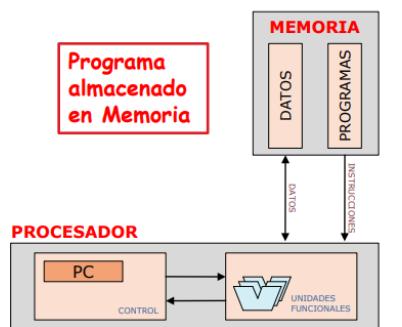


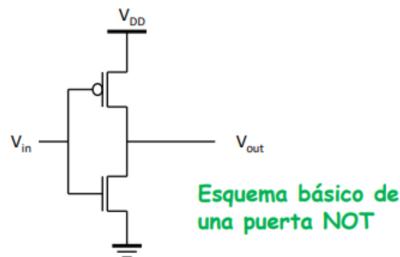
Medidas

Depende de dónde esté un KB son **1000 (DISCO)** o **1024 (RAM)**.

Máquina von Neumann



- Tecnología utilizada: **CMOS**
 - Elemento básico: **el transistor**



- Memoria accesible por dirección.
 - Las posiciones de memoria se pueden leer/escribir las veces que sean necesarias.
 - Tanto los datos como las instrucciones se almacenan en memoria.
 - No existe ninguna señal para diferenciar en memoria datos de instrucciones.
 - Las instrucciones se ejecutan en secuencia.
 - Existe un registro (**PC**) que apunta siempre a la instrucción en ejecución.
 - Existen instrucciones explícitas para romper el secuenciamiento.
 - Las instrucciones son **imperativas**, especifican **cómo** obtener los operandos, **qué** operación hay que realizar y **dónde** dejar el resultado.

- La tecnología se identifica por la longitud de la puerta del transistor medida en micrómetros (μm , 10^{-6} m) o nanómetros (nm, 10^{-9} m).
 - Tamaño del átomo de silicio: 0,111 nm

Evaluación de un sistema informático: Métricas

■ Coste:

- Tamaño del *die* (dado).
 - Complejidad:
esfuerzo requerido en el diseño, validación y fabricación del procesador.
 - Coste ambiental y social.

■ **Rendimiento:** Inversa del tiempo que tarda en completarse una tarea.

Formas básicas de mejorar el rendimiento:

- Jerarquía de Memoria
 - Paralelismo
 - Segmentación

■ **Consumo:** Energía consumida por unidad de tiempo (watos). Normalmente, mayor rendimiento requiere mayor consumo.

- **Fiabilidad:** Tiempo entre fallos/reparaciones.
Sistemas tolerantes a fallos.

Evaluación del coste

■ Factor de yield: fracción de circuitos correctos

■ Coste de un circuito integrado

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

■ Coste del *die* (dato)

$$\text{Coste del die} = \frac{\text{Coste del waffer}}{\text{Dies per waffer} \times \text{Die yield}}$$

■ Dies per wafer (oblea)

Dies per wafer (oblate)

$$\text{Dies per wafer} = \frac{\text{Area útil}}{\text{Die area}} = \frac{\pi \times (\text{diametro}/2)^2}{\text{Die area}} + \frac{\pi \times \text{diameter}}{\sqrt{2} \times \text{Die area}}$$

Waffer area / die area

Compensación por los "dies" incompletos de los bordes

■ Die yield

$$\text{Die yield} = \text{Waffer yield} \times \frac{1}{(1 + \text{defectos por unidad de area} \times \text{die area})^a}$$

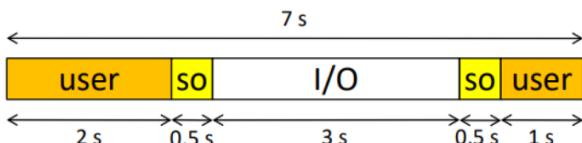
- α = medida de la complejidad, se aproxima al número de máscaras críticas

Latencia y Ancho de Banda

- **LATENCIA:** tiempo que transcurre entre la solicitud de un dato (a memoria por ejemplo) y la disponibilidad del mismo. Se mide en ciclos o unidades de tiempo (s).

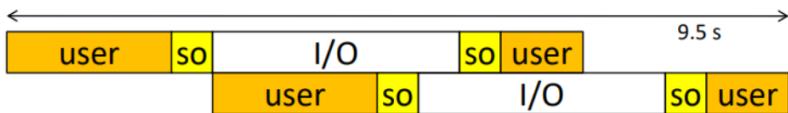
- **ANCHO de BANDA:** número de bytes transmitidos por unidad de tiempo. Se mide en KB/s, MB/s, GB/s (siempre potencias de 10).

■ Tiempo de respuesta (wall time)



- Tiempo de usuario: 3s
 - Tiempo de sistema: 1s
 - Tiempo de CPU: 4s
 - Tiempo de respuesta: 7s
 - Throughput: 1 proceso/7 s = 0,14 procesos/segundo

■ Productividad (throughput) = trabajo/tiempo



- Throughput = 2 procesos/9,5 s = 0,21 procesos/segundo

Rendimiento de un procesador

$$\frac{1}{\text{Rendimiento}} = \text{Tiempo de ejecución} = N \times \text{CPI} \times T_c$$

↑Tiempo ejecución ⇒ ↓Rendimiento

Número de instrucciones ejecutadas → Tiempo de ciclo → Número medio de ciclos por instrucción

$$\frac{1}{\text{Rendimiento}} = \frac{\text{tiempo}}{\text{Programa}} = \frac{\text{instrucciones}}{\text{Programa}} \times \frac{\text{ciclos}}{\text{instrucción}} \times \frac{\text{tiempo}}{\text{ciclo}}$$

Tiempo de ejecución

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

$$CPI = CPI_1 * I_1 + CPI_2 * I_2 + \dots + CPI_n * I_n = x \frac{\text{ciclos}}{\text{instr.}}$$

$$IPC = \frac{1}{CPI}$$

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)

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

- Para comparar el rendimiento de 2 computadores usaremos el tiempo de ejecución:

$$\text{Ganancia (Speedup)} = \frac{T_A}{T_B}$$

- Si un programa P tarda 4,5 segundos en el computador A y 2 segundos en el computador B:

$$\text{Ganancia} = \frac{T_A}{T_B} = \frac{4,5}{2} = 2,25 \Rightarrow B \text{ es 2,25 veces más rápido que A, usaremos } 2,25x$$

- También podemos usar porcentajes

$$\left(\frac{T_A}{T_B} - 1 \right) \cdot 100$$

- ✓ Una ganancia del 125% \Rightarrow B es 2.25 veces más rápido.

Límites en la mejora del Rendimiento

2. **Fórmula de Amdahl:** La ley se expresa matemáticamente de esta forma:

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

- S es la aceleración máxima o *speedup* teórico del sistema.
 - P es la fracción del programa que puede ser paralelizable (expresada en porcentaje).
 - N es el número de procesadores o núcleos de CPU.

■ Potencia y Energía

- La Energía se mide en unidades de trabajo: **Julios**
- Potencia es la energía consumida por unidad de tiempo: **Watios** (Julios/seg)

■ Energía y potencia eléctricas

$$\text{Potencia} = I \times V = \text{amperios} \times \text{voltios} = \text{watios}$$

$$\text{Energía} = P \times t = I \times V \times t = \text{amperios} \times \text{voltios} \times \text{segundo} = \text{watios} \times \text{segundo} = \text{julios}$$

■ La Potencia consumida por un circuito CMOS tiene 3 componentes:

- **Comutación:** debido a la conmutación entre niveles de tensión en la carga capacitiva efectiva de todo el chip.
- **Corriente de fugas:** los transistores no son ideales.
- **Corriente de cortocircuito:** los dos transistores del inversor están activos cuando la entrada cambia de tensión.

Potencia y energía de conmutación/dinámica

$$P = C * V^2 * f = \text{faradios} * \text{voltios}^2 * \text{hercios}$$

$$E = C * V^2 = \text{faradios} * \text{voltios}^2$$

Potencia de fugas/estática

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

■ Métricas de eficiencia

$$\text{Eficiencia energética} = \frac{\text{rendimiento}}{\text{watio}} = \frac{1}{\text{tiempo} \times \text{watio}} = \frac{1}{\text{Energía consumida}}$$

■ Métricas para caracterizar la fiabilidad:

- Fiabilidad: tiempo de funcionamiento continuo sin fallos
 - ✓ $\text{MTTF} = \text{Mean Time To Failure}$
- Tasa de fallos (Failure rate)

$$\lambda = \frac{1}{\text{MTTF}}$$

- Interrupción del servicio se mide como el tiempo medio necesario para restablecerlo
 - ✓ $\text{MTTR} = \text{Mean Time To Repair}$
- Tiempo medio entre fallos (Mean Time Between Failures)
 - ✓ $\text{MTBF} = \text{MTTF} + \text{MTTR}$
- Disponibilidad (availability): Fracción del tiempo en que un sistema está funcionando.

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

■ El tiempo entre fallos se aproxima a una distribución exponencial donde:

- p = probabilidad de que se produzca un fallo
- $\lambda = 1/\text{MTTF}$ (failure rate)
- t = tiempo transcurrido

$$p = 1 - e^{-\lambda t}$$

■ Ejercicio

Sistema formado por:

- 1 CPU (incluye placa base y memoria) $\text{MTTF} = 1.000.000$ horas
- 2 Discos $\text{MTTF} = 500.000$ horas
- 1 Fuente de alimentación $\text{MTTF} = 200.000$ horas

Calcular el MTTF del sistema:

$$\frac{1}{\text{MTTF}_{\text{sistema}}} = \frac{1}{\text{MTTF}_{\text{CPU}}} + \frac{1}{\text{MTTF}_{\text{disco}}} + \frac{1}{\text{MTTF}_{\text{fuente}}} = \frac{1}{10^6} + \frac{2}{500 \cdot 10^3} + \frac{1}{200 \cdot 10^3} = \frac{1+4+5}{10^6} = \frac{1}{10^5}$$

$$\text{MTTF}_{\text{sistema}} = 100.000 \text{ horas}$$

Lenguaje Máquina MIPS (características RISC)

- Instrucciones aritméticas acceden sólo a registros
 - En algunos casos un operando puede ser inmediato
- Solo las instrucciones Load y Store pueden acceder a memoria.
- Referencias a memoria con modos de direccionamiento simples
 - Base + Desplazamiento
- Instrucciones de longitud fija
- Muchos registros
- EIP: Contador de programa. Apunta a la siguiente instrucción a ejecutar
- Registros: Se usan muy frecuentemente como variables de acceso rápido
- Códigos de Condición
 - Almacenan información respecto al comportamiento de las últimas instrucciones ejecutadas
 - Se usan en los saltos condicionales
- Memoria
 - Vector direccionable a nivel de byte, Little Endian
 - Código, datos usuario, datos SO
 - Pila para soportar gestión de subrutinas

Visión del programador

Espacio de memoria

- Espacio lineal de 2^{32} posiciones de 1 byte: [0 - $2^{32}-1$]
- Modo protegido / Modelo plano de memoria/ Little Endian

Registros disponibles

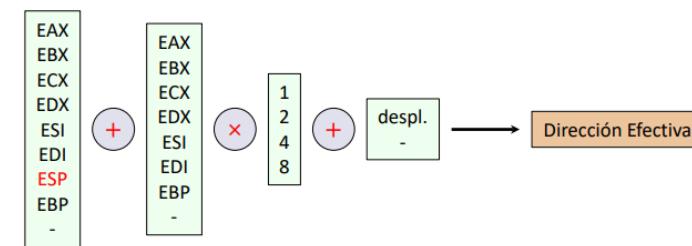
32 bits	16 bits	8 bits	
%eax	%ax	%ah, %al	
%ebx	%bx	%bh, %bl	
%ecx	%cx	%ch, %cl	
%edx	%dx	%dh, %dl	
%esi	%si		
%edi	%di		
%esp	%sp		Reservados para uso específico de subrutinas
%ebp	%bp		
%eip			Contador programa
%eflags			Palabra de estado

Lenguaje Máquina x86 (características CISC)

- Instrucciones pueden referenciar diferentes tipos de operandos
 - inmediato, registros, memoria
- Instrucciones aritméticas pueden leer/escribir en memoria, pero sólo 1 de los 2 operandos puede estar en memoria
- Referencias a memoria pueden suponer cálculos complejos
 - $Rb + S \cdot Ri + D$
- Instrucciones pueden tener diferente longitud
- Pocos registros

Modos de direccionamiento

- Inmediato: \$19, \$-3, \$0x2A, \$0x2A45
 - Codificado con 1, 2 ó 4 bytes
- Registro: %eax, %ah, %esi
- Memoria: $D(Rb, Ri, s) \rightarrow M[Rb+Rixs+D]$
 - D: desplazamiento codificado con 1, 2 ó 4 bytes
 - Rb: registro base. Cualquiera de los 8 registros
 - Ri: registro índice. Cualquier excepción %esp
 - S: factor escala: 1, 2, 4 u 8



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

Instrucciones	Descripción	Notas	Ejemplo
MOVx op1, op2	op2 \leftarrow op1	x = {L, W, B}	MOV B \$-1,%AL
MOV\$xy op1, op2	op2 \leftarrow ExtSign(op1)	xy = {BW, BL, WL}	MOVSBW %CH,%AX
MOVZxy op1, op2	op2 \leftarrow ExtZero(op1)	xy = {BW, BL, WL}	MOVZWL %BX,%EDX
PUSHL op1	%ESP \leftarrow %ESP - 4; M[%ESP] \leftarrow op1		PUSHL 12(%EBP)
POPL op1	op1 \leftarrow M[%ESP]; %ESP \leftarrow %ESP + 4;		POPL %EAX
LEAL op1, op2	op2 \leftarrow &op1	op1: memoria	LEAL (%EBX,%ECX),%EAX

Instrucciones	Descripción	Notas	Ejemplo
ADDx op1, op2	op2 \leftarrow op2+op1	x = {L, W, B}	ADDL \$13,%EAX
SUBx op1, op2	op2 \leftarrow op2-op1	x = {L, W, B}	SUBW %CX,%AX
ADCx op1, op2	op2 \leftarrow op2+op1+CF	x = {L, W, B}	ADCL %EDX,%EAX
SBBx op1, op2	op2 \leftarrow op2-op1-CF	x = {L, W, B}	SBBL %ECX,%EAX
INCx op1	op1 \leftarrow op1+1	x = {L, W, B}	INCL %EAX
DECx op1	op1 \leftarrow op1-1	x = {L, W, B}	DECW %BX
NEGx op1	op1 \leftarrow -op1	x = {L, W, B}	NEGL %EAX

Instrucciones	Descripción	Notas	Ejemplo
IMUL op1, op2	op2 \leftarrow op2·op1	op2: registro	IMUL (%EBX),%EAX
IMUL inm,op1,op2	op2 \leftarrow op1·inm	inm: constante	IMUL \$3,%EAX,%ECX
IMULL op1	%EDX%EAX \leftarrow op1·%EAX	op1: mem. o reg. (Enteros)	IMULL (%EBX)
MULL op1	%EDX%EAX \leftarrow op1·%EAX	op1: mem. o reg. (Naturales)	MULL (%EBX)
CLTD	%EDX%EAX \leftarrow ExtSign(%EAX)		CLTD
IDIVL op1	%EAX \leftarrow %EDX%EAX / op1 %EDX \leftarrow %EDX%EAX % op1	op1: mem. o reg. (Enteros)	IDIVL (%EBX)
DIVL op1	%EAX \leftarrow %EDX%EAX / op1 %EDX \leftarrow %EDX%EAX % op1	op1: mem. o reg. (Naturales)	DIVL %ESI

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

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

Ejemplo de IF-THEN-ELSE

Ejemplo:

```
int max(int x, int y) {  
    int max;  
    if (x>y) max = x;  
    else max = y;  
    return max;  
}
```

Traducción:

```
max: pushl %ebp  
      movl %esp, %ebp  
      subl $4, %esp  
      movl 8(%ebp), %ecx  
      cmpl 12(%ebp), %ecx  
      jle else  
if:   movl 8(%ebp), %eax  
      jmp endif  
else: movl 12(%ebp), %eax  
endif: movl %ebp, %esp  
       popl %ebp  
       ret  
# x→8[%ebp]  
# y→12[%ebp]  
# resultado en %eax
```

Ejemplo 1 de WHILE

Traducción:

```
gcd:  pushl %ebp  
      movl %esp, %ebp  
while: cmpl $0, 12(%ebp)  
       je end  
       movl 8(%ebp), %eax  
       cmpl 12(%ebp), %eax  
       jle else  
       subl 12(%ebp), %eax  
       movl %eax, 8(%ebp)  
       jmp endif  
else: subl %eax, 12(%ebp)  
endif: jmp while  
end:  popl %ebp  
      ret  
# a→8[%ebp]  
# b→12[%ebp]
```

Ejemplo de DO-WHILE

Ejemplo:

```
int ContA(char v[]) {  
    int i, cont;  
    cont = 0;  
    i = 0;  
    do {  
        if (v[i] == 'a') cont++;  
        i++;  
    } while (v[i] != '.');  
    return cont;  
}
```

Traducción:

```
ContA: pushl %ebp  
      movl %esp, %ebp  
      subl $8, %esp  
      movl $0, %eax # cont  
      movl $0, %edx # i  
do:   movl 8(%ebp), %ecx  
      cmpb $'a', (%ecx,%edx)  
      jne endif  
      incl %eax;  
endif: incl %edx  
      cmpb $'.', (%ecx,%edx)  
      jne do  
end:  movl %ebp, %esp  
      popl %ebp  
      ret  
# @v→8[%ebp]
```

Ejemplo de FOR

Traducción:

```
sumV: pushl %ebp  
      movl %esp, %ebp  
      subl $4, %esp  
      movl $0, %eax # sum  
      movl $0, %ecx # i  
for:  cmpl 12(%ebp), %ecx  
      jge end  
      movl 8(%ebp), %edx  
      addl (%edx,%ecx,4), %eax  
      incl %ecx  
      jmp for  
end:  movl %ebp, %esp  
      popl %ebp  
      ret  
# @v→8[%ebp]  
# N→12[%ebp]
```

Vectores

■ Declaración en C:

```
tipo nombre[tamaño]; //indexado a partir de 0
```

■ Almacenamiento en posiciones consecutivas de memoria

- Acceso elemento $V[i]: @\text{inicio } V + i \cdot \text{tam}$ (tam: tamaño de los elementos de V)

■ Ejemplos:

Declaración en C	Tamaño elemento	Tamaño vector	@elemento i
char A[12];	1B	12B	@\text{inicio } A + i
char *B[80];	4B	320B	@\text{inicio } B + 4 \cdot i
double C[1024];	8B	8KB	@\text{inicio } C + 8 \cdot i
int *D[5];	4B	20B	@\text{inicio } D + 4 \cdot i
int E[100];	4B	400B	@\text{inicio } E + 4 \cdot i

Ejemplo:

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

Traducción:

```
Vi: 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

■ Declaración en C:

```
tipo nombre[NumFilas][NumColumnas]; //indexado a partir de (0,0)
```

■ Almacenamiento por filas en posiciones consecutivas de memoria

- Acceso elemento $A[i][j]: @\text{inicio } A + (i \cdot \text{NumColumnas} + j) \cdot \text{tam}$ (tam: tamaño de los elementos de A)

■ Ejemplos:

Declaración en C	Tamaño elemento	Tamaño matriz	@elemento (i, j)
char A[80][25];	1B	2000B	@\text{inicio } A + i \cdot 25 + j
char *B[80][10];	4B	3200B	@\text{inicio } B + (i \cdot 10 + j) \cdot 4
double C[1024][100];	8B	800KB	@\text{inicio } C + (i \cdot 100 + j) \cdot 8
int *D[5][90];	4B	1800B	@\text{inicio } D + (i \cdot 90 + j) \cdot 4
int E[100][30];	4B	12000B	@\text{inicio } E + (i \cdot 30 + j) \cdot 4

Matrices 3-dimensiones

■ Ejemplo, matriz de enteros de 3 dimensiones:

```
int M3D[10][64][48]; // cada int ocupa 4 bytes
```

■ La matriz se almacena en posiciones consecutivas de memoria: cara a cara y en cada cara por filas.

■ Acceso al elemento $M3D[\text{cara}][\text{fila}][\text{columna}]$:

- $@\text{inicio } M3D + (\text{cara} \cdot 64 \cdot 48 + \text{fila} \cdot 48 + \text{columna}) \cdot 4$

Ejemplo:

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

Traducción:

```
Mfc: 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
```

Ejemplo:

```
void Copia(int M[50][80], int X[50][80]) {
    int i, j;
    for (i=0; i<50; i++)
        for (j=0; j<80; j++)
            M[i][j] = X[i][j];
}
```

Traducción:

```
Copia: pushl %ebp          # i → %ecx
    movl %esp, %ebp          # j → %edx
    salvar reg
    movl 8(%ebp),%edi # @M → 8[%ebp]
    movl 12(%ebp),%esi # @X → 12[%ebp]
    xorl %ecx, %ecx
fori: cmpl $50, %ecx
    jge endi
    cuerpo-FORi
    incl %ecx
    jmp fori
endi: restaurar reg
    popl %ebp
    ret
```

#cuerpo-FORi:

```
xorl %edx, %edx
```

```
forj: cmpl $80, %edx
```

```
jge endj
```

```
cuerpo-FORj
```

```
incl %edx
```

```
jmp forj
```

```
endj:
```

```
#cuerpo-FORj:
imull $80, %ecx, %eax
addl %edx, %eax
movl (%esi,%eax,4),%ebx
movl %ebx,(%edi,%eax,4)
```

Instrucciones ejecutadas: $15 + 50 \cdot (7 + 80 \cdot 8) = 32.267$

■ Instrucciones SIMD (Single Instruction Multiple Data)

Optimización: Desenrollar 4 + SIMD

```
Copia: pushl %ebp
        movl %esp, %ebp
        salvar reg
        movl 8(%ebp),%edi # @M → 8[%ebp]
        movl 12(%ebp),%esi # @X → 12[%ebp]
        xorl %ecx,%ecx
loop:   movdqa (%esi,%ecx,4),%xmm0
        movdqa %xmm0,(%edi,%ecx,4)
        addl $4,%ecx
        cmpl $4000,%ecx
        jl loop
        restaurar reg
        popl %ebp
        ret

movdqa: mov double quadword (128 bits) aligned
        (dirección de inicio de X y M debe ser múltiplo de 16)
        (existe movdq u=unaligned pero es menos eficiente)
%xmm0: registro de 128 bits para las extensiones SSE
        (en 128 bits podemos almacenar 4 enteros)
```

Objetivo: reducir el número de instrucciones.

- El bucle se ejecuta 1/4 de veces y además tiene menos instrucciones

Instrucciones ejecutadas: $11 + 1000 \cdot 5 = 5.011$

■ Ejemplo: Calcular el vector máximo de 2 vectores de caracteres

```
void maxv(char a[], b[], max[]) {
    int i;
    for (i=0;i<8000;i++) {
        if (a[i]>b[i]) max[i]=a[i];
        else max[i]=b[i];
    }
}
```

```
for:  cmpl $8000, %esi      ; i < 8000
      jge fin
      movdqa (%ebx, %esi),%xmm0
      pmaxsb (%ecx, %esi),%xmm0
      movdqa %xmm0, (%edx, %esi)
      addl $16, %esi
      jmp for

fin:   ; Restaurar Registros
      popl %ebp
      ret
```

9 instrucciones cada iteración

vs

7 instrucciones cada 16 iteraciones

```
maxv: pushl %ebp
        movl %esp, %ebp
        ; Salvar Registros
        movl 8(%ebp),%ebx      ; ebx ← @a
        movl 12(%ebp),%ecx      ; ecx ← @b
        movl 16(%ebp),%edx      ; edx ← @max
        xorl %esi,%esi         ; i ← 0
```

```
for:  cmpl $8000, %esi      ; i < 8000
      jge fin
      movb (%ebx, %esi),%al ; al ← a[i]
      cmpb (%ecx, %esi),%al ; a[i] > b[i]
      jle else
      movb %al, (%edx, %esi); max[i]←a[i]
      jmp cont
else:  movb (%ecx, %esi),%al
      movb %al, (%edx, %esi); max[i]←b[i]
cont:  incl %esi           ; i++
      jmp for

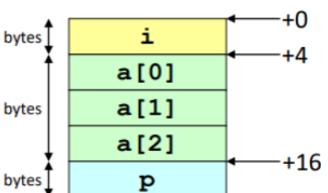
fin:   ; Restaurar Registros
      popl %ebp
      ret
```

■ Estructuras (struct)

- conjunto heterogéneo de datos
 - ✓ almacenados de forma contigua en memoria
 - ✓ referenciados por su nombre

Ejemplo:

```
typedef struct {
    int i;
    int a[3];
    int *p;
} X;
X S;
Init(&S);
```



Ejemplo:

```
void Init (X *S) {
    (*S).i = 1;
    S->a[2] = 0;
    S->p = &(*S).a[0];
}
```

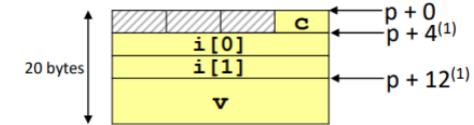
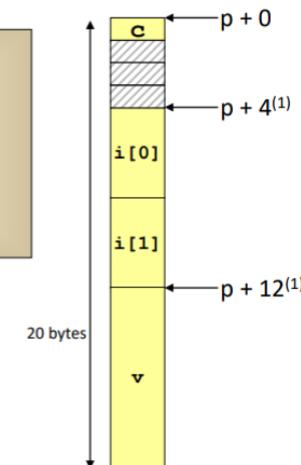
Traducción:

```
Init: push %ebp
      movl %esp, %ebp
      movl 8(%ebp), %edx
      movl $1, (%edx)
      movl $0,12(%edx)
      leal 4(%edx), %eax
      movl %eax,16(%edx)
      popl %ebp
      ret
```

■ **k = 4** debido a que el elemento *double* se trata a nivel de alineamiento como un elemento de 4 bytes.

Ejemplo:

```
struct S1 {
    char c;
    int i[2];
    double v;
}*p;
```



¡La dirección de inicio y el tamaño de la estructura han de ser múltiplo de 4!

(1) Múltiplo de 4

Alineamiento de datos

Alineamiento en linux-32 (gcc):

- **char** (1 byte): alineado a 1-byte (**no hay restricciones en la @**)
- **short** (2 bytes): alineado a 2-bytes (**el bit más bajo de la @ debe ser 0**)
- **int** (4 bytes): alineado a 4-bytes (**los 2 bits más bajos de la @ deben ser 00**)
- **puntero** (4 bytes): alineado a 4-bytes
- **double** (8 bytes): alineado a 4-bytes
- **Long double** (12 bytes): alineado a 4-bytes

Offsets dentro de una estructura:

- deben satisfacer los requerimientos de alineamiento de sus elementos

Dirección de la estructura

- Cada estructura tiene un requerimiento de alineamiento k
- k = el mayor de los alineamientos de cualquier elemento
- La @ inicial y el tamaño de la estructura debe ser múltiplo de k

Los parámetros se pasan por la pila de derecha a izquierda

- Los vectores y matrices siempre se pasan por referencia
- Los structs se pasan por valor, no importa el tamaño
- Los parámetros de tipo carácter (1 byte) ocupan 4 bytes
- Los parámetros de tipo short (2 bytes) ocupan 4 bytes

Las variables locales están alineadas en la pila con la misma convención que en un struct

- Char en cualquier dirección
- Short en direcciones múltiplos de 2
- Integer en direcciones múltiplos de 4
- El tamaño del conjunto de variables locales debe ser múltiplo de 4 para que la pila quede bien alineada

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 una subrutina.
Si es necesario, el LLAMADOR ha de salvarlos

Los resultados se devuelven siempre en %eax

La pila debe quedar siempre alineada a 4

Terminología

- Parámetros
- ✓ **Valor**
- ✓ **Referencia**
- Variables locales
- **Invocación**
- Retorno resultado
- Cuerpo subrutina

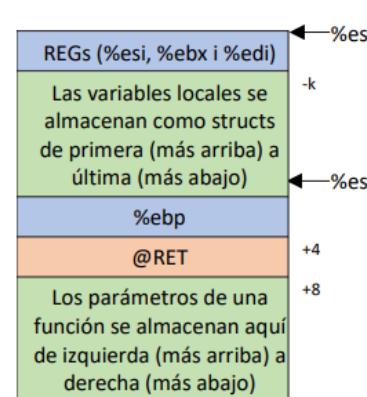
```
int DOT(int v1[], int v2[], int N) {
    int i, sum;
    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];
    return sum;
}
```

```
void PDOT(int M[10][10], int *p) {
    int i;
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0], &M[i][0], 10);
}
```

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

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

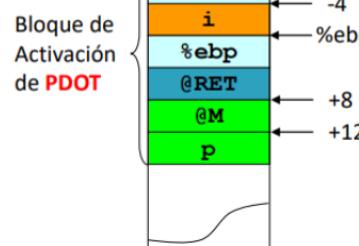
Bloque de activación

PILA

```
{código llamador PDOT}
empilar parámetros PDOT
call PDOT
...

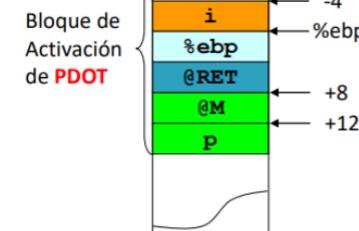
```

```
PDOT: pushl %ebp
movl %esp, %ebp
subl $4, %esp
salvar registros
-
-
-
```



1. Paso de parámetros

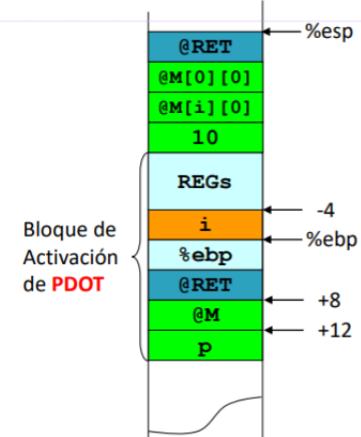
```
PDOT: -
-
-
pushl $10
imull $10,-4(%ebp),%edx
movl 8(%ebp),%ebx
leal (%ebx,%edx,4),%eax
pushl %eax
pushl %ebx
```



2. Llamada a la subrutina

```
PDOT: -
-
-
pushl $10
imull $10,-4(%ebp),%edx
movl 8(%ebp),%ebx
leal (%ebx,%edx,4),%eax
pushl %eax
pushl %ebx
call DOT
```

```
void PDOT(int M[10][10], int *p) {
    int i;
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0],&M[i][0],10);
}
```



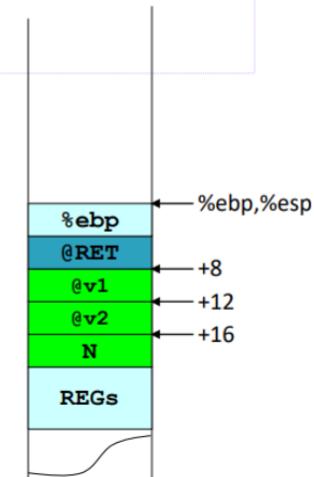
3. Enlace dinámico y puntero al bloque de activación

```
DOT: pushl %ebp
movl %esp, %ebp
```

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;
```

```
    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];
```

```
    return sum;
}
```



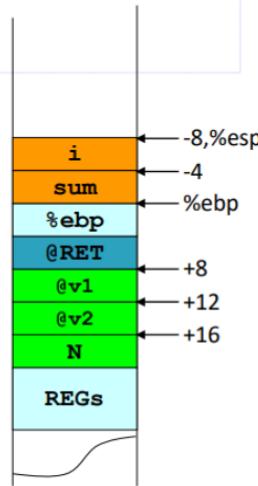
4. Reserva espacio para variables locales

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
```

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;

    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];

    return sum;
}
```



5. Salvar estado del llamador

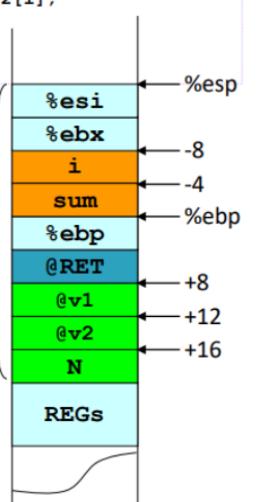
```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
```

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;

    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];

    return sum;
}
```

Bloque de Activación de DOT



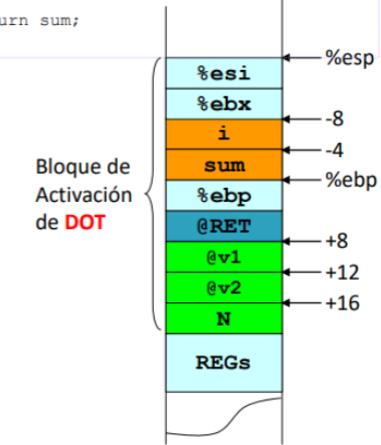
6. Cuerpo subrutina

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp), %ebx
      movl 12(%ebp), %esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for: cmpl 16(%ebp), %edx
      jge end
      movl (%esi,%edx,4), %ecx
      imull (%ebx,%edx,4), %ecx
      addl %ecx, -4(%ebp)
      incl %edx
      jmp for
end:
```

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;

    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];

    return sum;
}
```



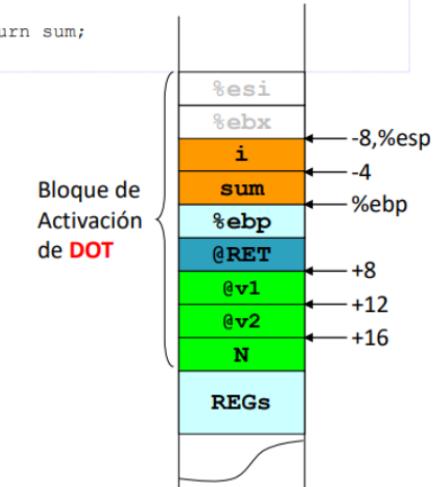
8. Restaurar estado llamador

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp), %ebx
      movl 12(%ebp), %esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for: cmpl 16(%ebp), %edx
      jge end
      movl (%esi,%edx,4), %ecx
      imull (%ebx,%edx,4), %ecx
      addl %ecx, -4(%ebp)
      incl %edx
      jmp for
end: movl -4(%ebp), %eax
      popl %esi
      popl %ebx
```

```
int DOT(int v1[], int v2[], int N) {
    int i, sum;

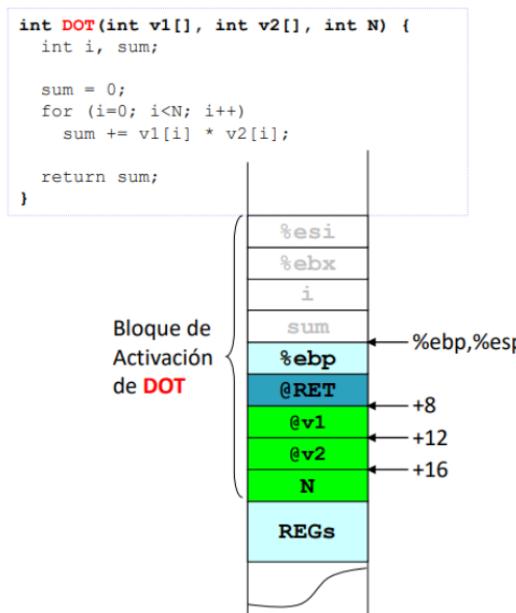
    sum = 0;
    for (i=0; i<N; i++)
        sum += v1[i] * v2[i];

    return sum;
}
```



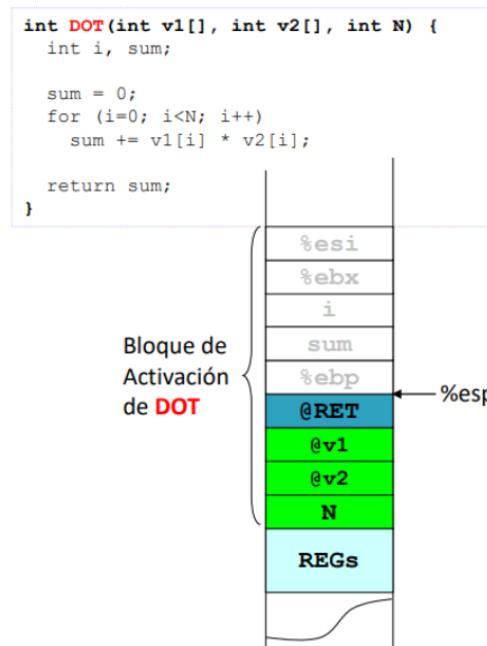
9. Eliminar variables locales

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp), %ebx
      movl 12(%ebp), %esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp), %edx
      jge end
      movl (%esi,%edx,4), %ecx
      imull (%ebx,%edx,4), %ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end: movl -4(%ebp), %eax
      popl %esi
      popl %ebx
      movl %ebp,%esp
```



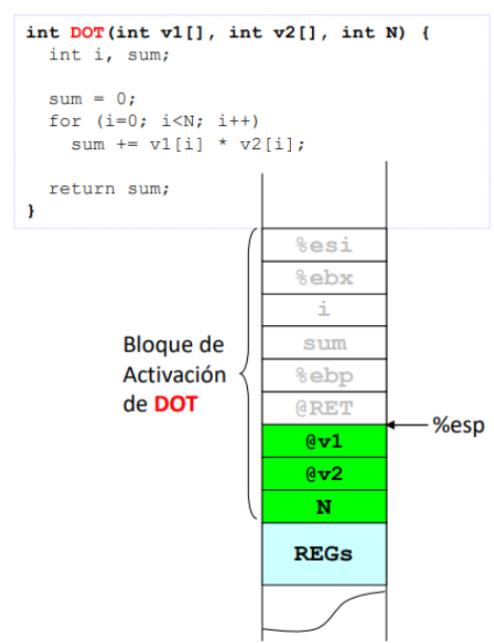
10. Deshacer enlace dinámico

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp), %ebx
      movl 12(%ebp), %esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp), %edx
      jge end
      movl (%esi,%edx,4), %ecx
      imull (%ebx,%edx,4), %ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end: movl -4(%ebp), %eax
      popl %esi
      popl %ebx
      movl %ebp,%esp
      popl %ebp
```



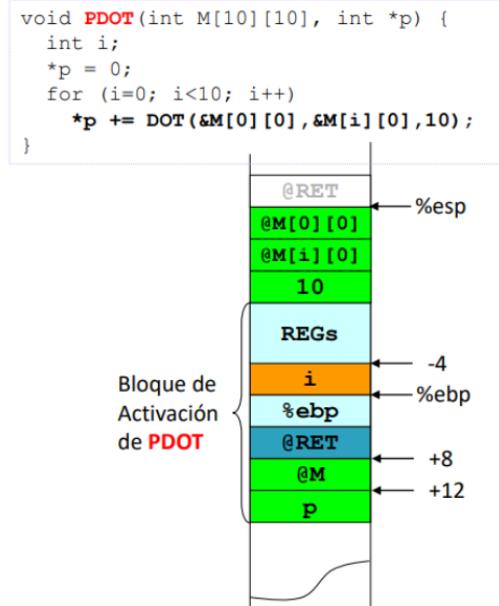
11. Retorno subrutina

```
DOT: pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp), %ebx
      movl 12(%ebp), %esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp), %edx
      jge end
      movl (%esi,%edx,4), %ecx
      imull (%ebx,%edx,4), %ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end: movl -4(%ebp), %eax
      popl %esi
      popl %ebx
      movl %ebp,%esp
      popl %ebp
      ret
```



11. Volvemos a la subrutina

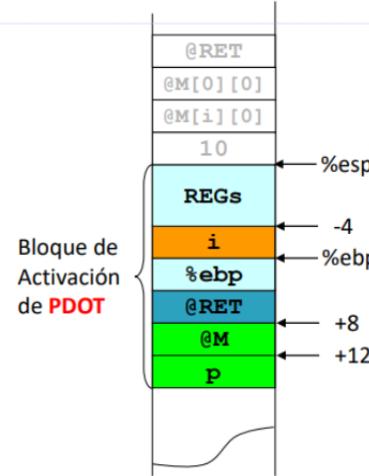
```
PDOT: -
      -
      -
      pushl $10
      imull $10,-4(%ebp),%edx
      movl 8(%ebp), %ebx
      leal (%ebx,%edx,4), %eax
      pushl %eax
      pushl %ebx
      call DOT
```



12. Eliminar parámetros

```
PDOT: -
-
-
pushl $10
imull $10,-4(%ebp),%edx
movl 8(%ebp),%ebx
leal (%ebx,%edx,4),%eax
pushl %eax
pushl %ebx
call DOT
addl $12,%esp
```

```
void PDOT(int M[10][10], int *p) {
    int i;
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0],&M[i][0],10);
}
```



Gestión de Registros

- Los registros **%eax, %ecx, %edx** se pueden modificar en el interior de una subrutina.
Si es necesario, el **LLAMADOR** ha de salvarlos

```
PDOT: -
-
-
movl $0, %ecx
for: cmpl $10, %ecx
        jge ffor
        pushl %ecx
        pushl $10
        imull $10,%ecx,%edx
        movl 8(%ebp),%ebx
        leal (%ebx,%edx,4),%eax
        pushl %eax
        pushl %ebx
        call DOT ;puede machacar %ecx
        addl $12,%esp
        movl 12(%ebp),%ebx
        addl %eax,(%ebx)
        popl %ecx
        incl %ecx
        jmp for:
ffor:
```

Mejor aun usar %ebx, %esi o %edi en lugar de %ecx como contador de bucle

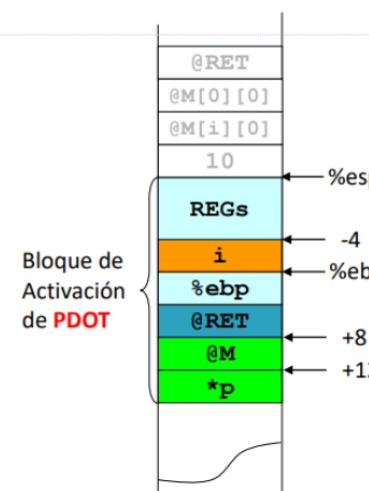
```
void PDOT(int M[10][10], int *p) {
    int i; // almacenamos i en %ecx
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0],&M[i][0],10);
}
```

```
DOT: pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        pushl %ebx
        pushl %esi
        ...
        movl (%esi,%edx,4),%ecx
        imull (%ebx,%edx,4),%ecx
        ...
        popl %esi
        popl %ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

13. Recoger/usar resultado

```
PDOT: -
-
-
pushl $10
imull $10,-4(%ebp),%edx
movl 8(%ebp),%ebx
leal (%ebx,%edx,4),%eax
pushl %eax
pushl %ebx
call DOT
addl $12,%esp
movl 12(%ebp),%ebx
addl %eax,(%ebx)
```

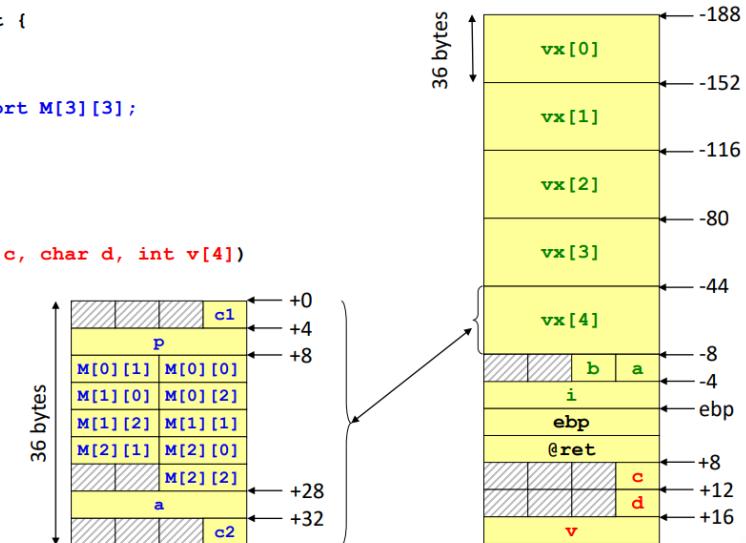
```
void PDOT(int M[10][10], int *p) {
    int i;
    *p = 0;
    for (i=0; i<10; i++)
        *p += DOT(&M[0][0],&M[i][0],10);
}
```



Ejemplos de Subrutinas y Structs

```
typedef struct {
    char c1;
    char *p;
    unsigned short M[3][3];
    int a;
    char c2;
} X;
```

```
int rut (char c, char d, int v[4])
{
    X vx[5];
    char a;
    char b;
    int i;
    ...
}
```



■ **Localidad Temporal:** si traemos un dato (o instrucción) de memoria, sería útil guardarlos “cerca” del procesador para que los futuros accesos sean más rápidos.

■ **Localidad Espacial:** si traemos un dato (o instrucción) de memoria, sería útil traer también los datos próximos y dejarlos “cerca” del procesador. Esto sólo tiene sentido si traer datos próximos sólo cuesta un poco más que traer un solo dato.

Tipos de Memoria de Semiconductores

■ **Memoria Estática** (SRAM, Static RAM). Cada celda de memoria equivale a 1 biestable (6-8 transistores). En comparación con las DRAM son **rápidas**, tienen un **alto consumo, pequeñas** (poca capacidad) y **caras**.

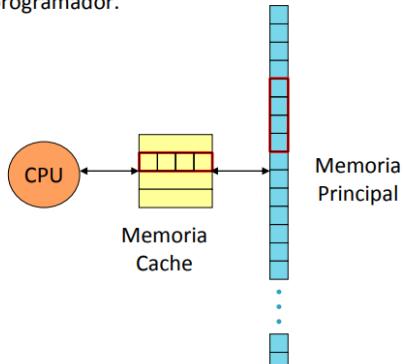
→ Memoria Cache

■ **Memoria Dinámica** (DRAM, Dynamic RAM). Cada celda se comporta como un condensador (1-1.x transistores). En comparación con las SRAM son **lentas**, tienen un **bajo consumo, grandes** (muchoa capacidad) y **baratas**. Problema del refresco.

→ Memoria Principal

Principios de Funcionamiento de las Memorias Cache

■ **Memoria Cache:** memoria **pequeña** y **rápida** que almacena una parte del contenido de una memoria más grande y lenta. La memoria cache se encargará de que la información que se almacene sea útil. Esta memoria es transparente al programador.



■ **Objetivo:**

- **Velocidad** de la memoria cache.
- **Capacidad** de la memoria principal.
- **Coste** de la memoria principal más un porcentaje razonable.

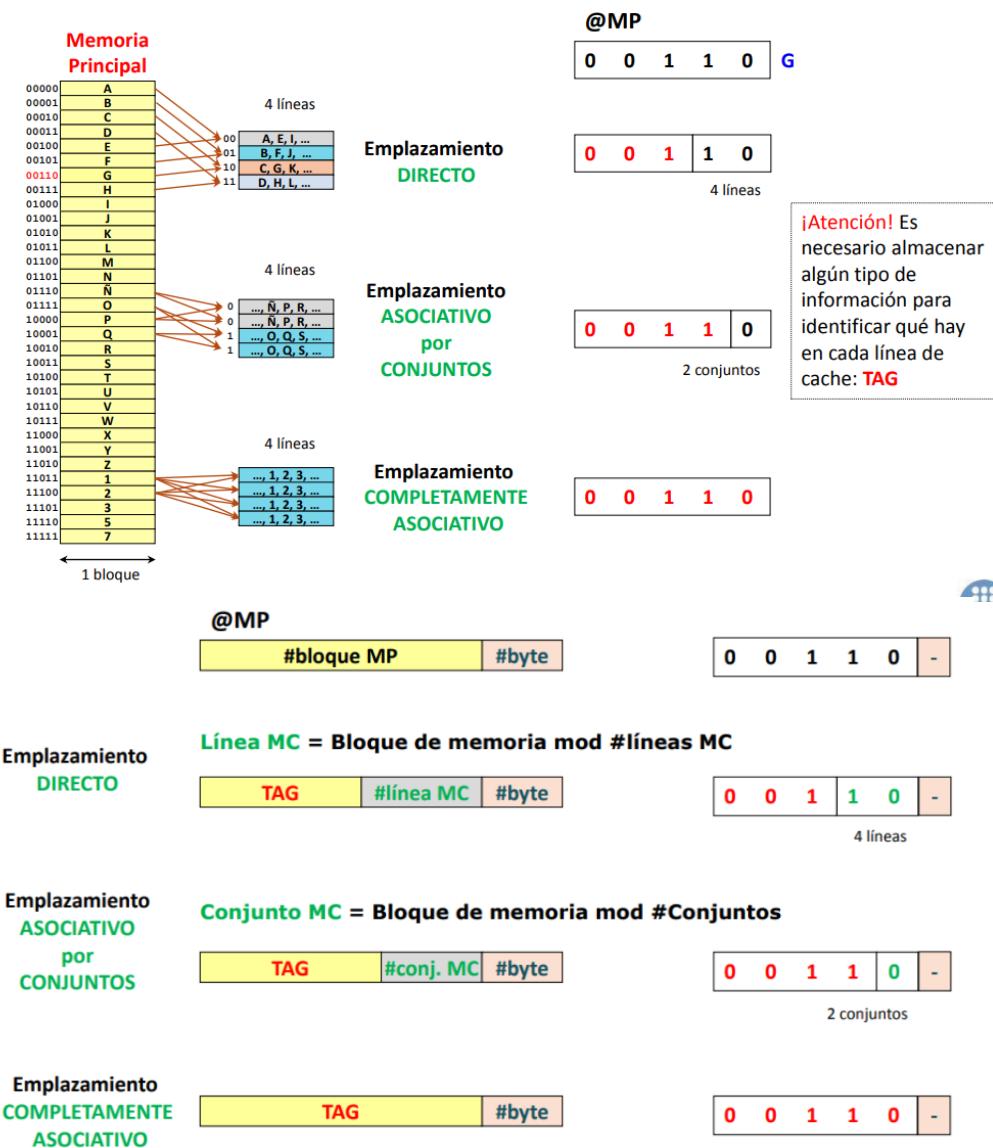
■ Esta solución es posible debido a la **localidad de los programas**. La cache retiene información recientemente usada e información próxima a la recientemente usada.

- **Conceptos:**
- Acierto / Fallo
 - Línea de cache / Bloque de memoria
 - Algoritmos emplazamiento
 - Algoritmos reemplazo
 - Políticas de Escritura
 - Evaluación

■ **En cualquier Memoria Cache hay que definir:**

- **Algoritmo de Emplazamiento:** determina en qué líneas de MC puede colocarse un bloque. Determina, también, dónde hay que buscar un dato.
- **Algoritmo de reemplazo:** determina qué línea se ha de eliminar de la cache para dejar espacio a un nuevo bloque.
- **Políticas de escritura:** determina cómo se hacen las escrituras. En cualquier caso, al final siempre se ha de escribir en MP.

Algoritmos de Emplazamiento



Evaluación Algoritmos emplazamiento

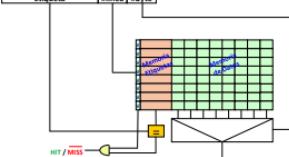
Tasa de fallos. Usando códigos con localidad espacial.

Líneas de 32B; cada elemento de un vector ocupa 4B; $v[i]$, $w[i]$ y $x[i]$ alineadas a 1GB (p.e.)

```
for (i=0; i<N; i++)
    f(v[i]);
```

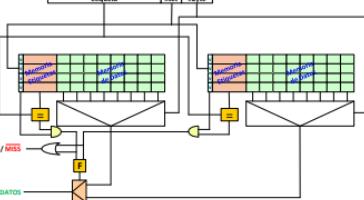
```
for (i=0; i<N; i++)
    g(v[i],w[i]);
```

```
for (i=0; i<N; i++)
    h(v[i],w[i],x[i]);
```



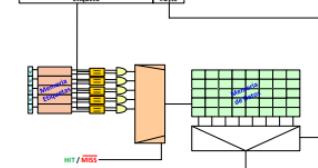
Cache Directa

$$\begin{aligned} m_A &= 0.125 \\ m_B &= 1 \\ m_C &= 1 \end{aligned}$$



Cache 2-asociativa

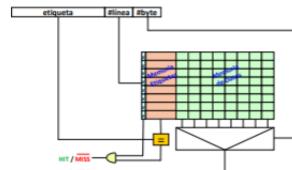
$$\begin{aligned} m_A &= 0.125 \\ m_B &= 0.125 \\ m_C &= 1 \end{aligned}$$



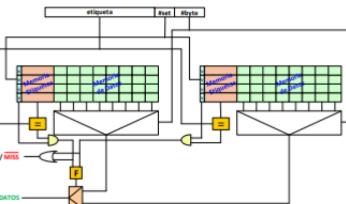
Cache Completamente Asociativa

$$\begin{aligned} m_A &= 0.125 \\ m_B &= 0.125 \\ m_C &= 0.125 \end{aligned}$$

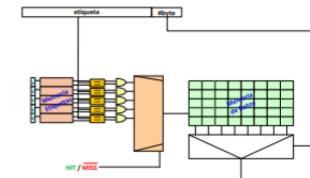
Tiempo de acceso, depende del camino crítico



Cache Directa



Cache 2-asociativa



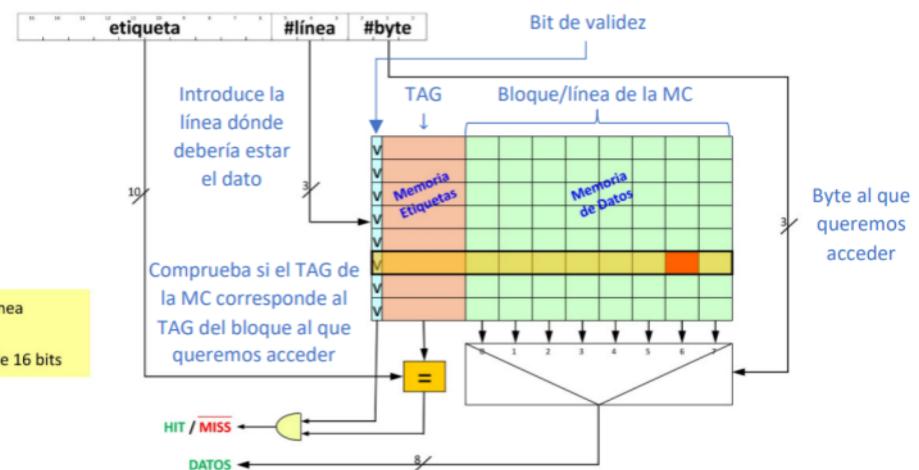
Cache Completamente Asociativa

Disminuye el TIEMPO de ACCESO

Aumenta la TASA de FALLOS*

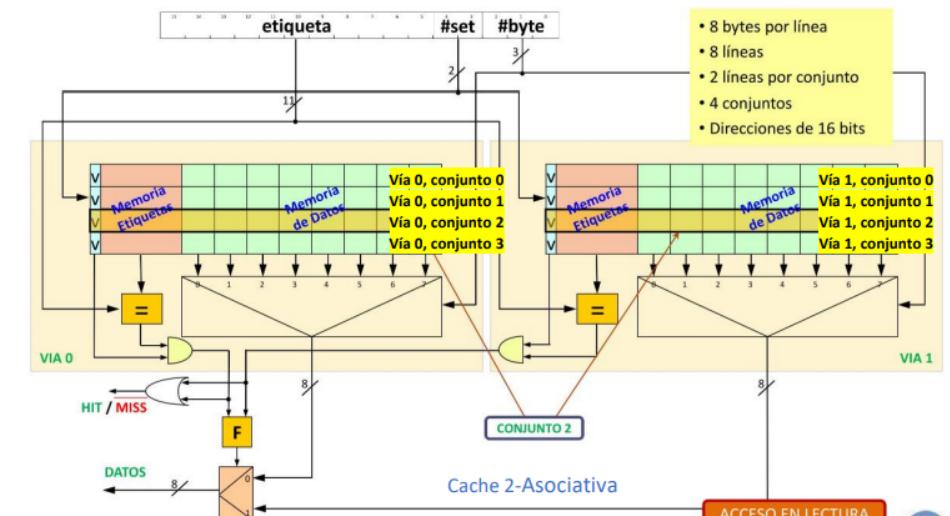
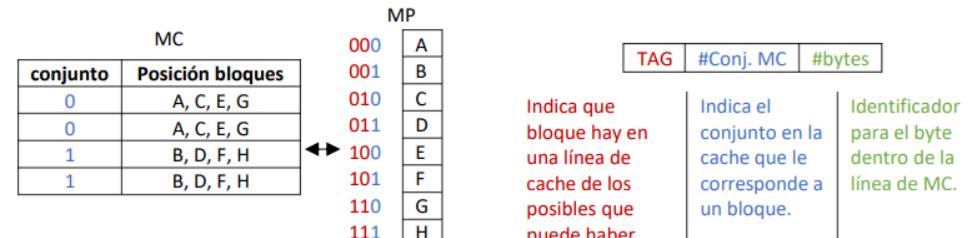
* En general

- Cache Directa: Acceso (Memoria datos y etiquetas), comparar etiqueta y validar línea.
- Cache Asociativa por conjuntos: Acceso (Memoria datos y etiquetas), comparar etiqueta, validar línea y seleccionar vía.
- Cache Completamente Asociativa: Acceso (Memoria etiquetas), comparar etiqueta, validar línea y Acceso (Memoria datos).

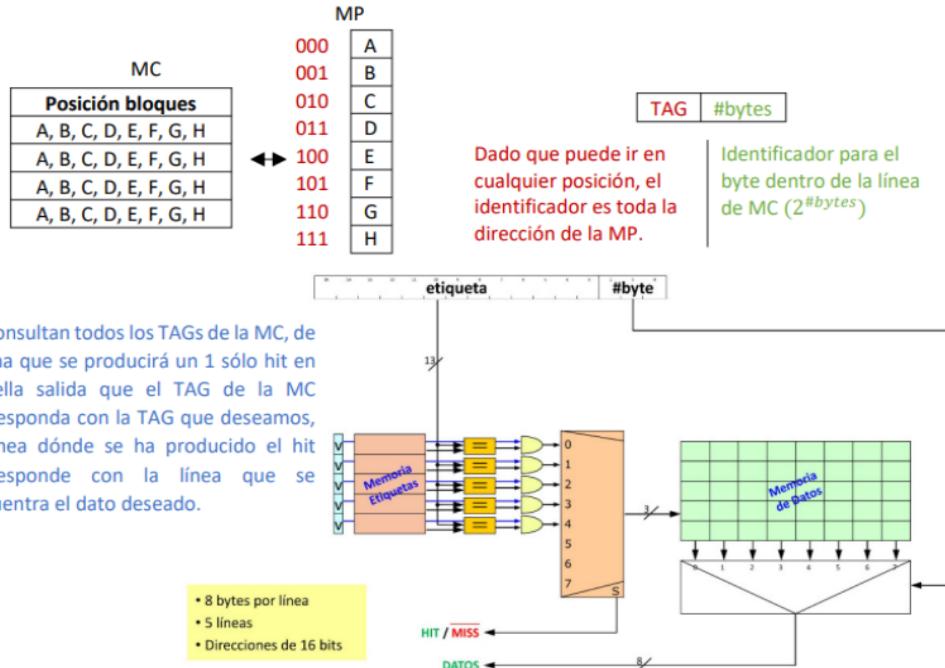


- Emplazamientos asociativos por conjuntos: la cache se divide en un número de conjuntos y dependiendo de los bits de menos peso, un bloque de la MP es asignado un conjunto.

Ejemplo de una cache de 4 líneas:



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).



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.

Políticas de Escritura

Premisa: las escrituras, finalmente, se han de hacer en Memoria Principal.

¿Cuándo se actualiza la Memoria Principal?:

■ WRITE THROUGH (escritura a través o escritura inmediata)

- Se actualiza simultáneamente EL DATO en la MC y la MP.
- El tiempo de servicio es el tiempo de acceso a MP.
- La MP siempre está actualizada.
- Se puede reducir el tiempo de escritura utilizando buffers.

■ COPY BACK (escritura diferida)

- En una escritura sólo se actualiza EL DATO en MC.
- Para cada línea se añade un bit de control (dirty bit) que indica si la línea ha sido modificada o no.
- Se actualiza el bloque en la MP cuando la línea (si ha sido modificada) ha de ser reemplazada.
- Las escrituras son rápidas (velocidad de MC).
- El tiempo de penalización en caso de fallo aumenta.
- Durante un tiempo existe una inconsistencia entre MP y MC.

¿Qué hacer en caso de fallo en escritura?

■ 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 MC. Esto obliga a realizar la escritura directamente en MP.

■ En una MC Directa no tiene sentido hablar de algoritmo de reemplazo.

Principales algoritmos de Reemplazo

Reemplazo Aleatorio

- Se selecciona aleatoriamente una línea de entre todas las candidatas a ser reemplazadas.
- Es un algoritmo muy sencillo de implementar.
- A pesar de su aparente falta de sentido, funciona bastante bien.

Reemplazo FIFO (First In First Out)

- De entre todas las líneas candidatas a ser reemplazadas, se selecciona la que lleva más tiempo en la cache.
- Es un algoritmo muy sencillo de implementar, sólo es necesario utilizar un contador con módulo.
- Tiene un comportamiento patológico no deseable porque no tiene en cuenta la utilización.

Reemplazo LRU (Least Recently Used)

- De entre todas las líneas candidatas a ser reemplazadas, se selecciona la que lleva más tiempo en la cache sin ser utilizada.
- Este algoritmo da buenos resultados. Teniendo en cuenta el comportamiento de los programas, parece la opción más lógica.
- Sin embargo, es muy costoso de implementar si el grado de asociatividad es alto. El coste de implementar este algoritmo es $n!$ (siendo n el grado de asociatividad).
- Normalmente se implementa un algoritmo PseudoLRU. Un algoritmo LRU ha de mantener información de en qué orden se ha accedido a todas las líneas de un conjunto. En un algoritmo pseudoLRU sólo se mantiene parte de esa información.

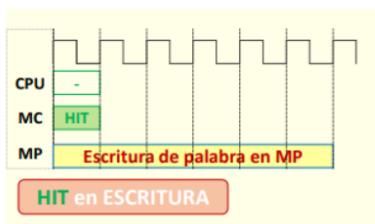
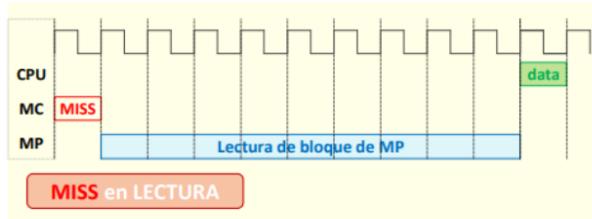
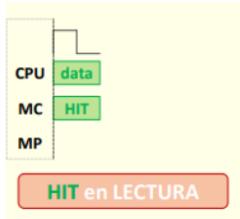
Operaciones a realizar en un acceso a cache

Modelo utilizado:

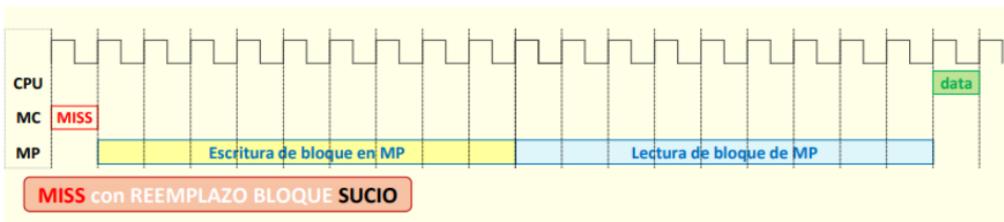
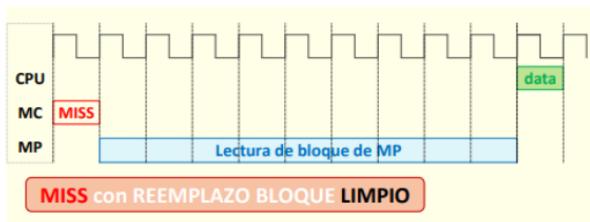
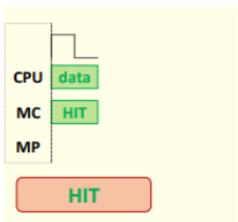
- Cache de datos
- Políticas de Escritura (2 casos):
 - Write Through + Write NO Allocate
 - Copy Back + Write Allocate
- Tamaño de línea: 32B
- Tiempo de servicio en caso de acierto: 1 ciclo
- Memoria Principal:
 - Lectura línea: 9 ciclos
 - Escritura línea: 9 ciclos
 - Escritura palabra: 6 ciclos



■ Write Through + Write NO Allocate



■ Copy Back + Write Allocate



- Write Through + Write NO Allocate

- Lectura $t_{lectura}$
- 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 $t_{escritura} = tasa_{escritura} * t_{escritura\ palabra\ MP}$

■ Medida de Rendimiento

• Tasa de Aciertos

$$h = \frac{\# \text{aciertos}}{\# \text{referencias}}$$

• Tasa de Fallos

$$m = \frac{\# \text{fallos}}{\# \text{referencias}} = 1 - h$$

• ¿De qué dependen?

- ✓ Del tamaño de Cache
- ✓ Del tamaño de Bloque
- ✓ De los algoritmos de Emplazamiento y Reemplazo
- ✓ Del Programa evaluado

■ Medida de Rendimiento

• Tiempo medio de acceso (coste de un acceso a memoria): Tma

• Componentes:

- ✓ Coste de un acceso en acierto: tsa
- ✓ Coste de un acceso en fallo: tsf = tsa + tpf
- ✓ Penalización de un fallo: tpf

• Tiempo medio de acceso a Memoria:

$$\begin{aligned} Tma &= h \cdot tsa + m \cdot tsf \\ &= tsa + m \cdot tpf \end{aligned}$$

- A menor asssociativitat, millor

- Com més gran sigui la caché, pitjor

■ Medida de Rendimiento

• Tiempo de ejecución de un programa:

$$Tejec = N \cdot CPI \cdot Tc$$

N: instrucciones ejecutadas

CPI: ciclos por instrucción en media

Tc: tiempo de ciclo

$$\checkmark CPI = CPI_{ideal} + CPI_{mem}$$

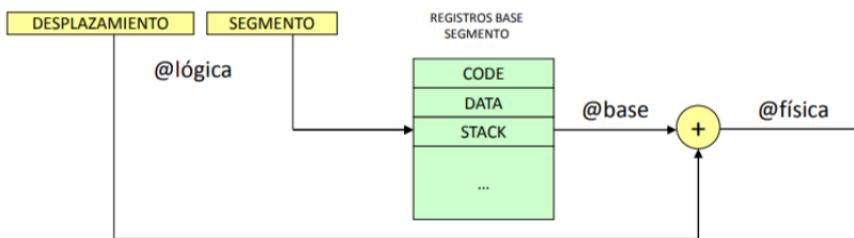
✓ CPI_{mem} son los ciclos que perdemos por tener una cache imperfecta ($m \neq 0$).

$$CPI_{mem} = nr \cdot (Tma - tsa)$$

$$CPI_{mem} = nr \cdot m \cdot tpf \quad (\text{caso particular de una MC de sólo lectura})$$

Segmentación

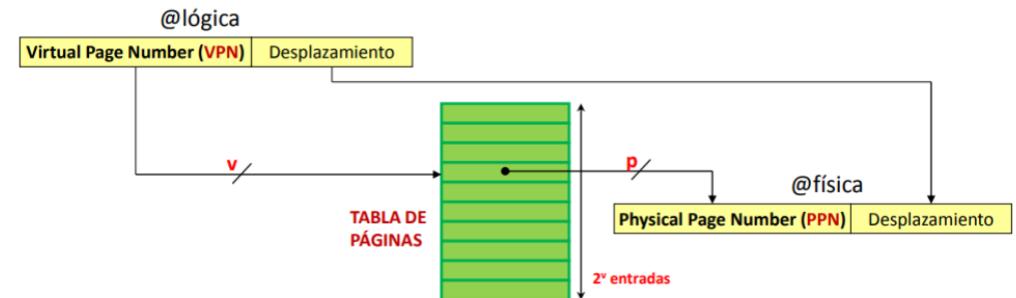
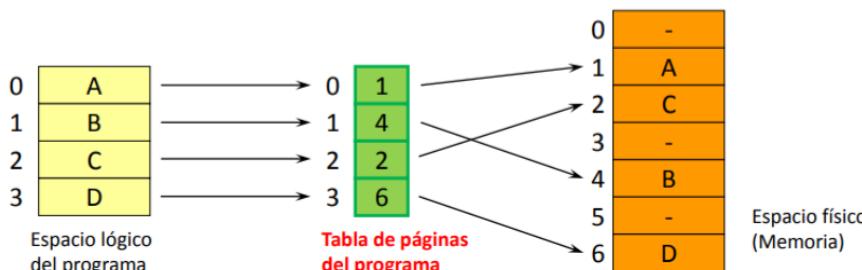
- El programa se descompone en segmentos: código, datos, pila, ...
- Cada segmento se identifica por su dirección inicial y tamaño.
- Los segmentos se almacenan de forma contigua en memoria y de forma disjunta entre segmentos.
- El mecanismo de traducción es bastante simple:



- Un cambio de contexto (usuario o programa) implica cambiar el contenido de los registros.
- Acceso lento: Acceso al banco de registros de segmentos y suma.
- Reubicación muy simple.
- Fragmentación de la memoria.
- Permite protección de los segmentos.

Paginación

- El espacio lógico se divide en bloques de tamaño fijo → PÁGINAS**
- Los sistemas actuales tienen páginas con tamaño entre 4 y 16 KB**
- El espacio físico (MP) se divide en marcos de tamaño una página (*frames, tramas*).**
- Los programas se trocean en páginas (están en disco)**
- Una página puede colocarse en CUALQUIER marco de página de MP (correspondencia completamente asociativa)**
- Las páginas se copian desde disco a MP cuando son referenciadas**
- Hace falta una estructura de datos para saber qué hay en cada marco de página → TABLA DE PÁGINAS.**



- Cálculo rápido de la dirección (no hay operaciones aritméticas).**
- Fragmentación (ficheros pequeños ocupan 1 página completa).**
- Reubicación muy simple.**
- Permite protección de páginas.**
- Página físicas y virtuales tienen el mismo tamaño.**
- VPN y PPN pueden tener longitud diferente.**
- En la mayoría de sistemas se cumple: $2^v > 2^p$.**

Paginación

La paginación consiste en que yo cojo la memoria principal y la divido en trozos iguales que se llaman "marcos de página". Entonces para cada página de un proceso le asigno un marco de página. Entonces, en vez de cargar todo el proceso, solamente cargo aquella parte del proceso que necesito.

La página 6 del proceso P2 corresponde a la página 4 de memoria principal.

(páginas virtuales) → corresponden a → (páginas físicas)

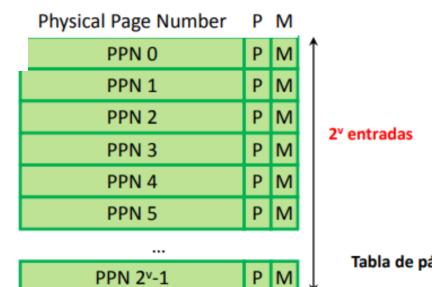
DISCO -- página --> RAM -- bloque --> CACHÉ -- dato --> CPU

La memoria virtual es completamente asociativa (+ LRU). El sistema operativo tiene que pensárselo muy bien antes de reemplazar una página.

PÁGINA	DESPLAZAMIENTO
El número de la página (este número se traduce para ir de MV a MF)	Los bits de desplazamiento de la página (se mantienen constantes en la traducción). Tiene tantos bits como el tamaño de la página (4KB → 12 bits)

El tamaño de la tabla de páginas es de 2^{52} filas (con páginas de 4KB).

- 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 ha sido modificada en MP).



Traducción de direcciones con TLB

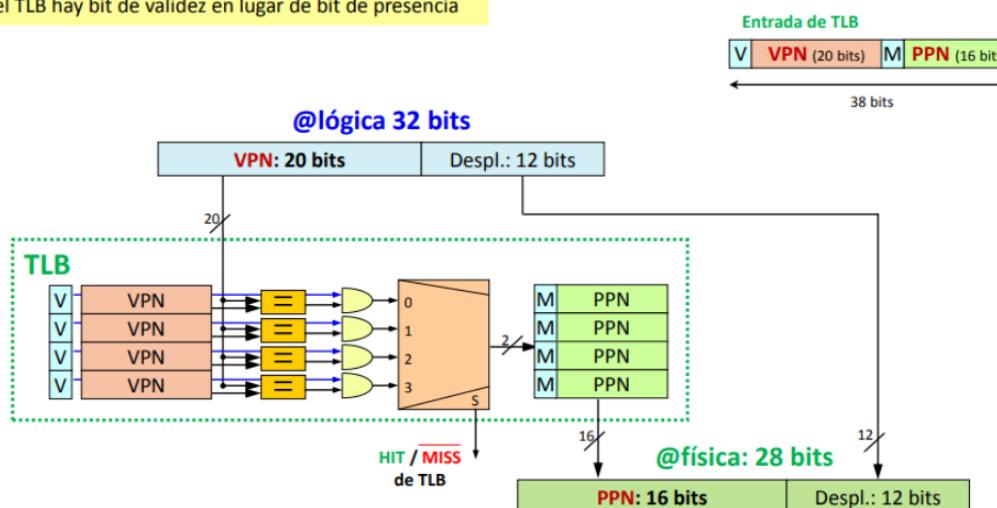
Translation Lookaside Buffer (TLB)

- Sirve para **acelerar** el proceso de traducción de direcciones
- Tiene una **estructura similar** (campos) a la Tabla de Páginas
- Sólo guarda **algunas** de las entradas de la TP
- Contiene más entradas de página que las páginas que caben en la cache L1 (contiene traducciones de datos residentes en L2 y en MP).
- Procesadores de mediados de los 90 tenían TLB de 128 entradas, 32-64 KB de cache L1 y páginas de 4-16KB

Características principales:

- Integrado en el mismo chip que en el procesador
- Pocas entradas (64-128) (1 entrada por página)
- Completamente asociativo
- Tasa de fallos muy baja
- Muy rápido (debido a que tiene pocas entradas de pocos bits)
- Algoritmo de reemplazo (LRU, PseudoLRU, FIFO, Random, ..)

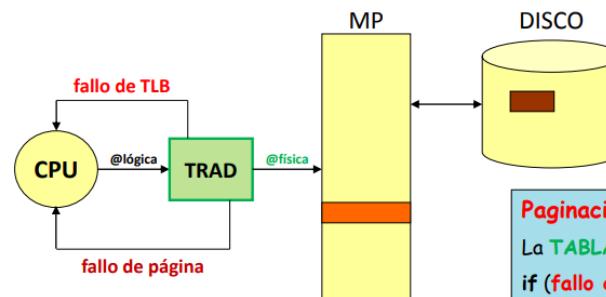
- 32 bits de dirección lógica
- 28 bits de dirección física
- Páginas de 4KB (2^{12} bytes)
- TLB de 4 entradas
- En el TLB hay bit de validez en lugar de bit de presencia



Memoria virtual

La Memoria Virtual permite:

- Ejecutar un programa con 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

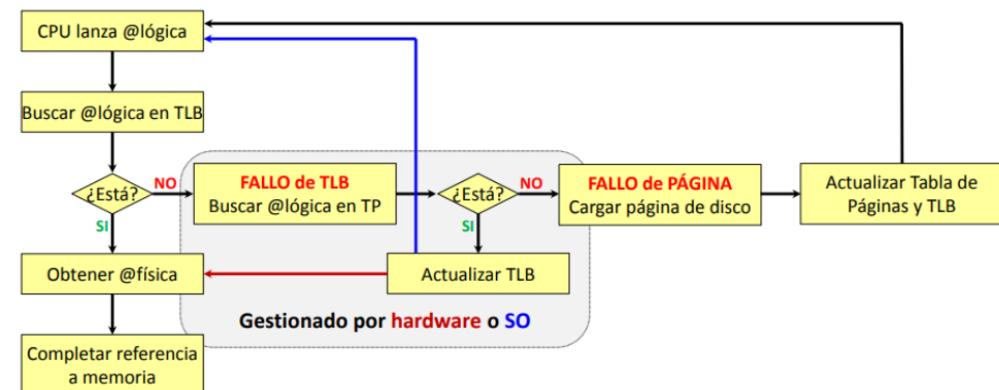


Paginación bajo demanda

```
La TABLA PÁGINAS tiene un bit de presencia
if (fallos de página){
    Reemplazar página: MP → DISCO
    Cargar la página solicitada: DISCO → MP
}
```

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

Paginación bajo demanda (con TLB)

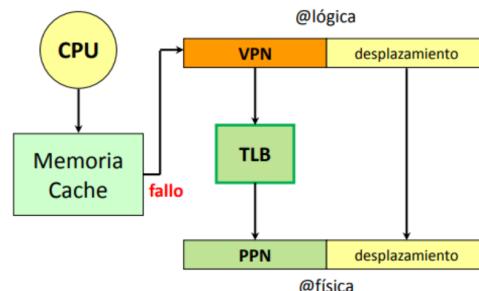


Fallo de TLB

- Se puede resolver mediante una excepción o incluso por hardware.
- Requiere un tiempo relativamente corto para ser resuelto si la página está en la Tabla de Páginas
- Típicamente, se resuelve en unos centenares de ciclos

Juntando Memoria Virtual y Memoria Cache Introducción

Traducción **después** de acceder a Memoria Cache



- Memoria Cache de direcciones lógicas
 - Se realiza traducción **SÓLO** en caso de fallo en MC
 - **Aumenta el coste de un fallo** de MC

Cualquier optimización a realizar tiene como objetivo final reducir el tiempo de ejecución:

$$T_{\text{exe}} = N \cdot CPI \cdot T_c$$

↓

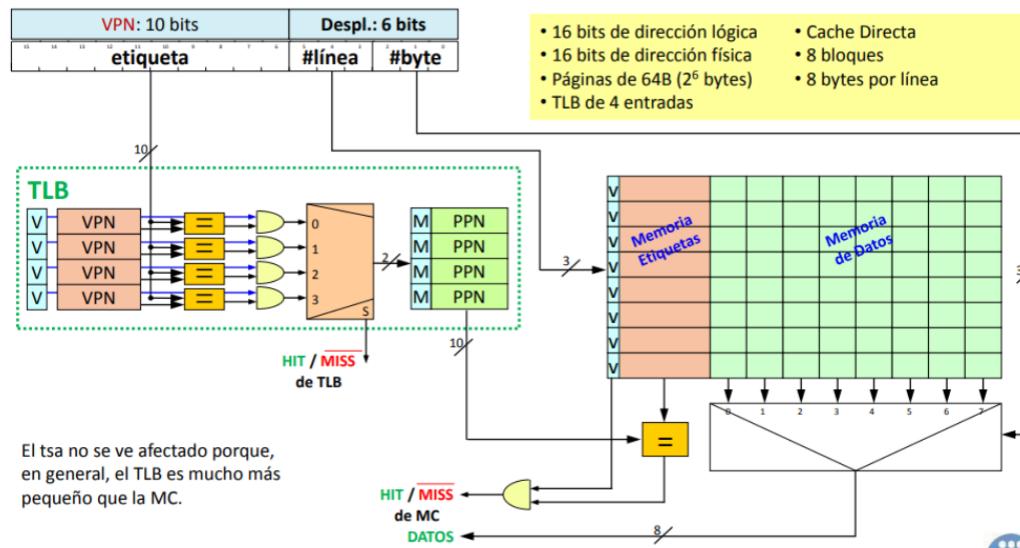
$$T_{\text{exe}} = N \cdot (CPI_{\text{id}} + CPI_{\text{mem}}) \cdot T_c$$

↓

$$T_{\text{exe}} = N \cdot (CPI_{\text{id}} + n_r \cdot m \cdot t_{\text{pf}} + CPI_{\text{WR}}) \cdot T_c$$

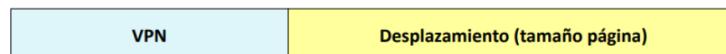
- Hay elementos que dependen del lenguaje máquina (N , n_r) en los cuales no influye la jerarquía de memoria.
 - La jerarquía de memoria puede influir en:
 - Tasa de fallos (objetivo $m \downarrow$)
 - Tiempo de penalización por fallo (objetivo $t_{pf} \downarrow$)
 - Coste de las escrituras (objetivo $CPI_{WR} \downarrow$)
 - Ancho de banda con memoria

Traducción en TLB y acceso a Memoria Cache simultáneos



El tsa no se ve afectado porque, en general, el TLB es mucho más pequeño que la MC.

- Se busca en la MC con la parte de la dirección que corresponde al desplazamiento (línea y byte de la línea)
 - La memoria de etiquetas contiene etiquetas **FÍSICAS**
 - Se traduce únicamente la **página LÓGICA** que corresponde a la etiqueta y se comprueba si la línea de la MC es el bloque buscado.
 - Restringe el tamaño de la Memoria Cache:
 - $\# \text{conjuntos} \cdot \text{tamaño línea} \leq \text{tamaño página}$



Clasificación de los fallos de cache (3C model)

Los fallos de cache pueden dividirse en tres categorías:

- **Carga (compulsory)**: se producen la primera vez que se accede a una posición de memoria.
 - **Capacidad**: todas las líneas que necesita un programa no caben en la Memoria Cache.
 - **Conflicto**: se producen cuando varios bloques se mapean en el mismo lugar de la MC (sólo en MC directas y asociativas por conjuntos)
 - [En los multiprocesadores aparecen los fallos de **Coherencia**]

Técnicas básicas para mejorar el rendimiento de la cache

■ Aumentar el tamaño de bloque ($m \downarrow$). Reduce los fallos de carga, pero puede ser contraproducente ($m \uparrow$).

■ Aumentar el tamaño de cache ($m \downarrow$). Reduce los fallos de capacidad (y conflicto), pero aumenta el tiempo de acceso a la cache ($tsa \uparrow$) y el consumo ($w \uparrow$).

■ Aumentar el grado de asociatividad ($m \downarrow$). Reduce los fallos de conflicto, pero aumenta el tiempo de acceso a la cache ($tsa \uparrow$).

■ Caches multinivel ($t_{pf} \downarrow$). L1 pequeña ($m \uparrow$ y $tsa \downarrow$) y L2 grande ($m \downarrow$ y $tsa \uparrow$).

■ Dar más prioridad a las lecturas que a las escrituras ($CPI_{WR} \downarrow$).

El coste de las escrituras se puede reducir (ocultar) utilizando buffers de escrituras.

■ Reducir el coste de un acierto en cache ($tsa \downarrow$): caches pequeñas y simples, predicción de vía y trace caches.

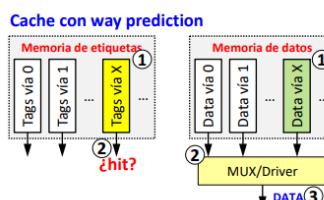
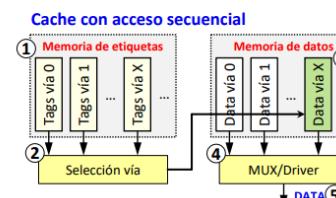
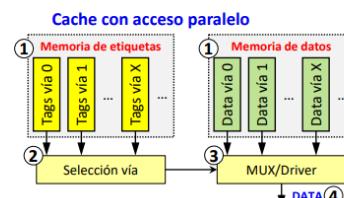
■ Aumentar el ancho de banda de cache ($BW \uparrow$): caches segmentadas, caches multi-banco y caches no bloqueantes.

■ Reducir el coste de los fallos ($t_{pf} \downarrow$): early restart y merging write buffers.

■ Reducir la tasa de fallos ($m \downarrow$): optimizaciones del compilador.

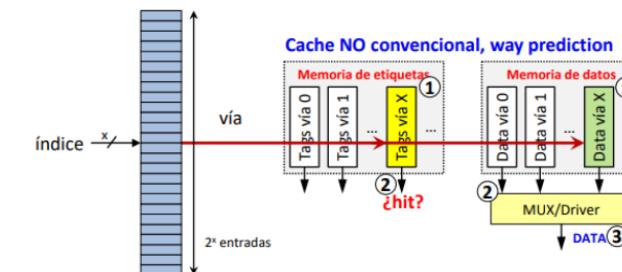
■ Reducir el coste de los fallos ($t_{pf} \downarrow$) y la tasa de fallos ($m \downarrow$) vía paralelismo: pre-búsqueda hardware y pre-búsqueda software.

Predicción de vía para reducir el tiempo de acceso



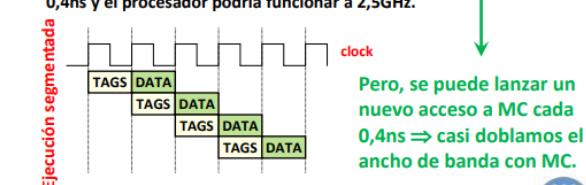
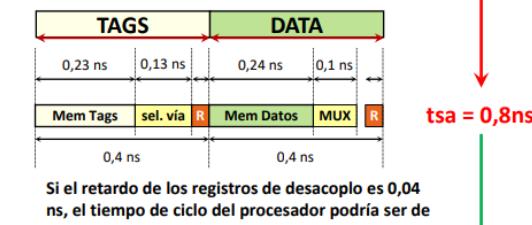
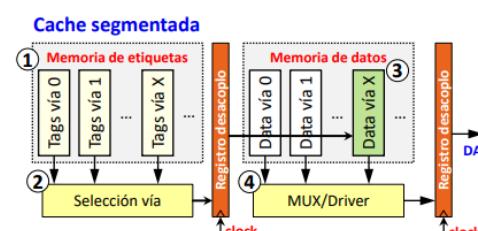
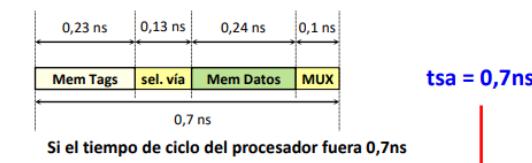
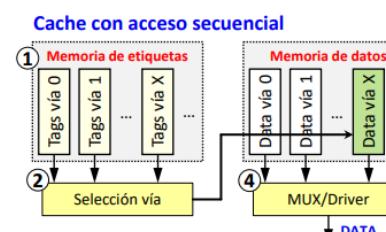
- En una cache con acceso en paralelo, el consumo es elevado. El tiempo de acceso viene determinado por el acceso a memoria y la selección de la vía.
- Una cache con acceso secuencial es lenta, pero reduce sustancialmente el consumo porque sólo accede a los datos de la vía seleccionada.
- En una cache con way prediction, el consumo se reduce porque sólo accedemos a la vía indicada por la predicción y reduce el tiempo de acceso en caso de acierto, porque no necesita la selección de vía.

■ ¿Cómo se realiza la predicción?

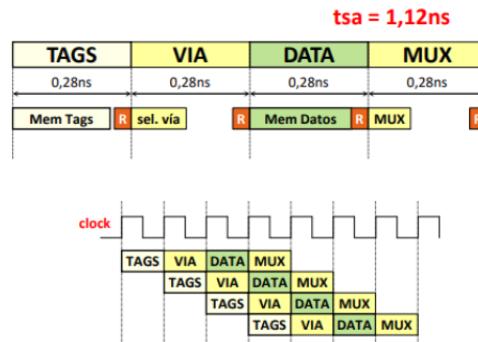
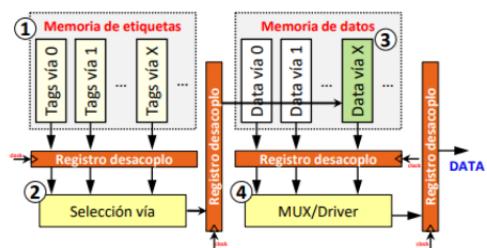


- El índice podría ser el PC de la instrucción en ejecución \Rightarrow La tabla sería demasiado grande.
- Podemos utilizar unos pocos bits del PC (como en una cache).
- La tasa de aciertos en la predicción depende (entre otras cosas) del tamaño de la tabla (2^x).
- Los programas tienen una «ejecución predecible» (localidad).
- La tabla de predicción se actualiza con el comportamiento de los accesos previos.

Cache segmentada



Se puede aumentar el grado de segmentación



MC (accesos de 4B)	tsa	Tc	f	Ancho banda máx.
NO segmentada	0,7ns	0,7ns	1,43GHz	5,71 GB/s
Segmentada 2 etapas	0,8ns	0,4ns	2,5GHz	10 GB/s $\approx \times 2$
Segmentada 4 etapas	1,12ns	0,28ns	3,57GHz	14,29 GB/s $\approx \times 3$

La latencia de 1 acceso individual aumenta ($tsa=1,12\text{ns}$), pero el ancho de banda aumenta porque se pueden realizar 4 accesos en paralelo (se puede iniciar un acceso a MC cada 0,28ns).

ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$R2 \leftarrow R0+57$																				
$R3 \leftarrow M[R2]$																				
Fallo de cache																				
$R5 \leftarrow R7-R4$																				
$R4 \leftarrow R5<<2$																				
$R6 \leftarrow M[R4]$																				
$R7 \leftarrow R5+R4$																				
$R9 \leftarrow R6+R7$																				
$R3 \leftarrow R3*R1$																				
$R4 \leftarrow R3+17$																				
$M[R2] \leftarrow R4$																				
$R2 \leftarrow R2+8$																				

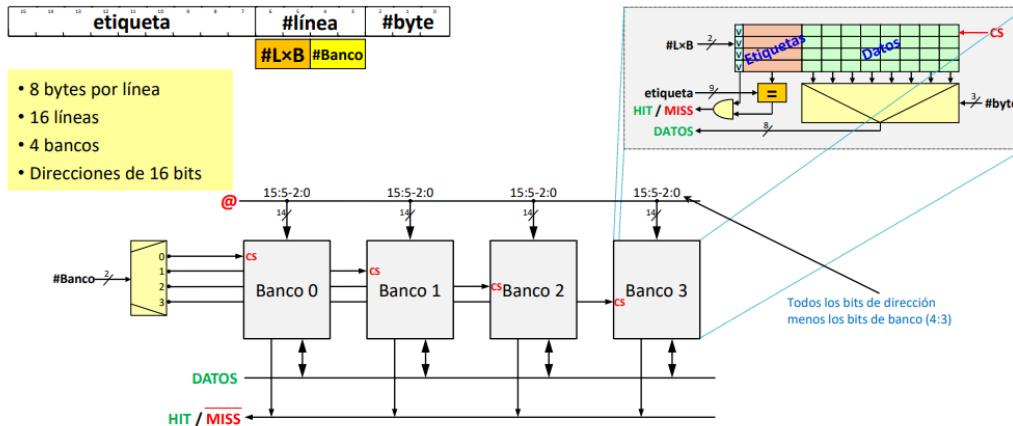
- Una Non Blocking Cache usa **MSHRs** (Miss Status Handler Register) para gestionar los fallos pendientes.
- El número de MSHRs condiciona el número de fallos que puede soportar la MC sin detener el procesador.
- La Idea original de los MSHRs es que el compilador/programador separe lo suficiente los accesos a memoria ($R3 \leftarrow M[R2]$) de su uso ($R3 \leftarrow R3*R1$) para que, en caso de fallo de cache, el procesador no se detenga
- El segundo fallo ($R8 \leftarrow M[R4]$) no da problemas porque R8 no se usa próximamente. Pero si se usase...

ciclo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$R2 \leftarrow R0+57$																				
$R3 \leftarrow M[R2]$																				
Fallo de cache																				
$R5 \leftarrow R7-R4$																				
$R4 \leftarrow R5<<2$																				
$R6 \leftarrow M[R4]$																				
$R7 \leftarrow R5+R4$																				
$R9 \leftarrow R6+R7$																				
$R3 \leftarrow R3*R1$																				
$R4 \leftarrow R3+17$																				
$M[R2] \leftarrow R4$																				
$R2 \leftarrow R2+8$																				

- ... sería preciso mantener información del registro destino de 2 fallos de cache
- Los MSHRs mantienen información de cual es el registro destino del load para controlar cuando se ha de bloquear el procesador.

- En una **Non Blocking Cache**, cuando se produce un fallo de cache el procesador continúa la ejecución de instrucciones y sólo se detiene cuando necesita el dato que ha provocado el fallo de cache.

Cache organizada en Bancos

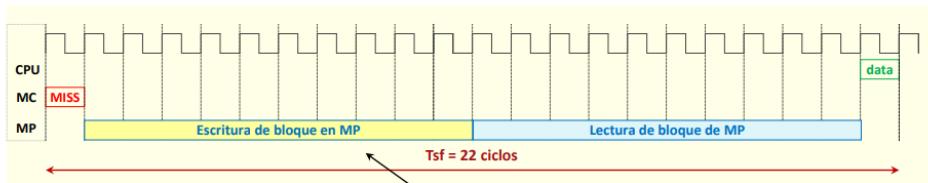


■ Esta organización permite:

- Reducir consumo, sólo es necesario activar el banco al que se accede.
- Realizar accesos concurrentes.

Reducir la penalización por fallo

■ Punto de partida

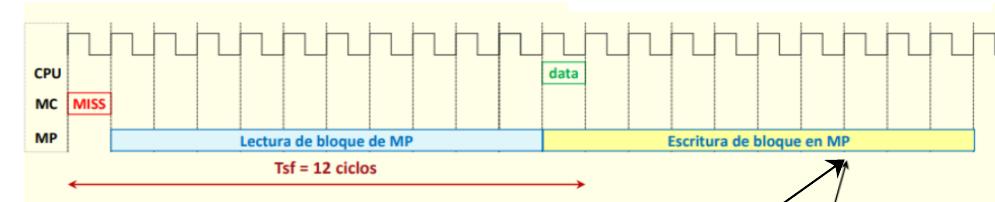


- Copy Back + Write Allocate
- 32 bytes por bloque
- **MISS** con REEMPLAZO BLOQUE SUCIO
- Fallo en el byte 16
- Lectura de bloque: 10 ciclos
- Escritura de bloque: 10 ciclos

La escritura sólo es necesaria si el bloque a reemplazar ha sido modificado.

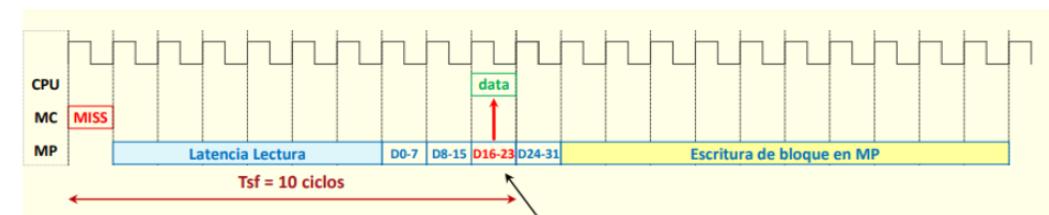
■ Actualizar MP después de leer línea

Se deja el bloque en un buffer y se escribe en MP cuando se pueda.



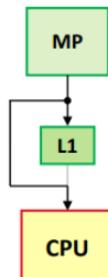
■ Continuación Anticipada (*Early Restart*):

en cuanto llega el dato que ha provocado el fallo, se envía al procesador

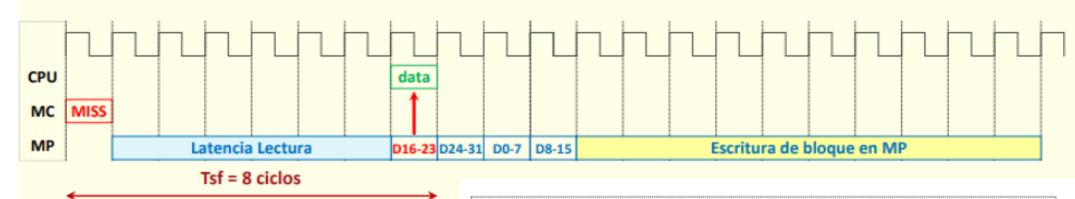


- Copy Back + Write Allocate
- 32 bytes por bloque
- **MISS** con REEMPLAZO BLOQUE SUCIO
- Fallo en el byte 16
- Lectura de bloque:
 - Latencia: 6 ciclos
 - Transferencia: 8B por ciclo
- Escritura de bloque: 10 ciclos

Cuando llega a la MC el dato que ha provocado el fallo, se envía simultáneamente a la CPU.



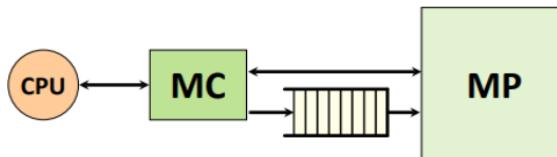
■ Transferencia en desorden + Continuación Anticipada: se envía en primer lugar el dato que ha provocado el fallo.



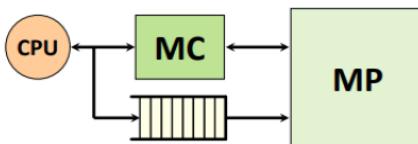
En todos los casos, y para que estos mecanismos sean efectivos, cuando se pasa el dato al procesador y mientras se acaba de servir el fallo, la MC ha de ser capaz de recibir nuevos accesos (Cache no bloqueante).

Buffers de Escritura

- Si la cache es COPY BACK para reducir la penalización en caso de fallo hay que dar prioridad a leer el boque que contiene el dato que provoca el fallo a la escritura en MP del bloque reemplazado.



- Si la cache es WRITE THROUGH el coste de una escritura es el coste de escribir en Memoria Principal (no es aceptable).



- Solución: poner un buffer de n entradas entre la CPU y MP.

- Las escrituras se almacenan en el buffer
- El acceso al buffer es rápido (equivale a acceder a MC, 1 ciclo).
- Las escrituras se realizan cuando el bus entre MC y MP no está ocupado.
- Unas pocas entradas son suficientes.
- Cuando se realiza un acceso a MC, también hay que consultar en el buffer las datos pendientes de escribir

- Situación común

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

- Cuando realizamos escrituras con localidad espacial, el buffer se llenará muy rápido y será poco eficiente.

Solución: Merge Buffers

@	V	data
100	1	M[100]
104	1	M[104]
108	1	M[108]
112	1	M[112]
116	1	M[116]
120	1	M[120]
-	0	-
-	0	-

Buffer convencional

@	V	data	V	data	V	data	V	data
100	1	M[100]	1	M[104]	1	M[108]	1	M[112]
116	1	M[116]	1	M[120]	0	-	0	-
-	0	-	0	-	0	-	0	-

Merge Buffer

Se puede aprovechar el hecho de que escribir un bloque de memoria tiene prácticamente el mismo coste que escribir una palabra.

Escritura dato en MC

Cuando se realiza una lectura, se puede leer el dato en paralelo con la comprobación de si es un acierto o un fallo.

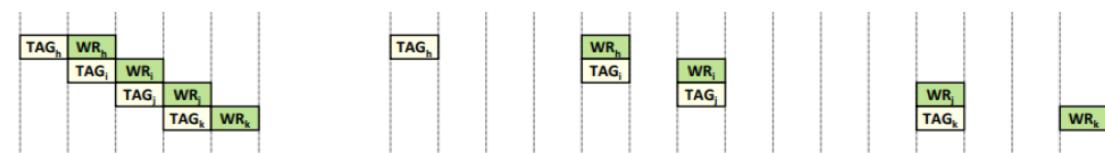
Una escritura en MC no se puede realizar hasta que sabemos que es un acierto.

Si una lectura necesita 1 ciclo, una escritura necesita 2 ciclos:

- **CICLO 1:** Leer datos, leer etiqueta y comprobar acierto, escribir en el fill buffer la línea completa modificando el dato que indica la escritura (D5).
- **CICLO 2:** En caso de acierto, escribir el contenido del fill buffer en la línea correspondiente.

Solución: ESCRITURAS SEGMENTADAS

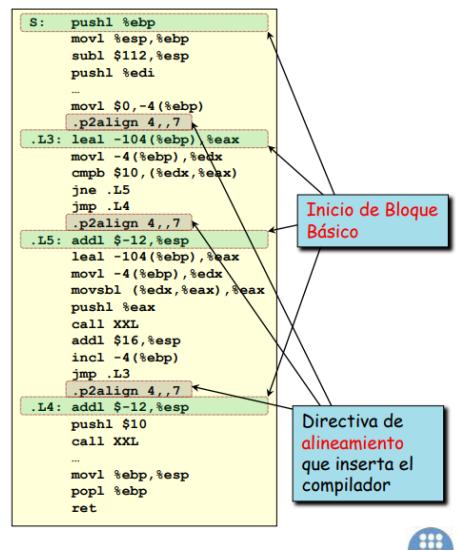
- En el primer ciclo se comprueba si es acierto o fallo y, en caso de acierto, se deja el dato a escribir en un registro intermedio.
- En la siguiente escritura (mientras se comprueba si es acierto o fallo) se realiza la escritura anterior.
- Una escritura individual sigue tardando 2 ciclos, pero desde el punto de vista del procesador sólo cuesta un ciclo.



Optimizaciones de código para reducir tasa fallos

■ Reordenación de código

- Alineando los puntos de entrada de los bloques básicos con el inicio de la línea de cache, aumenta la probabilidad de acierto en cache para código secuencial.
- Si el compilador considera que un salto condicional se comportará la mayoría de las veces en el mismo sentido, puede organizar el código para que, en ese caso, el código se ejecute en secuencia.



■ Reordenación de los datos

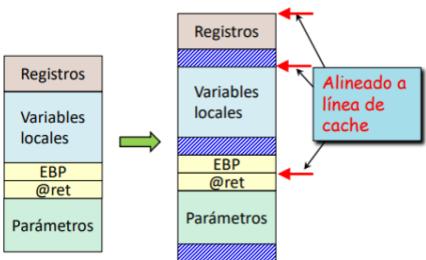
- Los datos pueden ubicarse en memoria para evitar conflictos.

// Código Original
int v[1024],w[1024];
int A[1024][1024];

// Código Transformado
int v[1024],d[p],w[1024];
int A[1024][1024+k];

El objetivo es evitar que las direcciones iniciales de los vectores y las filas de la matriz se mapeen en el mismo bloque (o conjunto) de cache.

- Las diferentes partes del bloque de activación de una subrutina pueden alinearse con el inicio de la línea de cache para aprovechar la localidad espacial.



■ Loop Fusion (Fusión de bucles)

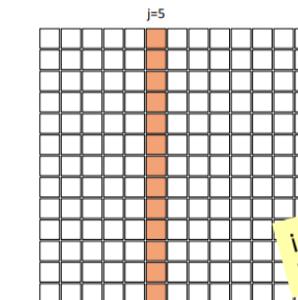
// Código Original
for(i=0; i < N; i++)
 a[i] = b[i] * c[i];

for(i=0; i < N; i++)
 d[i] = a[i] * c[i];

// Código Transformado
for(i=0; i < N; i++){
 a[i] = b[i] * c[i];
 d[i] = a[i] * c[i];
}

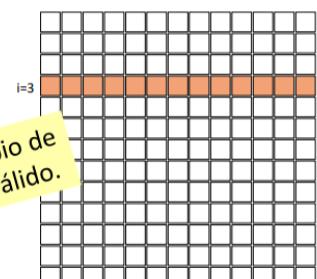
■ Loop Interchange (Intercambio de bucles)

// Código Original
for (j=0; j<100; j++)
 for (i=0; i<100; i++)
 x[i][j] = 2 * x[i][j];



Los accesos consecutivos a memoria de datos están separados por 100-4 bytes.

// Código Transformado
for (i=0; i<100; i++)
 for (j=0; j<100; j++)
 x[i][j] = 2 * x[i][j];



¡Atención! El intercambio de bucles no siempre es válido.

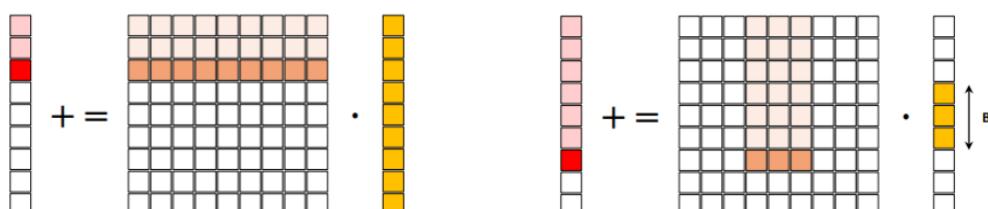
Los accesos consecutivos a memoria de datos están separados por 4 bytes.
¡Se aprovecha la localidad espacial!

■ Blocking (MxV)

// Código Original
int A[N][N],x[N],y[N];
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
 x[i] += A[i][j]*y[j]

// Código Transformado
for (jj=0; jj<N; jj=jj+B)
 for (i=0; i<N; i++)
 for (j=jj; j<min(N,jj+B); j++)
 x[i] += A[i][j]*y[j]

- La matriz se recorre por filas, se aprovecha la localidad espacial.
- Si el vector "y" es más grande que la cache se producirán fallos de capacidad.



No accedido

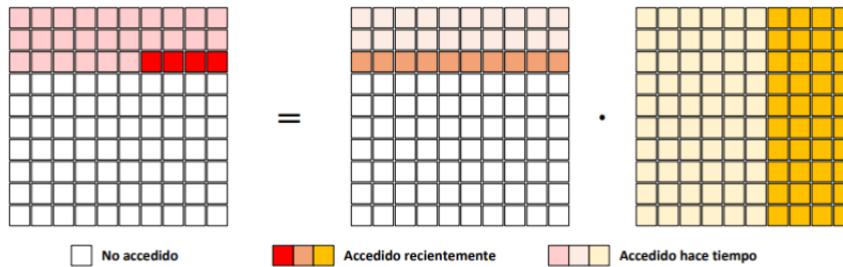
Accedido recientemente

Accedido hace tiempo

■ Blocking (MxM)

```
// Código Original
for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
        r=0;
        for (k=0; k<N; k++) {
            r=r+y[i][k]*z[k][j];
        }
        x[i][j]=r;
    }
```

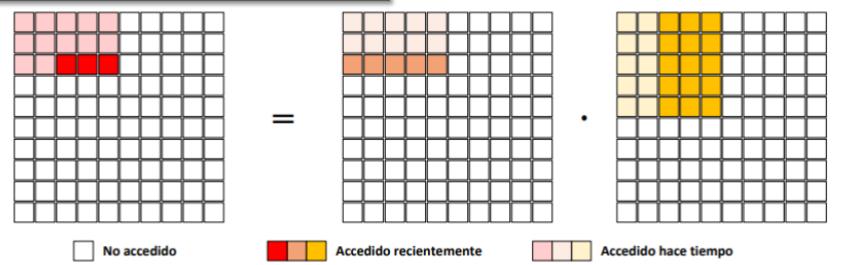
- Hay accesos a fila y columna, el intercambio de bucles no es suficiente.
- Para una i determinada se accede a $y[i] [*] N$ veces y se recorre completamente la matriz z .
- Suponiendo que no hay fallos por conflicto necesitamos que en la cache quepan N^2+N+1 elementos para que no se produzcan fallos de capacidad.
- Si la cache es menor se producirán fallos de capacidad. Como máximo serían $2N^3+N^2$ accesos a memoria para $2N^3$ operaciones.



■ Blocking (MxM)

```
// Código Transformado
for (jj=0; jj<N; jj=jj+B)
    for (kk=0; kk<N; kk=kk+B)
        for (i=0; i<N; i++)
            for (j=jj; j<min(jj+B,N); j++) {
                r=0;
                for (k=kk; k<min(kk+B,N); k++)
                    r= r + y[i][k]*z[k][j];
                x[i][j]=r;
            }
```

- El objetivo del blocking es que los datos a los que se accede en los bucles más internos quepan en la cache.
- El parámetro B (*blocking factor*) se ajusta al tamaño de la cache disponible.
- Suponiendo que no hay fallos por conflicto necesitamos que en la cache quepan $N^2/B+N/B+1$ elementos para que no se produzcan fallos de capacidad.



Prefetch

- **Objetivo:** Reducir los fallos de CARGA

- **Estrategia:** Especulamos con la localidad y traemos a MC aquella información que creemos que será utilizada en un futuro cercano, antes de que sea solicitada.

- Los accesos a instrucciones son más sencillos de predecir que los accesos a datos.
- La información ha de llegar a tiempo, ni muy pronto ni demasiado tarde.

- **El problema del prefetch es que podemos traer información NO ÚTIL a la cache, ocupando espacio de MC y ancho de banda entre MC y MP.**

Prefetch Hardware

Prefetch Hardware de instrucciones en el Alpha 21064

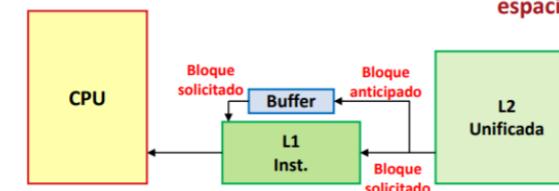
- Cuando se produce un fallo de cache, se traen el bloque solicitado (**bloque i**) y el siguiente en secuencia (**bloque i+1**), si no está ya en la cache.

- El bloque solicitado se deja en la MC y el siguiente en el buffer de prefetch.

- Si hay fallo en cache, pero acierto en el buffer, se sirve el fallo desde el buffer, se pasa el bloque $i+1$ a la MC y se trae al buffer el siguiente (**bloque i+2**)

- El buffer puede tener varias entradas

Esquema útil cuando hay **localidad espacial** (instrucciones).



Prefetch Hardware de datos

Prefetch en fallo

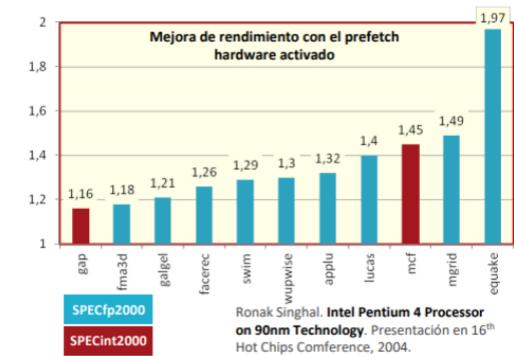
- Prefetch del bloque $i+1$ cuando se falla en el bloque i

Esquema OBL (One Block Lookahead)

- Prefetch del bloque $i+1$ cuando se accede al bloque i

Prefetch con stride

- Si se observa una secuencia de accesos a bloque B , $B+N$, $B+2*N$, entonces se hace prefetch de $B+3*N$...



Prefetch Software (datos)

- Se utilizan instrucciones especiales (las insertan el compilador o bien el programador de LM) para traer los datos de forma anticipada.
- En general, el prefetch puede introducir tráfico (entre MP y MC) innecesario.
Con el prefetch software tenemos más control (se puede reducir el tráfico inútil).
- Se pierde tiempo ejecutando instrucciones de prefetch (¿inútiles?).
- Ejemplo:

```
// Código Original
for(i=0; i < N; i++)
    sum = sum + b[i]*c[i];
```



```
// Código Transformado
for(i=0; i < N; i++){
    prefetch(&b[i+P]);
    prefetch(&c[i+P]);
    sum = sum + b[i]*c[i];
}
```

- El valor de P no es trivial de calcular:
 - Si hacemos prefetch muy cercano ($P \downarrow$), es posible que el dato no llegue a tiempo.
 - Si hacemos prefetch muy lejano ($P \uparrow$), tendremos polución en la cache.
- Normalmente, no ejecutaremos el prefetch en todas las iteraciones,
lo haremos p.e. 1 de cada 4 (dependiendo del tamaño de línea de cache).
- Si suponemos que los vectores están alineados y que caben 4 elementos por línea,
la transformación podría ser:

```
// Código Original
for(i=0; i < N; i++)
    sum = sum + b[i]*c[i];
```



```
// Código Transformado
for(i=0; i < N; i=i+4){
    prefetch(&b[i+4*P]);
    prefetch(&c[i+4*P]);
    sum = sum + b[i]*c[i];
    sum = sum + b[i+1]*c[i+1];
    sum = sum + b[i+2]*c[i+2];
    sum = sum + b[i+3]*c[i+3];
}
// P = 1, 2, 3, ...
```