Tema 1 – Fundamentos de diseño y evaluación de computadores

Fallos

$\lambda = 1/MTTF$ MTTF (Mean Time To Failure) λ (Tasa de fallos)

$MTBF = MTTF + MTTR$ MTTR (Mean Time To Repair)

$Availability = MTTF/MTBF$ MTBF (Mean Time Between Failures)

Tiempo de ciclo

$$t_{ciclo} = \frac{1}{f_{clock}}$$

Tiempo de ejecución

$$t_{ejecución} = n.instrucciones * CPI * t_{ciclo}$$

$$CPI = CPI_1 * I_1 + CPI_2 * I_2 + \cdots + CPI_n + I_n = \boldsymbol{x}\ \frac{ciclos}{instr.} \qquad IPC = \frac{1}{CPI}$$

Ganancia

$$ganancia\ (speedup) = \frac{T_A}{T_B} = B\ es\ \boldsymbol{x}\ número\ de\ veces\ más\ rápido\ que\ A$$

$$\rightarrow \left(\frac{T_A}{T_B} - 1\right) * 100 =\ B\ es\ un\ \boldsymbol{x}\ \%\ más\ rápido\ que\ A$$

Ley de Amdahl

$$t_{ejecución} = t_{fase\ 1} + t_{fase\ 2} + \cdots + t_{fase\ n} \qquad Speed\text{-}up = \frac{1}{1 - fm + \frac{fm}{gm}}$$

Fm = fase mejorada gm = ganancia obtenida

MIPS (Millones de Instrucciones Por Segundo)

$$MIPS = \frac{n.\ instr.}{t_{ej}.* 10^6} = \frac{f_{clock}}{CPI * 10^6} = \frac{1}{CPI * t_{ciclo} * 10^6}$$

MFLOPS (Millones de Operaciones en Punto Flotante por Segundo)

$$MFLOPS = \frac{\#op.\ punto\ flotante}{t_{ej} * 10^6}$$

Evaluación de coste

$$Coste\ circuito\ integrado = \frac{Coste\ die + Coste\ testeo + Coste\ empaquetado\ y\ test\ final}{Yield\ final\ (test)}$$

$$Coste\ die = \frac{Coste\ waffer}{Dies\ por\ waffer * Die\ yield}$$

$$Dados\ por\ oblea\ /\ Dies\ por\ waffer = \frac{Area\ útil}{Die\ area} = \frac{\pi * \left(\frac{diametro}{2}\right)^2}{Die\ area} = \frac{\pi * diametro}{\sqrt{2 * Die\ area}}$$

$$Die\ yield = waffer\ yield * \left(1 + \frac{defectos\ por\ unidad\ de\ area * die\ area}{\alpha}\right)^{-\alpha}$$

$\alpha =$ medida de la complejidad, se aproxima al número de máscaras críticas

Consumo

$$P = I * V = amperios * voltios = watios$$

$$E = P * t = watios * segundos = julios$$

P = Potencia
I = Intensidad o Corriente
V = Tensión
E = Energia
t = Tiempo
C = Capacidad

Potencia y energía de conmutación/dinámica

$$P = C * V^2 * f = faradios * voltios^2 * hercios$$

$$E = C * V^2 = faradios * voltios^2$$

Potencia de fugas/estática

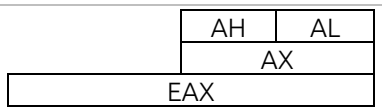
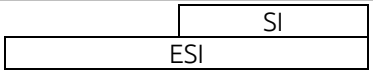
$$P = I_{de\ fuga} * V$$

Métricas de eficiencia

$$Eficiencia\ enerégtica = \frac{rendimiento}{watio} = \frac{1}{tiempo * watio} = \frac{1}{Energía\ consumida}$$

Tema 2 – Interfaz Alto Nivel - Ensamblador

Registros

32 bits	16 bits	8 bits	
%eax	%ax	%ah, %al	
%ebx	%bx	%bh, %bl	
%ecx	%cx	%ch, %cl	
%edx	%dx	%dh, %dl	
%esi	%si		
%edi	%di		
%esp	%sp		Reservados para uso específico
%ebp	%bp		
%eip			Contador programa
%eflags			Palabra de estado

Ejemplos de modos de direccionamiento

(%eax,%ebx)	M[eax+ebx]
-3(%eax,%ebx)	M[eax+ebx-3]
(%eax,%ebx,4)	M[eax+ebx·4]
(,%ebx,4)	M[ebx·4]
12(%eax)	M[eax+12]
(%eax)	M[eax]
3(%eax,%esi,2)	M[eax+esi·2+3]
4	M[4]
$4	4
%eax	Registro eax
%al	8 bits de menor peso de eax

Sólo valores: 1, 2, 4 y 8

Instrucciones de movimientos de datos

Instrucciones	Descripción	Notas	Ejemplo
MOVx op1, op2	op2 ← op1	x = {L, W, B}	MOVB $-1,%AL
MOVSxy op1, op2	op2 ← ExtSign(op1)	xy = {BW, BL, WL}	MOVSBW %CH,%AX

MOVZxy op1, op2	op2 ← ExtZero(op1)	xy = {BW, BL, WL}	MOVZWL %BX,%EDX
PUSHL op1	%ESP ← %ESP – 4; M[%ESP] ← op1		PUSHL 12(%EBP)
POPL op1	op1 ← M[%ESP]; %ESP ← %ESP + 4;		POPL %EAX
LEAL op1, op2	op2 ← &op1	op1: memoria	LEAL (%EBX,%ECX),%EAX

Instrucciones aritméticas

Instrucciones	Descripción	Notas	Ejemplo
ADDx op1, op2	op2 ← op2+op1	x = {L, W, B}	ADDL $13,%EAX
SUBx op1, op2	op2 ← op2-op1	x = {L, W, B}	SUBW %CX,%AX
ADCx op1, op2	op2 ← op2+op1+CF	x = {L, W, B}	ADCL %EDX,%EAX
SBBx op1, op2	op2 ← op2-op1-CF	x = {L, W, B}	SBBL %ECX,%EAX
INCx op1	op1 ← op1+1	x = {L, W, B}	INCL %EAX
DECx op1	op1 ← op1-1	x = {L, W, B}	DECW %BX
NEGx op1	op1 ← -op1	x = {L, W, B}	NEGL %EAX
IMUL op1, op2	op2 ← op2·op1	op2: registro	IMUL (%EBX),%EAX
IMUL inm,op1,op2	op2 ← op1·inm	inm: constante	IMUL $3,%EAX,%ECX
IMULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (Enteros)	IMULL (%EBX)
MULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (Naturales)	MULL (%EBX)
CLTD	%EDX%EAX ← ExtSign(%EAX)	CLTD	
IDIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (Enteros)	IDIVL (%EBX)
DIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (Naturales)	DIVL %ESI

Instrucciones lógicas

Instrucciones	Descripción	Notas	Ejemplo
ANDx op1, op2	op2 ← op2&op1	x = {L, W, B}	ANDL $13,%EAX
ORx op1, op2	op2 ← op2|op1	x = {L, W, B}	ORW %CX,%AX
XORx op1, op2	op2 ← op2^op1	x = {L, W, B}	XORL %EDX,%EAX
NOTx op1	op1 ← ~op1	x = {L, W, B}	NOTB %AH
SALx k,op1	op1 ← op1<<k (aritm.)	x = {L, W, B}, k: inm. o %CL	SALL $1,%EAX
SHLx k,op1	op1 ← op1<<k (log.)	x = {L, W, B}, k: inm. o %CL	SHLW %CL,%DX
SARx k,op1	op1 ← op1>>k (aritm.)	x = {L, W, B}, k: inm. o %CL	SARL $1,%EAX
SHRx k,op1	op1 ← op1>>k (log.)	x = {L, W, B}, k: inm. o %CL	SHRW %CL,%DX
CMPx op1, op2	op2-op1	x = {L, W, B}, activa flags	CMPL $13,%EAX
TESTx op1, op2	op2&op1	x = {L, W, B}, activa flags	TESTW %CX,%AX

Instrucciones de secuenciamiento

Instrucciones	Descripción	Notas	Ejemplo
JMP etiq	EIP ← EIP+despl.	EIP ← &etiq	JMP loop
JMP op	EIP ← op	op: reg. o mem.	JMP (%ebx,%esi,4)

Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {E, NE, G, GE, L, LE, …} (**Z**)	JLE else
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {A, AE, B, BE, …} (**N**)	JA loop
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {Z, NZ, C, NC, O, …} (flags)	JNC error
CALL etiq	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← EIP+despl.	Guardar @retorno EIP ← &etiq	CALL sub
CALL op	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← op	op: reg. o mem.	CALL (%EBX)
RET	EIP ← M[%ESP]; %ESP ← %ESP+4	RET	

Instrucciones	Flags	Descripción
JE etiq	ZF	Igual / cero
JNE etiq	~ZF	No igual / no cero
JS etiq	SF	Negativo
JNS etiq	~SF	No negativo
JG etiq	~(SF^OF)&~ZF	Mayor (con signo)
JGE etiq	~(SF^OF)	Mayor o igual (con signo)
JL etiq	(SF^OF)	Menor (con signo)
JLE etic	(SF^OF)|ZF	Menor o igual (con signo)
JA etiq	~CF&~ZF	Mayor (sin signo)
JAE etiq	~CF	Mayor o igual (sin signo)
JB etiq	CF	Menor (sin signo)
JBE etiq	CF^ZF	Menor o igual (sin signo)

Vectores

$$v[i] \rightarrow @inicio\_v + i * tamaño$$

```
int Vi(int V[100], int i) {
   return V[i];
}
```

```
pushl %ebp
movl %esp, %ebp
movl 8(%ebp),%ecx          # @V -> 8[%ebp]
movl 12(%ebp),%edx         # i -> 12[%ebp]
movl (%ecx,%edx,4),%eax
popl %ebp               # resultado en %eax
ret
```

Matrices

$$M[i][j] \rightarrow @inicio_M + (i * num_{columnas} + j) * tamaño$$

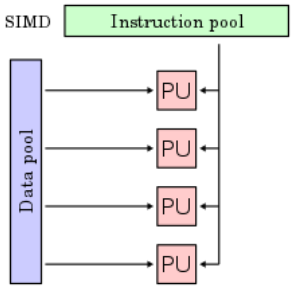
```
int Mfc(int M[50][80], int fil, int
col) {
   return M[fil][col];
}
```

```
pushl %ebp
movl %esp, %ebp
imull $80,12(%ebp),%eax    # fil → 12[%ebp]
addl 16(%ebp),%eax         # col -> 16[%ebp]
movl 8(%ebp),%ecx          # @M -> 8[%ebp]
movl (%ecx,%eax,4),%eax
popl %ebp               # resultado en %eax
ret
```

Optimizaciones:

- Desenrollar: hacer varias lecturas por bucle.
- Utilizar instrucciones SIMD.
- Utilizar acceso secuencial en vez de acceso aleatorio.

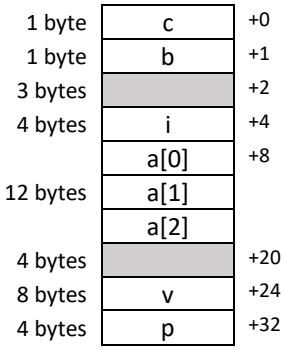
Instrucciones SIMD



Es una técnica usada para conseguir paralelismo a nivel de datos, son una serie de instrucciones que aplican la misma operación sobre un conjunto de datos (más o menos grandes), de esta manera ejecutamos por cada unidad de datos, la instrucción deseada, pero a nivel de conjunto, lo que hace que el tiempo de ejecución de un programa sea menor, dado que se reducen el número de instrucciones a ejecutar.

Tipos de datos estructurados

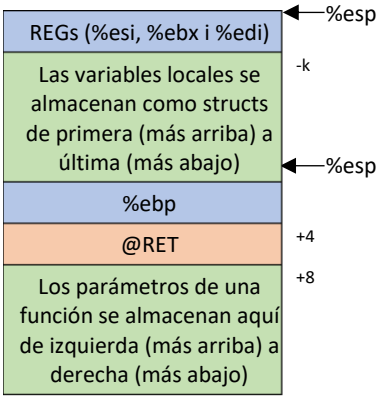
```c
// Un tipo de dato ocupa k bytes, por lo que la
dirección debe ser múltiplo de k
typedef struct {
   char c;     // ocupa 1 byte
   char b;     // ocupa 1 byte
   int i;      // ocupa 4 bytes
   int a[3];      // ocupa 3 * 4 = 12 bytes
   double v;   // ocupa 8 bytes
   int *p;     // ocupa 4 bytes
} S;
```



Gestión de subrutinas

- Los registros:
  - %ebp, %esp se salvan **siempre** implícitamente en la gestión de subrutinas.
  - %ebx, %esi, %edi se han de **salvar si son modificados**.
  - %eax, %ecx, %edx se pueden modificar en el interior de la subrutina porque se han de **salvar** en el **llamador**.
- Los **resultados** se devuelven siempre en **%eax**.
- La pila siempre debe quedar alineada a 4.

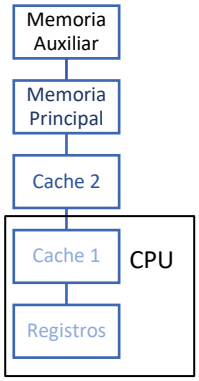
Bloque de activación o pila




```
pushl %ebp
movl %esp, %ebp
# X es el valor que ocupan las variables locales
subl $X, %esp

# para salvar los registros
pushl %ebx
pushl %esi
pushl %edi
# el push de los parametros para llamar a una
función se hace de derecha a izquierda

addl $Y, %esp
# y -> valor de los parámetros

# devolvemos el valor a los registros
popl %edi
popl %esi
popl %ebx

movl %ebp, %esp
popl %ebp
ret
```

Tema 3 – Jerarquía de Memoria

Memoria Cache



Memorias estáticas (SRAM, Static RAM): son rápidas, pequeñas (poca capacidad), alto consumo (1 biestable, 6-8 transistores) y caras.

→ Se utilizan para **Memoria Cache**

Memorias dinámicas (DRAM, Dynamic RAM): son lentas, bajo consumo (1 transistor, que se comporta como un condensador), grandes (mucha capacidad) y baratas.

→ Se utilizan para **Memoria Principal**

Propiedades de los programas:

Localidad Temporal: Si accedemos a una posición de memoria, es muy probable que se vuelva a acceder a la misma posición en un futuro cercano.

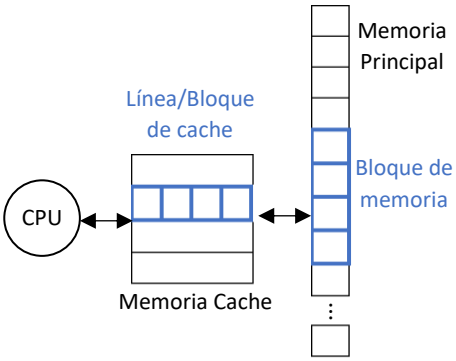
- Las instrucciones, dentro de un bucle, se acceden repetidamente a las mismas instrucciones.
- Los datos, dentro de un bucle, se acceden repetidamente a las mismas variables.

→ Traemos un dato o instrucción de memoria "cerca" del procesador para que los futuros accesos sean más rápidos.

Localidad Espacial: Si accedemos a una posición de memoria, es muy probable que se acceda a posiciones próximas en un futuro.

- Las instrucciones se ejecutan en secuencia.
- Los datos, dentro de un bucle, se suelen recorrer de manera secuencial.

→ Es útil traer, además de la misma instrucción o dato, los que son próximos a este, además el costo de traer los datos o instrucciones próximas sólo es un poco mayor.



Conceptos

- Acierto/Hit: cuando la posición de memoria que buscamos está en la cache.
- Fallo:/Miss cuando la posición de memoria que buscamos no está en la cache.

Algoritmos de emplazamiento: determina en que línea de la MC se coloca o dónde buscar un bloque de la MP.

- **Emplazamiento directo**: dónde cada bloque de la MP va directamente a una línea de la MC, dependiendo de los bits de menos peso. Es el algoritmo con menos tiempo de acceso pero más tasa de fallos. Ejemplo de una cache de 4 líneas :

MC

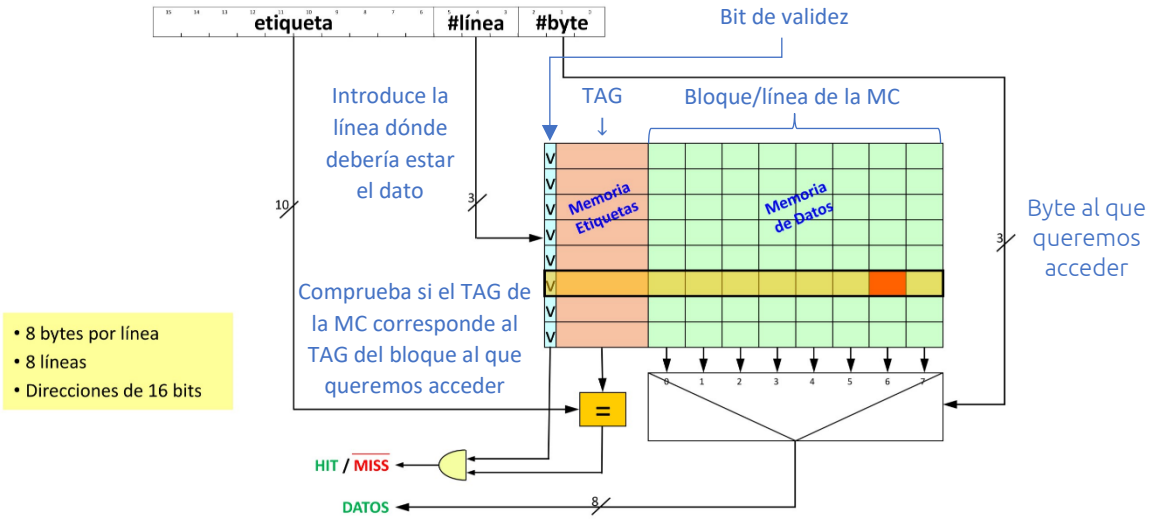
línea	Posición por bloques
00	A, E
01	B, F
10	C, G
11	D, H

MP

000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

TAG	#Línea MC	#bytes
Identificador de bloque, indica que bloque hay en la línea de la MC de los posibles que puede haber.	Indica la posición en la línea de cache que le corresponde a un bloque.	Identificador para el byte dentro de la línea de MC

Un **bit de validez** estará a 0 (no válido) cuándo al encender el ordenador una línea se llena de 0s y 1s pero que no hacen referencia a nada, se pondrá a 1 (válido) al cargar un dato por primera vez.



- **Emplazamiento asociativos por conjuntos**: la cache se divide en un número de conjuntos y dependiendo de los bits de menos peso, un bloque de la MP es asignado un conjunto. Ejemplo de una cache de 4 líneas:

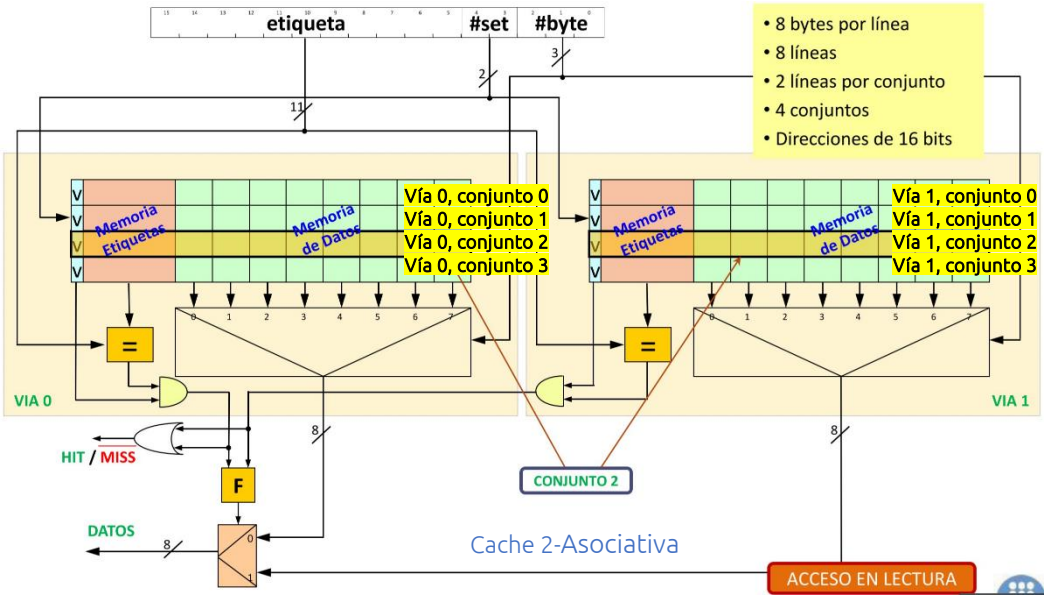
MC

conjunto	Posición bloques
0	A, C, E, G
0	A, C, E, G
1	B, D, F, H
1	B, D, F, H

MP

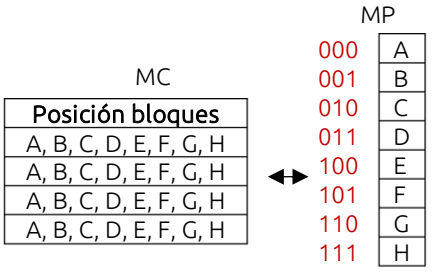
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

TAG	#Conj. MC	#bytes
Indica que bloque hay en una línea de cache de los posibles que puede haber.	Indica el conjunto en la cache que le corresponde a un bloque.	Identificador para el byte dentro de la línea de MC.



Se accede al conjunto, es decir a todas las vías del mismo conjunto a la vez y se comprueba si la línea del TAG de la MC corresponde con el TAG de la posición a la que queremos acceder, en el caso de que alguna de las líneas del conjunto corresponda (HIT) se devuelve el dato deseado, en caso contrario (MISS).

- **Emplazamiento completamente asociativo**: un bloque de la MP puede ir a cualquier línea de la MC. Es el algoritmo con más tiempo de acceso pero menos tasa de fallos. Ejemplo de una cache de 4 líneas:

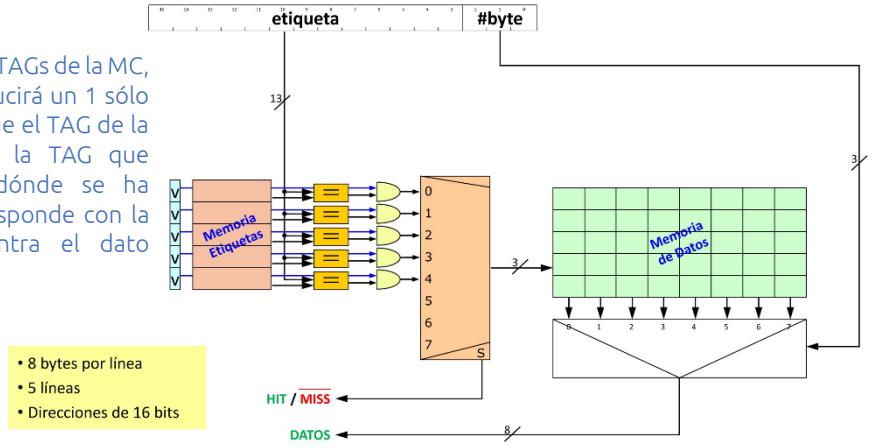


TAG	#bytes

Dado que puede ir en cualquier posición, el identificador es toda la dirección de la MP.

Identificador para el byte dentro de la línea de MC ($2^{\#bytes}$)

Se consultan todos los TAGs de la MC, de forma que se producirá un 1 sólo hit en aquella salida que el TAG de la MC corresponda con la TAG que deseamos, la línea dónde se ha producido el hit corresponde con la línea que se encuentra el dato deseado.



Algoritmos de remplazo: determina que línea se ha de eliminar de la MC para dejar espacio para un nuevo bloque de la MP.

- **Aleatorio:** se escoge aleatoriamente una línea de entre todas las candidatas para ser remplazada.
- **FIFO** (First In First Out): se selecciona la línea que lleva más tiempo en la cache para ser remplazada.
- **LRU** (Least Recently Used): se selecciona la línea que más tiempo sin ser usada en la cache para ser remplazada.
- Un algoritmo LRU, cuando el número de líneas de conjunto crece mucho el algoritmo se hace difícil de implementar y muy lento, por lo que para simplificar se hace un **pseudoLRU**, dónde el número de estados se simplifica, es decir, no tiene en cuenta todas las posibilidades.

La elección del algoritmo de remplazo depende del programa en si y el tamaño de MC usado.

Políticas de escritura: determina en qué momento se escriben en MP.

- ¿Cuándo se actualiza la MP?
  - WRITE THROUGH (escritura inmediata): el dato se actualiza a la vez en MP y en MC.
  - COPY BACK (escritura diferida): el dato se actualiza sólo en la MC (Se añade un bit, llamado dirty bit, por cada línea de la MC para indicar si ha sido modificada, si ha sido modificado al producirse un remplazo hay que escribir todo el bloque de la MC a la MP).
- ¿Qué hacer en caso de fallo en escritura? – Política de escritura en caso de fallo
  - WRITE ALLOCATE (con migración en caso de fallo): se trae el bloque de MP a MC y después se realiza la escritura.
  - WRITE NO ALLOCATE (sin migración en caso de fallo): el bloque **no** se trae a la MC, esto obliga a realizar la escritura directamente en MP.

Las mejores políticas de escritura como norma general:

- Write Trough + Write NO Allocate
  - Lectura $tasa_{lectura}$
    - Hit $t_{lectura\ hit} = tasa_{aciertos} * Tsa$ (Tsa = tiempo de servicio en caso de acierto en MC)
    - Miss $t_{lectura\ miss} = tasa_{fallos} * (Tsa * 2 + t_{lectura\ bloque\ MP})$
  - Escritura $t_{escritura} = tasa_{escritura} * t_{escritura\ palabra\ MP}$

$$t_{inveritdo} = \#accesos * [tasa_{lectura} * (t_{lectura\ hit} + t_{lectura\ miss}) + t_{escritura}]$$

- Copy Back + Write Allocate
  - Acierto $t_{lectura} = tasa_{aciertos} * Tsa$
  - Fallo $tasa_{fallos}$
    - Bloque modificado $t_{bloq.\ mod.} = tasa_{bloq.\ mod.} * (2 * Tsa + t_{esc.\ bloq.} + t_{lec.\ bloq.})$
    - Bloque no modificado $t_{bloq.\ no\ mod.} = tasa_{bloq.\ no\ mod.} * (2 * Tsa + t_{lec.\ bloq.})$

$$t_{invertido} = \#accesos * [t_{lectura} + tasa_{fallos} * (t_{bloq.\ mod.} + t_{bloq.\ no\ mod})]$$

Evaluación

$$(tasa\ aciertos)\ \mathrm{h} = \frac{\#aciertos}{\#accesos} \qquad (tasa\ fallos)\ \mathrm{m} = \frac{\#fallos}{\#accesos} = 1 - h$$

$$Tiempo\ medio\ acceso\ (T_{ma}) = h * tsa + m * tsf = tsa + m * tpf = \frac{\#ciclos}{\#accesos}$$

Probabilidad de acceso en un ciclo $P(acceso\ ciclo) = 1/T_{ma}$

tsa = Tiempo de servicio en caso de acierto en MC

Coste de un acceso en fallo $(tsf) = tsa + tpf$

tpf = Tiempo de penalización en caso de fallo

$$T_{ejec} = N * CPI * Tc$$

$$CPI = CPI_{Ideal} + CPI_{mem} \qquad IPC = CPI^{-1} = \frac{1}{CPI}$$

$$CPI_{ideal} = \frac{\#ciclos}{\#instr}$$

$$CPI_{mem} = [nr * (T_{ma} - tsa)] = nr * [tpf * m + tpa * (1 - m)]$$

$nr = \frac{\#accesos}{\#instr}$ (nr) número medio de referencias por instrucción

Cache Directa → Cuánto más memoria menos tasa de fallos.

→ Bloques grandes van mal para caches pequeñas y bien para caches grandes.

→ Bloques pequeños van bien para caches pequeñas y mal para caches grandes.

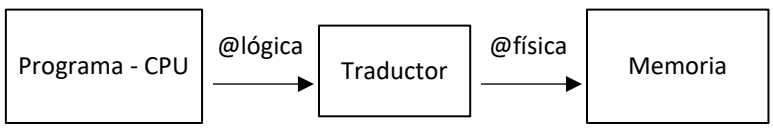
Cache Asociativa → Cuánto más grande menos fallos, pero más tiempo de tsa.

→ Cuánto más asociativa menos fallos, pero más tiempo de tsa.

Los procesadores integran una MC de instrucciones y una MC de datos por separado

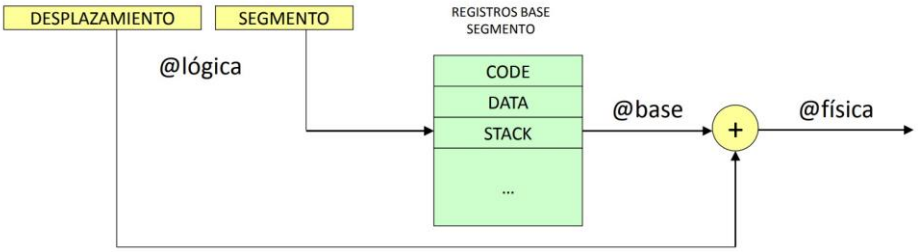
Memoria Virtual

Traducción de direcciones



Esquemas de traducción:

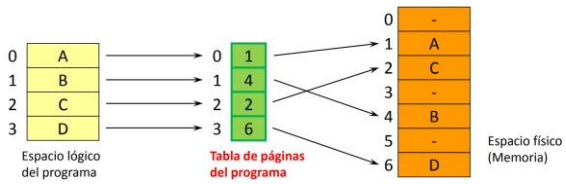
- **Segmentación**: esquema usado antiguamente que divide un programa en segmentos, un segmento puede ser un código, unos datos, la pila, etc. Un segmento se identifica por una dirección inicial y un tamaño, un segmento no tiene un tamaño fijo.



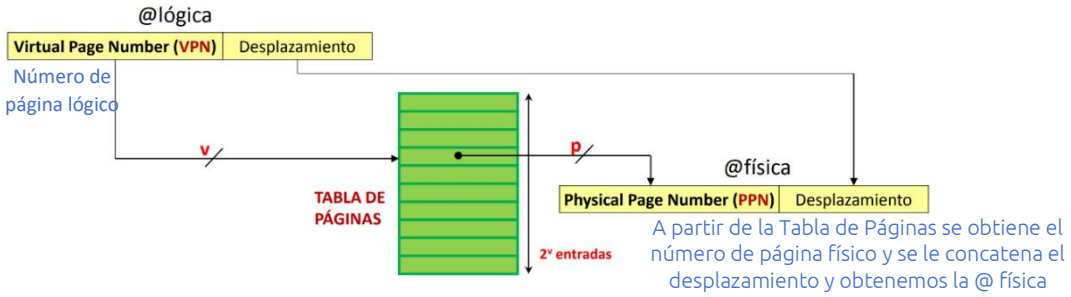
Una dirección lógica está formada por un segmento y un desplazamiento, el segmento accede al segmento que corresponde y se obtiene la @base a la que se le suma el desplazamiento para obtener la dirección física en la memoria principal.

- **Paginación:** esquema usado actualmente.
  - Divide el espacio lógico en bloques, llamadas páginas, de tamaño fijo (entre 4KB y 16KB).
  - Divide el espacio físico (MP) en frames (marcos) del tamaño de una página.

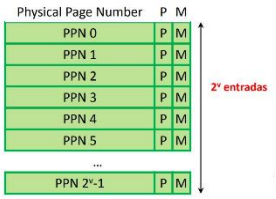
Por lo que trocea los programas en páginas, dónde cualquier @lógica puede colocarse en cualquier @física (correspondencia completamente asociativa). Las páginas se copian de disco a MP cuando son accedidas por primera vez. Debido a que la asociatividad es necesaria una estructura de datos para saber que hay en cada marco de página, a esto lo llamaremos **Tabla de Páginas del Programa**. Ejemplo:



Implementación hardware (Cómo se realiza la traducción)



Implementación de la Tabla de páginas:

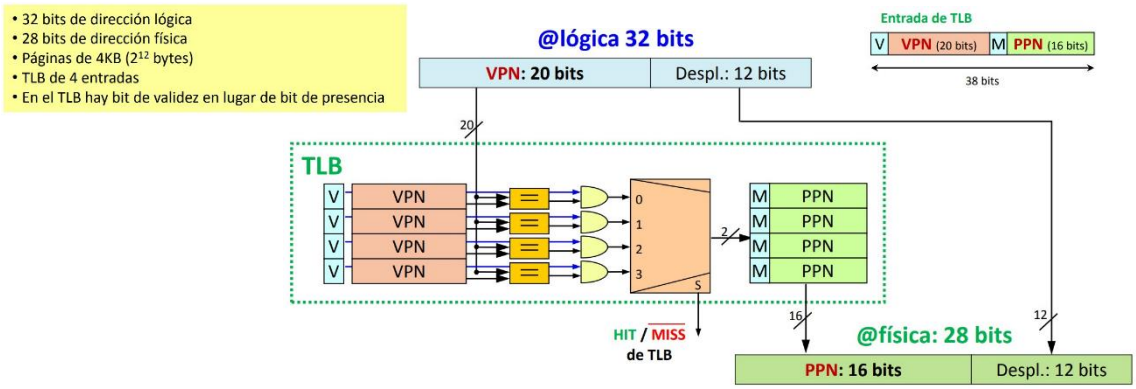


- Cada proceso tiene sus propias @lógicas y @físicas → Cada proceso tiene su propia Tabla de páginas.
- P (bit de presencia): indica si la página está almacenada en MP.
- M (bit de modificación): indica si la página a sido modificada.

Dado que la tabla de páginas es accedida en cada referencia a memoria, hace que sea un proceso muy lento debido a que hace un acceso a MP que necesita 1 acceso a la Tabla de Páginas y 1 acceso al dato, en el caso de una Tabla de Páginas de 1 nivel almacenada en MP.

Para ello, se usa la **TLB** (Translation Lookaside Buffer, Buffer de traducción anticipada), la cual no contiene datos o programas, sólo información para acelerar la traducción de direcciones. Hace el papel de "memoria cache" para la Tabla de Páginas, por lo que guarda algunas entradas de la TP.

Dado que una TP funciona de manera completamente asociativa, se usará una "cache" completamente asociativa:



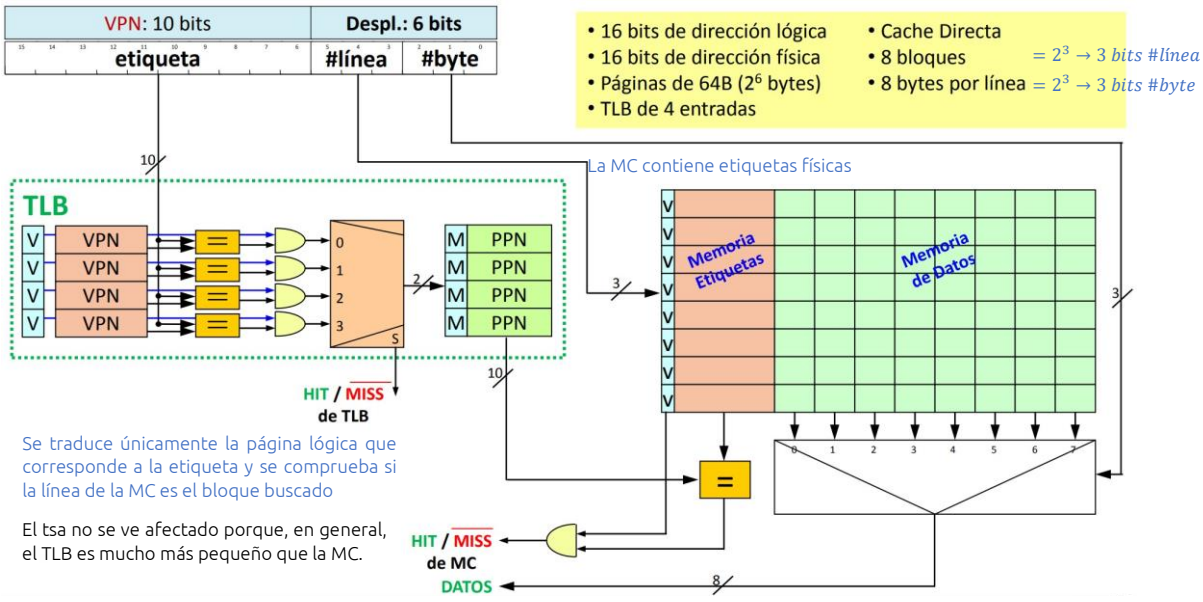
La **Memoria Virtual**:

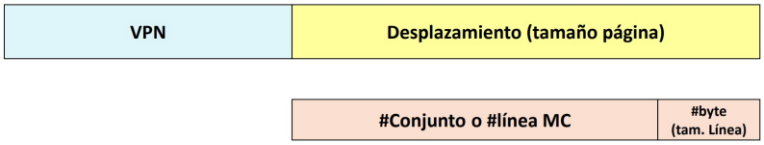
- Permite que un programa tenga un espacio lógico > espacio físico.
- Ejecutar un programa parcialmente cargado en memoria.
- Proteger el espacio de direcciones de los programas de ser accedido por otros programas.

La MV se gestiona mediante el SO y usa una política de escritura COPY BACK + WRITE ALLOCATE.

Memoria Virtual + Memoria Cache

- Traducción **antes** de acceder a MC: es la opción más lenta, ya que un acceso a memoria necesita un acceso TLB + acceso MC.
- Traducción **después** de acceder a MC:
  - la traducción se realiza sólo en cado de fallo en la MC
  - Este sistema hace que aumente el coste de un fallo de MC, porque habitualmente hay más @lógicas que @físicas, al haber más bits de @ necesitará más bits de etiqueta (será más lenta y más cara, porque será más grande y habrá más comparaciones).
  - Al hacer cambios de contexto habría que invalidar la MC porque las @lógicas serían incorrectas (falsos positivos), otra opción sería añadir bits en la MC para el PID del proceso, pero supondría un coste en tiempo mayor, en dinero y en consumo.
- Traducción en TLB y acceso a MC **simultáneos**: combina las ventajas de los sistemas anteriores, pero se restringe el tamaño de la MC → $\#\text{conjuntos} * \text{tamaño línea} \leq \text{tamaño página}$.





Conceptos Avanzados Memoria Cache

Predicción de vía

La cache con acceso paralelo accede de manera simultánea a todas las etiquetas y a todas las memorias de datos, luego selecciona la vía para comprobar si hay hit i el multiplexor selecciona la vía adecuada para la memoria de datos. Esto consume mucha energía, pero el acceso es rápido.

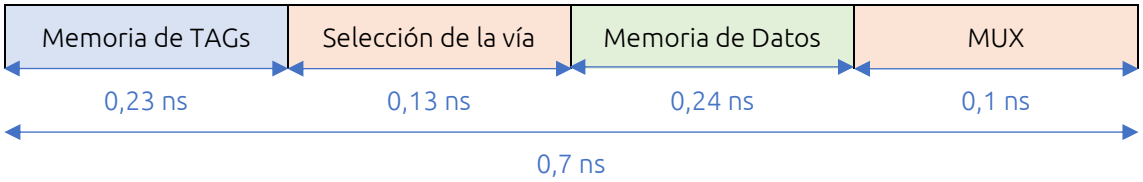
La cache con acceso secuencial accede uno a uno a la memoria de etiquetas comprobando si hay hit, si hay, entonces selecciona directamente la vía de la memoria de datos que corresponde. Esto hace que sea un proceso lento, pero reduce sustancialmente el consumo.

La cache con way prediction, hay un componente hardware que intenta predecir la vía donde se encuentra el tag i el dato correspondientes, por lo que en caso de acierto de predicción se accede sólo a una vía, en caso de fallo de predicción actúa como una cache en paralelo. Esto reduce el consumo más que una cache secuencial y reduce aún más el tiempo de acceso que la cache con acceso paralelo en caso de acierto de predicción.

El predictor está formado por una tabla, que indica dónde está una posible entrada a partir de guardar el acceso de los load y store.

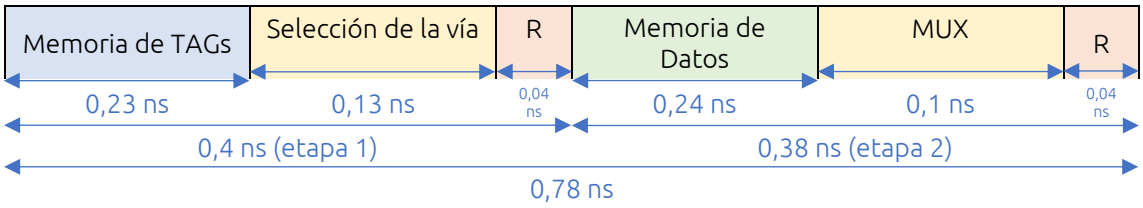
Segmentación

Las etapas que sigue un acceso a cache, de una cache secuencial, son:

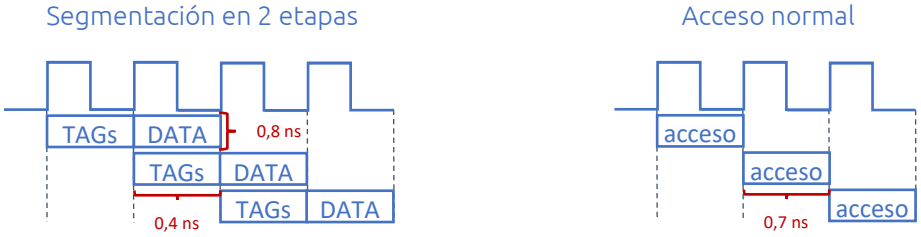


El tiempo de ciclo sería de 0,7 ns (1,43 GHz) para obtener un acceso

Si hacemos una cache segmentada de 2 etapas:

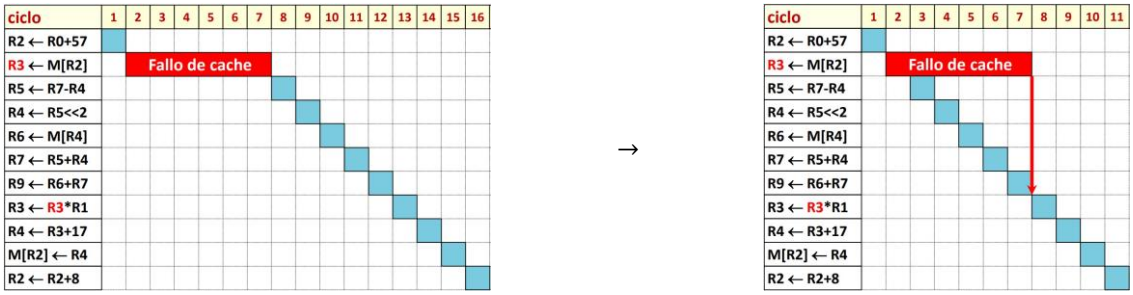


El tiempo de ciclo podría ser de 0,4 ns (el máximo entre ambas etapas), el tiempo de acceso de 1 dato sería de 0,8 ns, pero el tiempo de acceso por dato sería de 0,4 ns (2,5 GHz).



Si por ejemplo hacemos un cache segmentada de 4 etapas podríamos acceder a 1 dato por ciclo, siendo cada ciclo de 0,28 ns (3,57 GHz). Cada vez que segmentamos más la mejora es más sutil.

Non-Blocking caches (caches no bloqueantes)



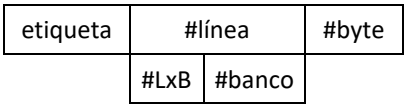
Una cache no bloqueante cuando haya un fallo de cache, en vez de pararse el procesador para esperar a resolver el fallo de cache, el procesador continua ejecutando instrucciones y sólo se parará cuando necesite un dato que está pendiente de fallo de cache. Un fallo de cache se guarda en un registro llamado **MSHRs** (Miss Status Handler Register) que gestiona los fallos pendientes, por lo que el número de registros MSHRs condiciona el número de fallos que puede soportar la MC sin detener el procesador.

Un MSHR guarda:

- El número de bloque en el que se ha producido el fallo.
- El registro destino (o registros destino, sí se ha producido más de un fallo en el mismo bloque de memoria), registro que debe dejar los datos.

Cache multibanco

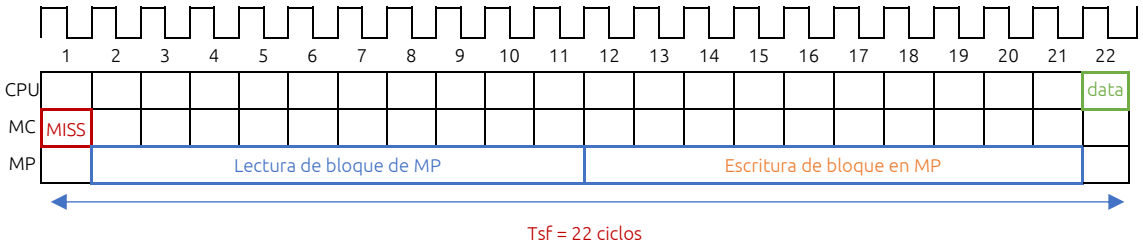
A partir del número de línea de MC (#línea MC o #conj. MC, en caso de cache asociativa), accedemos a un banco (#banco), éste es una cache más pequeña a la que se puede acceder de forma independiente, de esta manera en vez de acceder a toda la cache, accedemos sólo a una parte, reducimos el consumo y los accesos son concurrentes.



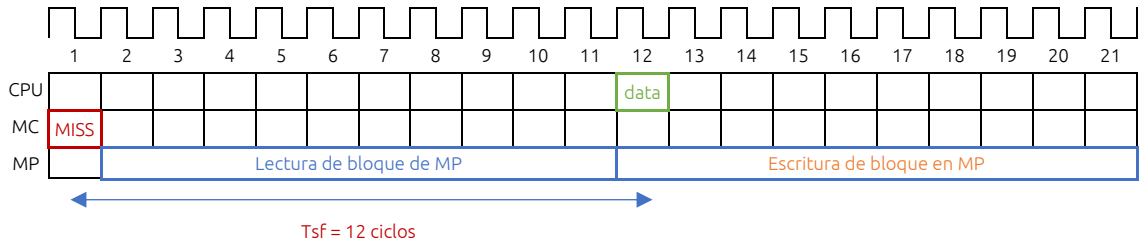
Los bits de más a la derecha tienen más variabilidad, por lo que hace que el número de conflictos se reduzcan

Técnicas de reducción de penalización en caso de fallo

En caso de Copy Back + Write Allocate, esto es lo que ocurriría en caso de fallo y de tener el dirty bit a 1:



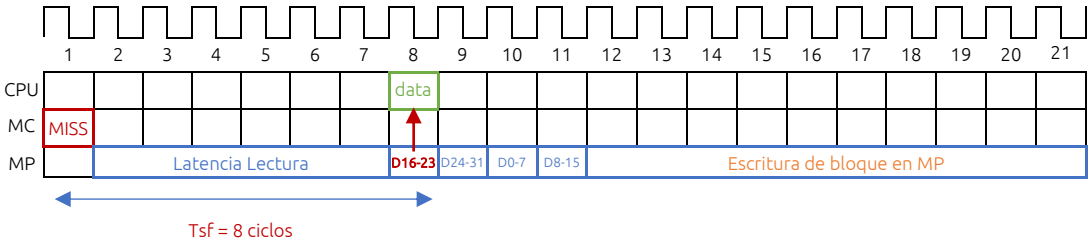
1. Si en vez de realizar primero la escritura en MP, hacemos la lectura del bloque que necesitamos para leer el dato y guardamos en buffer el bloque que debemos escribir, ahorramos tiempo. Este buffer guarda: los datos del bloque y la dirección de memoria principal dónde hay que dejar el bloque.



2. **Early Restart (continuación anticipada)**: Si partimos el proceso de lectura en: la latencia de lectura y el tiempo de transmisión de los datos (los bloques que se leen por ciclo), podemos

anticipar la lectura del byte que nos interesa para el programa en el mismo ciclo que se realiza la lectura de dicho sector del bloque.

3. **Transferencia en desorden + Continuación anticipada**: si además de la continuación anticipada, adelantamos la lectura del sector del bloque que nos interesa, el tiempo de penalización es aún menor.
   Para que el programa siga ejecutándose sin haber rellenado todo el bloque que hay que remplazar, se añade un **Fill Buffer** (que guarda dónde hay que guardar en MC) que una vez rellenado se encarga de remplazar la línea de la MC.



Para estas mejoras es **necesario** que la cache sea **no bloqueante**, ya que nos tiene que permitir seguir accediendo a memoria aunque no se haya completado la política de escritura.

Buffer de escritura

Para que se puedan producir más fallos de remplazo a la vez, el buffer en vez de ser de 1 entrada será de **n entradas**. Para prevenir que se haga una lectura de un bloque que aún no ha sido escrito en MP, hay que **consultar** el **buffer** al hacer lecturas en **MP**.

En el caso de Copy Back: el buffer está entre la MC y la MP y escribe bloques.

En caso de write through: el buffer está entre la CPU y la MP y escribe palabras.

Cuando se realizan escrituras de localidad espacial (bucle que va modificando un array), el buffer se llenará muy rápido, por lo que se usan los **Merge Buffers**, combinan múltiples escrituras consecutivas en un paquete de datos, de manera que escribe los paquetes de manera simultánea en MP, ya que escribir más de golpe es más eficiente que escribir palabra por palabra.

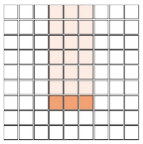
Hasta ahora, suponíamos que las lecturas y escrituras tardan lo mismo, pero la lectura de un dato se puede hacer de forma simultánea a la lectura del TAG, en cambio en una escritura no, la escritura necesita 2 ciclos (1 cliclo para leer la etiqueta y los datos y otro, en caso de acierto, de escribir el contenido). Para solucionar esto, se utilizan las **escrituras segmentadas**.

En el primer ciclo se comprueba si es un acierto o un fallo, en caso de acierto, se deja el dato a escribir en un registro intermedio y en la siguiente escritura (mientras se comprueba si es acierto o fallo) se realiza la escritura anterior, los ciclos que puede tardar entre ambos ciclos de una misma escritura es ilimitado.

Optimización de código para reducir tasa de fallos

- Reordenación del código:
  - Alinear los puntos de entrada de los bloques básicos (cuando empiezan bucles, subrutinas, …) con el inicio de la línea de cache.
  - Cambiar los saltos condicionales para que haya menos saltos y el código se ejecute de manera secuencial.
- Reordenación de los datos:
  - Alinear los datos de manera que no haya colisiones en cache (ejemplo: añadiendo variables entre dos vectores).

- Loop Fusion (Fusion de bucles): junta varios bucles en uno para aprovechar la localidad temporal, o separa un bucle en varias partes para evitar conflictos.
- Loop Interchange (Intercambio de bucles): cambiar el orden de los bucles, para que por ejemplo en vez de recorrer una matriz por columnas, hacerlo por filas (en caso de que en MP esté ordenado por filas).
- Blocking (MxV): en caso de que una fila sea demasiado grande para caber en un bloque de la MC, esta optimización reprograma el código para que se recorra la matriz en filas y columnas al mismo tiempo, es decir coge una parte de la fila de cada columna.



- Blocking (MxM): reprograma el código para recorrer varias matrices aprovechando la localidad espacial y temporal de todas las matrices, ya que recorre cada matriz de cierta manera para que se produzcan menos fallos en cache.

Prefetch

Reduce los fallos de carga. Lo hace especulando con la localidad, de manera que trae a la MC la información que cree que será utilizada en un futuro cercano, antes de ser solicitada, pero puede ser que no se use, ocupando así espacio en MC y utilizando ancho de banda entre la MC y MP.

- Prefetch Software:
  - Se utilizan instrucciones especiales para traer datos de forma anticipada.
- Prefetch Hardware:
  - Prefetch en fallo: cuando hay un fallo, además de traer el bloque necesario, traer el siguiente bloque.
  - Esquema OBL (One Block Lookahead): al acceder a un bloque, traer el siguiente bloque, siempre que este no esté en la cache ya o esté pendiente.
  - Prefetch con stride: se observa la secuencia de accesos, para adelantarse.

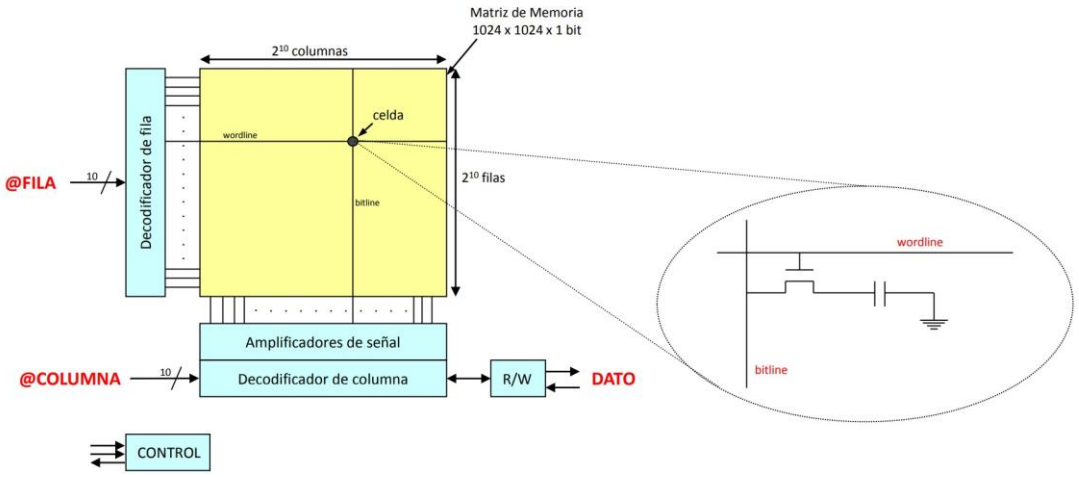
Los guarda en un buffer

RAM

Tipos de memoria:

- Memoria estática (SRAM, Static RAM): Cada celda de memoria equivale a 1 biestable (6-8 transistores) → Se usan en memorias cache.
  - Rápidas
  - Alto consumo
  - Pequeñas (poca capacidad)
  - caras
- Memoria dinámica (DRAM, Dynamic RAM): Cada celda se comporta como un condensador (1-1.x transistores) → Se suan en memorias principales.
  - Lentas
  - Bajo consumo
  - Grandes (mucha capacidad)
  - Baratas
  - Problema de refresco

Estructura interna de una DRAM



Operación de lectura de una DRAM:

1- Decodifica la @FILA, se activa la señal *wordline*.
2- Se accede a todas las celdas de la fila, los datos de toda la fila se envían a los amplificadores de señal y se recupera la tensión (el dato está en un condensador que se va descargando poco a poco)
3- Se decodifica @Columna, se selecciona una *bitline* y se envía el dato al buffer R/W.
4- Se envía el dato al exterior desde el buffer R/W.
5- La lectura destructiva, hay que reescribir la celda (y toda la fila) para recuperar el valor original y precargar los *bitlines* para el siguiente acceso a memoria.

Operación de escritura de una DRAM:

3 y 4- Prácticamente igual, la única diferencia es que la celda se reescribe con el dato que entra por el buffer R/W.
5- Hay que reescribir la celda con el nuevo valor (y el resto de la fila con el valor original).

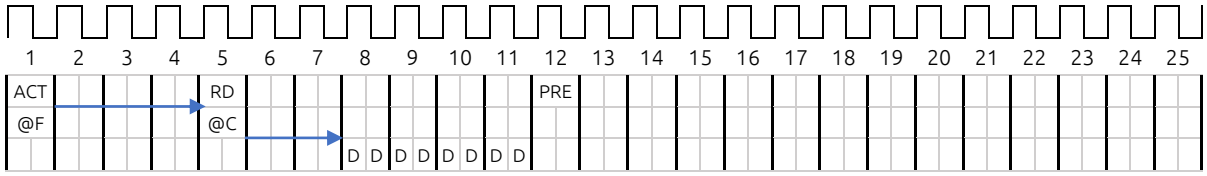
Cronogramas de acceso en Lectura y escritura

- ACTIVE (ACT)
- READ (RD)
- Decodifica la fila (@F)
- WRITE (WR)
- PRECHARGE (PRE)
- Decodifica la columna (@C)

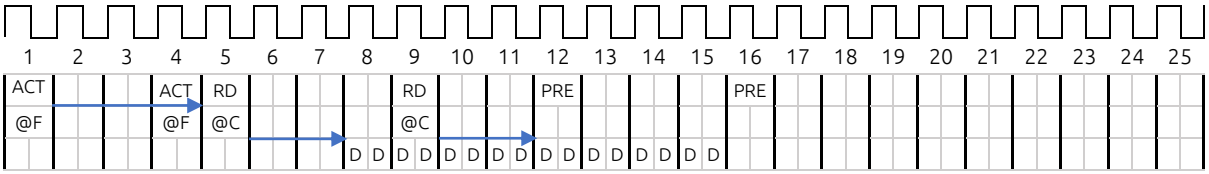
Acceso a una DRAM

Ejemplo con un DIMM de 8 chips de DDR-SDRAM (Double Data Rate Synchronous DRAM). El DIMM está configurado para leer y escribir ráfagas de 64 bytes. Latencia de fila de 4 ciclos, latencia de columna de 2 ciclos y latencia de precarga de 1 ciclo. (Cada medio ciclo se leen 8 bytes)

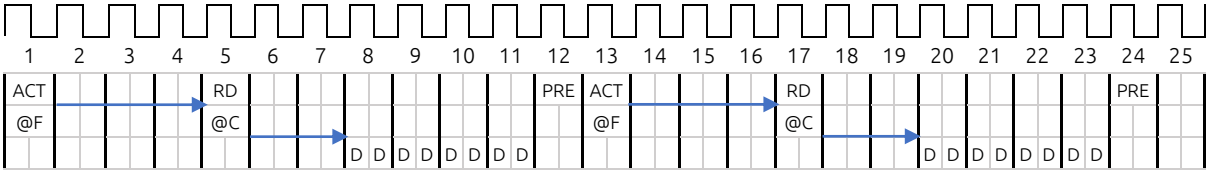
Lectura de un bloque de 64 bytes de la DDR



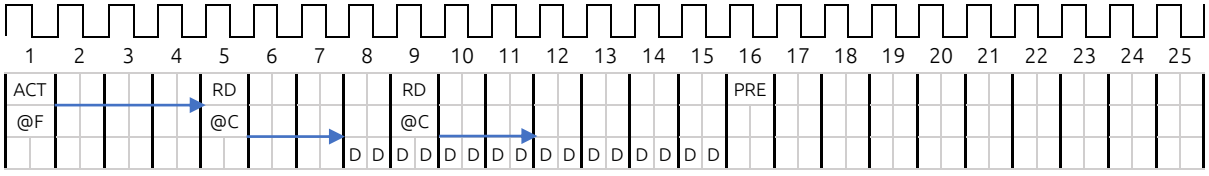
Lectura de dos bloques ubicados en distintos bancos



Lectura de dos bloques ubicados en el mismo banco pero en páginas distintas



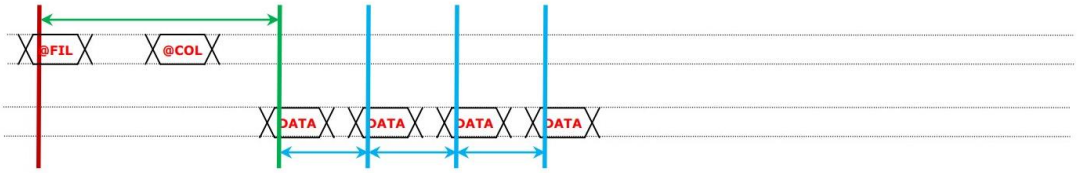
Lectura de dos bloques ubicados en el mismo banco y en la misma página



Lectura línea de cache: 4 lecturas de memoria principal

- Tiempo de acceso: 50 ns

- Latencia de columna: 15 ns



Coste de leer una línea de cache:

- Tiempo total = Tiempo de acceso + Latencia de columna * 3 = 95 ns
- Ancho de banda efectivo = #bytes / tiempo total = 32 Bytes / 95 ns = 336'84 MB/s
- Ancho de banda de pico = #bytes / latencia de columna = 8 Bytes / 15 ns = 533'33 MB/s

Tema 4 – Sistemas de Almacenamiento

RAID

- RAID 0
  - No tiene disco de paridad (No ofrece tolerancia de fallo).
  - $MTTF_{RAID\ 0} = MTTF_{disco}/\#discos$
- RAID 1
  - Utiliza discos adicionales con copias exactas de los datos ⇒ Utiliza el doble de discos.
  - Escrituras lentas, debido a que hay que escribirlas en los 2 discos.
- RAID 2 → Se usan $\log_2 \#discos$ de paridad
- RAID 3 → Se usa 1 disco para paridad
- RAID 4
  - Se usa 1 disco para paridad
  - El disco de paridad es el cuello de botella del sistema ⇒ El cuello de botella siempre se escribe en el disco de paridad.
  - Existen dos implementaciones de escritura:
    - Implementación intuitiva, necesita $N-2$ lecturas y 2 escrituras.
    - Implementación eficiente, necesita 2 lecturas y 2 escrituras
- RAID 5
  - Se usa el equivalente a 1 disco de paridad, pero está distribuido entre todos los discos.
  - Evita el cuello de botella del RAID 4.
  - En las escrituras se requieren 2 lecturas y 2 escrituras

Fiabilidad RAID 3, 4, 5 y 1 (de RAID 1 si N=2):

$MTTR$ = tiempo de cambiar disco + tiempo de reconstruir la información

$MTTF_N = MTTF_{disco}/N$ Tiempo medio hasta el fallo para N discos.

$MTTF_{N-1} = MTTF/(N-1)$ Tiempo medio hasta el fallo para N-1 discos

$$P(fallo\ 2ndo\ disco) = 1 - e^{-\lambda t} = 1 - e^{-\frac{MTTR}{MTTF_{N-1}}}$$

$$MTTF_{RAID\ 5} = MTTF_N * \frac{MTTF_{N-1}}{MTTR} = \frac{MTTF_{disco}^2}{N*(N-1)*MTTR}$$

- RAID 6
  - Se usan 2 discos para paridad, permite recuperar información aunque fallen 2 discos.
  - El sistema falla si falla un tercer disco.
  - En las escrituras siempre se necesitan 3 lecturas y 3 escrituras

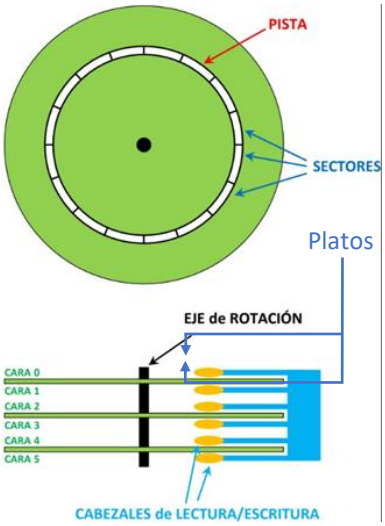
Niveles multi-RAID

$$RAID\ X+Y \neq RAID\ Y+X$$

$RAID\ X + Y$ consiste en crear grupos de discos con RAID X y después tratar estos grupos como discos individuales para crear un array con RAID Y.

Lecturas / Escrituras

Lecturas secuenciales → ancho de banda efectivo por disco * discos físicos (discos totales)
Lecturas aleatorias → ancho de banda efectivo por disco * discos físicos (discos totales)
Escrituras secuenciales → ancho de banda efectivo por disco * discos de datos
Escrituras aleatorias → ancho de banda efectivo por disco / coste escritura (depende del RAID)



Almacenamiento externo

$$\#\frac{pistas}{plato} = \frac{superficie_{plato}}{distancia\ entre\ pistas}$$

$$Capacidad_{disco} = \#platos * \#\frac{pistas}{plato} * \#\frac{sectores}{pista} * \#\frac{bytes}{sector}$$

$$Velocidad_{disco} = \frac{rpm}{60} * \#\frac{sectores}{pista} * \frac{bytes}{sector}$$

$$tiempo\ transferencia\ de\ un\ sector = \frac{\#\frac{bytes}{sector}}{velocidad_{disco}}$$

TEMA 5 – Diseño del Juego de Instrucciones

Existen diferentes tipos de arquitectura, aquí hay varios ejemplos:

Arquitectura	Pila	Acumulador	Registro / Memoria	Memoria / Memoria	Load / Store

`D = (A+B*C)/A`	`push A`
	`push B`
	`push C`
	`mul`
	`push A`
	`add`
	`div`
	`pop D`
		`load B`
		`mul C`
		`add A`
		`div A`
		`store D`
			`mov B, R0`
			`mul C, R0`
			`add A, R0`
			`div A, R0`
			`mov R0, D`
				`mul B, C, R0`
				`add A, R0, R0`
				`div A, R0, D`
					`load B, R0`
					`load C, R1`
					`mul R0, R1, R2`
					`load A, R0`
					`add R0, R2, R3`
					`div R0, R3, R4`
					`store R4, D`

Instrucciones	8	5	5	3	7
Accesos a Memoria	5	5	5	5	4

Arquitectura	Pila	Acumulador	Registro / Memoria	Memoria / Memoria	Load / Store

`A = (A-B*C)/(D+E)`	`push E`
	`push D`
	`add`
	`push C`
	`push B`
	`mul`
	`push A`
	`sub`
	`div`
	`pop A`
		`load B`
		`mul C`
		`store tmp`
		`load A`
		`sub tmp`
		`store A`
		`load D`
		`add E`
		`store tmp`
		`load A`
		`div tmp`
		`store A`
			`mov D,R1`
			`add E,R1`
			`mov B,R2`
			`mul C,R2`
			`sub R2,A`
			`div R1,A`
				`add D,E,R1`
				`mul B,C,R2`
				`sub A,R2,R2`
				`div R2,R1,A`
					`load D,R1`
					`load E,R2`
					`load B,R3`
					`load C,R4`
					`load A,R5`
					`add R1,R2,R6`
					`mul R3,R4,R7`
					`sub R5,R7,R8`
					`div R8,R6,R9`
					`store R9,A`

Instrucciones	10	12	6	4	10
Accesos a Memoria	6	12	6	6	6

Arquitectura	Pila	Acumulador	Registro / Memoria	Memoria / Memoria	Load / Store

`while(A!=B){`
`  if (A>B)`
`  A=A-B;`
`  else`
`  B=B-A;`
`}`
	`W: push A`
	`  push B`
	`  cmpne`
	`  Bfalse end`
	`  push B`
	`  push A`
	`  cmpg`
	`  Bfalse E`
	`  push B`
	`  push A`
	`  sub`
	`  pop A`
	`  br W`
	`E: push A`
	`  push B`
	`  sub`
	`  pop B`
	`  br W`
	`end:`
		`W: load A`
		`  cmpne B`
		`  Bfalse end`
		`  load A`
		`  cmpg B`
		`  Bfalse E`
		`  load A`
		`  sub B`
		`  store A`
		`  br W`
		`E: load B`
		`  sub A`
		`  store B`
		`  br W`
		`end:`
			`W: mov A,R1`
			`  cmp B,R1`
			`  Je end`
			`  Jle e`
			`  SUB B,R1`
			`  mov R1,A`
			`  br W`
			`E: sub R1,B`
			`  br W`
			`end:`
				`W: cmp B,A`
				`  je end`
				`  jle e`
				`  sub A,B,A`
				`  br W`
				`E: sub B,A,B`
				`  br W`
				`end:`
					`  load A,R1`
					`  load B,R2`
					`W: Seq R1,R2,R3`
					`  JNE R3,end`
					`  Sle R1,R2,R3`
					`  JNE e`
					`  sub R1,R2,R3`
					`  br W`
					`E: sub R2,R1,R3`
					`  br W`
					`end:`
					`  store R1,A`
					`  store R2,B`

Inst. Estáticas	18	14	9	7	12
Inst. dinámicas	13·Niter + 4	10·Niter + 3	(6 o 7)·Niter + 3	5·Niter + 2	6·Niter + 6
Accesos a Memoria	7·Niter	8·Niter	(3 o 4)·Niter	5·Niter	4

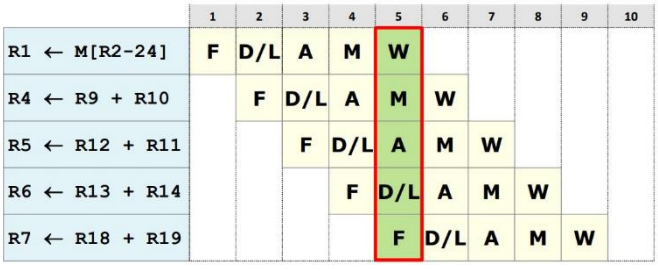
TEMA 6 – Segmentación y Paralelismo en el diseño de Computadores

Procesadores Secuenciales (CPI = 3-5)

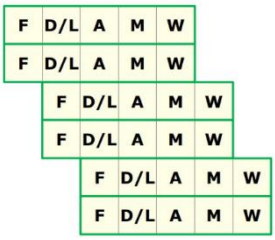
Procesadores Segmentados (CPI = 1)

Hay que tener en cuenta :

- Que no se produzcan errores de lecturas incorrectos, es decir, cuando un valor se lee antes de ser modificado por una instrucción anterior
- Que no se ejecuten instrucciones que han tenido un salto anterior a estas, es decir, si antes teníamos una instrucción de salto, hay que procurar que las instrucciones posteriores a esta, no se empiecen a ejecutar.



Procesadores Superescalares (CPI < 1)

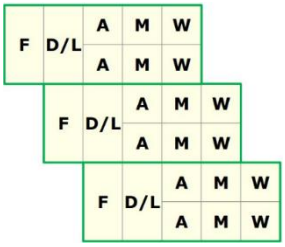


Son procesadores que disponen de múltiples unidades funcionales, que permiten iniciar más de una instrucción (u operación) por ciclo.

Sigue siendo un procesador segmentado.

Muchos procesadores superescalares permiten la ejecución fuera de orden (OoO Processors, Out of Order Processors). Las instrucciones se leen en orden, pero pueden ejecutarse en desorden. Pero las instrucciones se bloquean si sus operandos no estan disponibles, inician su ejecución cuando tiene sus operandos disponibles y la correspondiente U.F. está libre.

VLIW (CPI < 1)



Su objetivo es explotar ILP (Instruction Level Parallelism).

Una instrucción especifica múltiples operaciones independientes.

Cada operación se ejecuta en una unidad funcional predeterminada.

La planificación de las instrucciones es estática: realizada por el compilador, por lo que permite usar menos hardware de control y una arquitectura más simple.

El procesador sigue estando segmentado.

Multithreading

Son diferentes unidades funcionales dentro de un procesador que permiten ejecutar múltiples instrucciones/operaciones por ciclo.

Multiprocesadores

Es un computador con N procesadores, en cada procesador pueden ejecutarse varios threads, pertenecientes a una misma aplicación o a aplicaciones independientes. Los sistemas multiprocesador pueden utilizarse para:

- Una mayor velocidad: supercomputación – Ejecuta una aplicación de forma paralela entre todos los elementos de proceso del computador.
- Un mayor throughput: servidores de aplicaciones – Se consigue ejecutar más aplicaciones por unidad de tiempo.
- Una mezcla de ambos.

Organización:

- Multiprocesadores con memoria compartida: existe un único espacio de direcciones compartido por todos los procesadores, es decir, la red de interconexión permite aq cualquier procesador acceder a cualquier posición de memoria.
- Multiprocesadores con memoria distribuida: los procesadores sólo pueden acceder a su memoria local, es decir, la red de interconexión permite a cualquier procesador comunicarse con cualquiera de los procesadores del sistema.

Programación:

- Modelo de variables compartidas:
  - Las operaciones se dividen en procesos.
  - Los datos son compartidos por los procesos.
  - Se requieren primitivas de sincronización: señalización o acceso exclusivo.
- Modelo de paso de mensajes:
  - Las operaciones y los datos se descomponen en mensajes.
  - Los procesos sólo tienen acceso directo a los datos privados (locales).
  - Los datos no locales se acceden mediante intercambio de mensajes entre procesos.