log_2(k)$ ,  $k$  es el tamaño del bloque en bytes.
- Índice. El número de conjunto en donde se guarda el dato. No hay forma de saber en que bloque se guarda el dato. Ocupa los bits intermedios de la dirección. Tamaño =  $\log_2(n)$ ,  $n$  es el número de conjuntos que tiene el caché.

# Set associative caché

- Etiqueta. Ocupa los bits altos de la dirección.



# Ejemplo

- Suponer un caché 4-way con capacidad de 4K bytes y bloques de 4 palabras. Las direcciones son de 32 bits.
- ¿Cuántos bloques y cuántos conjuntos tiene el caché?
- ¿Cómo se divide la dirección?
- ¿Qué conjunto en el caché le toca a la dirección en la memoria 1714?



# Ejemplo

- ¿Cuántos bloques y cuántos conjuntos tiene el caché?
- Número de bloques:

$$\text{bloques} = \frac{4K}{4W} = \frac{4 \times 1024}{4 \times 4} = 256 \text{ bloques}$$

- Es un cache 4-way, por lo tanto cada conjunto tiene 4 bloques, es decir:

$$\text{conjuntos} = \frac{256}{4} = 64 \text{ conjuntos}$$

# Ejemplo

- ¿Cómo se divide la dirección?
- Tamaño del offset =  $\log_2(16) = 4$  bits
- Tamaño del índice =  $\log_2(64) = 6$  bits
- Tamaño de la etiqueta =  $32 - (6 + 4) = 22$  bits

# Ejemplo

- ¿Qué conjunto en el caché le toca a la dirección en la memoria 1714?
- Método 1 – usando las fórmulas.
- Datos:  $a = 1714$ ;  $k = 16$ ;  $n = 64$
- $d = 1714 \text{ div } 16 = 107$
- $b = 107 \text{ mod } 64 = 43$
- Conclusión: la dirección 1714 se guarda en el conjunto 43.



# Ejemplo

- ¿Qué conjunto en el caché le toca a la dirección en la memoria 1714?
- Método 2 – analizando la dirección en binario.
- $1714_{10} = 11010110010_2 = 1-101011-0010$
- Se convierte el índice a base 10.
- $101011_2 = 43_{10}$
- Conclusión: la dirección 1714 se guarda en el conjunto 43.

# Variantes de una forma

- Mapeo directo y fully associative se pueden ver como variantes de set associative.
- 1. Mapeo directo es 1-way set associative.
  - Un caché de  $n$  bloques se puede ver como un caché de  $n$  conjuntos y cada conjunto tiene 1 bloque.
- 2. Fully associative es  $n$ -way set associative.
  - Un caché de  $n$  bloques se puede ver como un caché de 1 conjunto de  $n$  bloques.

# Variantes de una forma

- Variantes para un caché de 8 bloques.

One-way set associative  
(direct mapped)

| Block | Tag | Data |
|-------|-----|------|
| 0     |     |      |
| 1     |     |      |
| 2     |     |      |
| 3     |     |      |
| 4     |     |      |
| 5     |     |      |
| 6     |     |      |
| 7     |     |      |

Two-way set associative

| Set | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|
| 0   |     |      |     |      |
| 1   |     |      |     |      |
| 2   |     |      |     |      |
| 3   |     |      |     |      |

Four-way set associative

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0   |     |      |     |      |     |      |     |      |
| 1   |     |      |     |      |     |      |     |      |

Eight-way set associative (fully associative)

| Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|
|     |      |     |      |     |      |     |      |     |      |     |      |     |      |     |      |

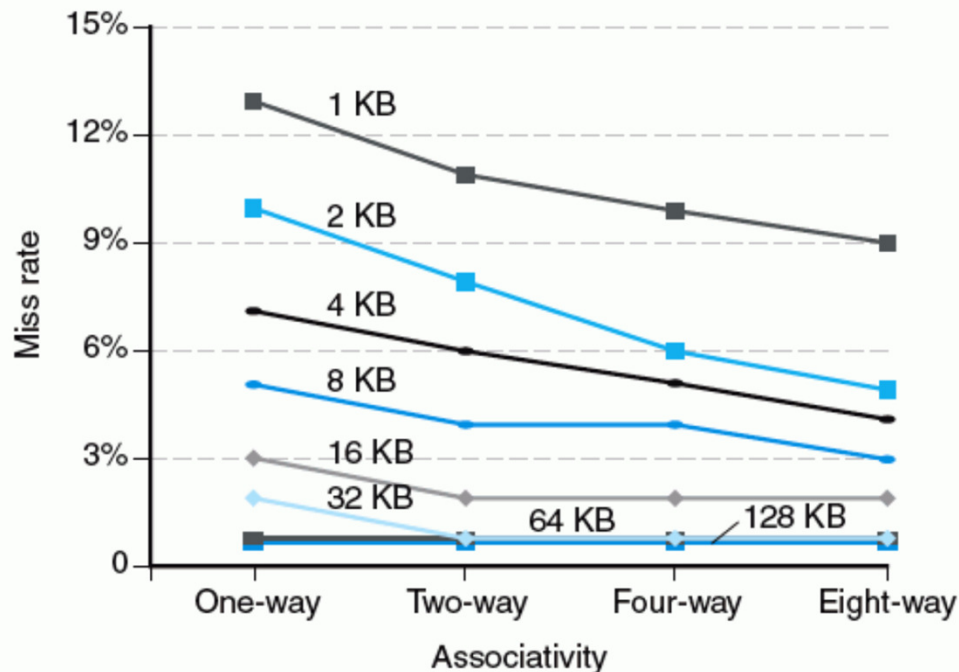


# Políticas de reemplazo

- Se necesita una política de reemplazo para cuando un conjunto está lleno y tiene que insertarse un nuevo bloque.
- Políticas más comunes:
- LRU (least recently used) o una aproximación.
- FIFO (first-in, first-out).
- Aleatorio.

# Efecto de la asociatividad

- Ventaja: generalmente decrecienta la tasa de fallas.
- Desventaja: incrementa el tiempo de éxito (hit time).



# Efecto de la asociatividad

- Ejemplo: caché de datos de 64 KB con bloques de 64 bytes.
- La asociatividad se varía desde 1-way (mapeo directo) hasta 8-way.
- Benchmark SPEC2000.

| Associativity | Data miss rate |
|---------------|----------------|
| 1             | 10.3%          |
| 2             | 8.6%           |
| 4             | 8.3%           |
| 8             | 8.1%           |



# Comparación

| Characteristic         | ARM Cortex-A8                        | Intel Nehalem                              |
|------------------------|--------------------------------------|--------------------------------------------|
| L1 cache organization  | Split instruction and data caches    | Split instruction and data caches          |
| L1 cache size          | 32 KiB each for instructions/data    | 32 KiB each for instructions/data per core |
| L1 cache associativity | 4-way (I), 4-way (D) set associative | 4-way (I), 8-way (D) set associative       |
| L1 replacement         | Random                               | Approximated LRU                           |
| L1 block size          | 64 bytes                             | 64 bytes                                   |
| L1 write policy        | Write-back, Write-allocate(?)        | Write-back, No-write-allocate              |
| L1 hit time (load-use) | 1 clock cycle                        | 4 clock cycles, pipelined                  |
| L2 cache organization  | Unified (instruction and data)       | Unified (instruction and data) per core    |
| L2 cache size          | 128 KiB to 1 MiB                     | 256 KiB (0.25 MiB)                         |
| L2 cache associativity | 8-way set associative                | 8-way set associative                      |
| L2 replacement         | Random(?)                            | Approximated LRU                           |
| L2 block size          | 64 bytes                             | 64 bytes                                   |
| L2 write policy        | Write-back, Write-allocate (?)       | Write-back, Write-allocate                 |
| L2 hit time            | 11 clock cycles                      | 10 clock cycles                            |
| L3 cache organization  | -                                    | Unified (instruction and data)             |
| L3 cache size          | -                                    | 8 MiB, shared                              |
| L3 cache associativity | -                                    | 16-way set associative                     |
| L3 replacement         | -                                    | Approximated LRU                           |
| L3 block size          | -                                    | 64 bytes                                   |
| L3 write policy        | -                                    | Write-back, Write-allocate                 |
| L3 hit time            | -                                    | 35 clock cycles                            |

# Resumen

- Caché de mapeo directo: un item está asociado con un solo bloque.
- Caché n-way: un item está asociado con un solo conjunto. Dentro de este conjunto, puede estar en cualquier bloque.
- Caché fully: un item está asociado con cualquier bloque.
- Políticas de escritura con éxito: write-through y write-back.

# Resumen

- Políticas de escritura con falla: write allocate y no write allocate.
- Para tomar ventaja de la locality espacial el bloque del caché debe ser mayor a un byte.
- Bloques grandes reducen la tasa de fallas y mejora la eficiencia al requerir menos espacio para las etiquetas.
- Bloques grandes aumentan el castigo por falla.



# Resumen

- El castigo por falla se puede reducir incrementando el ancho de banda de la memoria para transferir bloques de forma eficiente.
- Dos métodos comunes para aumentar el ancho de banda son memoria ancha y memoria interleaved.
- Interleaving tiene ventajas adicionales.

# Resumen

- Los cachés n-way set associative ( $n > 1$ ) por lo general tienen tasas de fallas menores que los cachés de mapeo directo.
- Los cachés n-way set associative ( $n > 1$ ) tienen mayor tiempo de éxito que los cachés de mapeo directo.
- Se puede reducir el tiempo de éxito usando  $n$  comparadores.