Tiempo de ejecución: tiempo transcurrido entre el inicio y el final de una tarea.

Productividad: número de tareas completadas por una cantidad de tiempo.

Tiempo de ejecución (resultado en s):

$$t_{ej} = n_{ciclos} * t_c = (I_A * CPI_A + I_B * CPI_B + \cdots) * \frac{1}{f_{clock}(en\ Hz)}$$

$$t_{ej} = I * CPI * t_c$$

CPI media del programa (resultado en cpi):

$$CPI = \frac{n_{cilcos}}{I} = \frac{I_A * CPI_A + I_B * CPI_B + \cdots}{I_A + I_B + \cdots}$$

Ganancia, speed-up o rendimiento relativo:

$$ganancia = s = \frac{t_{original}}{t_{mejorado}}$$

Mejorar S_x veces una parte P_x entonces la ganancia global es:

$$ganancia = s = \frac{1}{\frac{P_x}{S_x} + (1 - P_x)} \rightarrow P_x = \frac{t_{ahora}}{t_{original}}$$

Mejorar al máximo ($S_x \rightarrow \infty$):

$$\frac{1}{1 - P_x}$$

Consumo de energía:

$$P = C * V^2 * f_{commut} \rightarrow f_{commut} = f_{clock} * \alpha$$

- P → Potencia $(W, watt)$, $(J/s, joule\ por\ segundo)$.
- C → Capacitancia agregada de todos los transistores que conmuten $(F, farads)$.
- V → Voltaje $(V, voltios)$.
- α → Factor actividad.

$$E = P * t$$

- E → Energía $(J, joule)$.
- T → Tiempo $(s, segundo)$

Memoria

La ordenación utilizada en esta asignatura es Little Endian, es decir, cuando se coloca el dato en la memoria se introduce el valor de menos peso arriba y el de más peso abajo y se utilizará una memoria de 32 bits.

Variables

El tamaño de las variables (en C) son:

Tipo de variable		Memoria		Tamaño	Alineación		
`char`	→	`.byte`	→	1 byte	La dirección inicial de la	→	Múltiple de 1
`short`	→	`.half`	→	2 bytes	variable ha de ser	→	Múltiple de 2
`int`	→	`.word`	→	4 bytes	múltiple del tamaño de la	→	Múltiple de 4
`long long`	→	`.dword`	→	8 bytes	variable	→	Múltiple de 8

Nota: 1 byte son 8 bits, por lo tanto, al representarlo en hexadecimal son dos dígitos.

La declaración y alineación de variables almacenadas en memoria se les reserva el tamaño de la variable indiferentemente de si el valor no se extiende por todo el espacio reservado.

Operandos en memoria → `lw` y `sw` → Deben seguir cumpliendo la alineación de direcciones.
`lwu`, `lhu` → para valores unsigned.

Representación de enteros en base 2

Complemento a 2 (Ca2):

- Para pasar de positivo a negativo o al revés invertir los unos por ceros y viceversa y después sumar 1.
- El Rango es $x \in [-2^{n-1}, 2^{n-1} - 1]$.

Complemento a 1 (Ca1):

- Para pasar de positivo a negativo o al revés cambiar los unos por los ceros y viceversa.
- El 0 se representa de dos maneras $0000 \ldots / 1111 \ldots$

Símbolo y magnitud

- El número se representa de manera "normal" menos el bit de más peso que es 0 si es positivo y 1 si es negativo
- El 0 se representa de dos maneras $0000 \ldots / 1000 \ldots$

Bits	Naturales	Ca2	Ca1	Símbolo y magnitud
000	0	0	0	0
001	1	1	1	1
010	2	2	2	2
011	3	3	3	3
100	4	-4	-3	0
101	5	-3	-2	-1
110	6	-2	-1	-2
111	7	-1	0	-3

Exceso $2^{n-1} - 1$

$$valor + (2^{n-1} - 1) = exceso$$

Vectores → Es una agrupación unidimensional de elementos del mismo tipo identificados por un índice $[0, n-1]$.

Ejemplo: `int vec[5] = {0, 1, 2, 3, 4}` → `vec: .word 0, 1, 2, 3, 4`

`.space n` → reserva n espacios en la memoria.

Para calcular la dirección del i-ésimo elemento: `&vec[i] = &vec[0] + i * T`

Ejemplo:

```
int vec[100];
main() {
   int x; //$t1
   x = vec[3];
}
```

```
la $t0, vec
lw $t1, 12($t0)  #3*4 = 12
      o
la $t0, vec + 12
lw $t1, 0($t0)
```

String → Cadena de caracteres de tamaño variable, en C con el centinela '\0'

```
char cadena[20] = "Una frase";
char cadena[20] = {'U','n','a',' ','f','r','a','s','e','\0'};

cadena: .ascii "Una frase"   #9 caracteres
.space 11   #El centinela i 10 ceros
Cadena: .asciiz "Una frase"   #10 (incluye centinela)
.space 10
```

Puntero → Es una variable que contiene una dirección de memoria. Por lo tanto, en MIPS ocupa 32 bits. Si el puntero p contiene la dirección de la variable v decimos también que apunta a v.

- El operador (&): En C este operador delante de una variable devuelve la dirección de la variable
- El operador (*): En C este operador delante de un puntero devuelve el valor de la variable a la que apunta (la dirección contenida en) el puntero.

Ejemplos:

```
char *p;
void g() {
   char tmp;  //$t0
   tmb = *p;
}
```

```
g:…
  la $t2, p   #&p -> dirección del puntero
  lw $t3, 0($t2)  #p -> dirección a la que apunta el puntero
  lb $t0, 0($t3)  #*p -> variable que contiene el puntero
  …
```

```
char *p;
void g() {
   *p;= 3;
}
```

```
g: …
   la $t2, pglob  #&p -> dirección del puntero
   lw $t3, 0($t2) #p -> dirección a la que apunta el puntero
   li $t4, 3
   sb $t4, 0($t3) #*p = 3 -> la variable que contiene el puntero
   …
```

Nota: para sumar a `p` un valor deberemos multiplicar el valor que queremos sumar por el tamaño de la variable.

Relación entre punteros y vectores → En C un vector, es en realidad, un puntero que apunta al primer elemento.

Desplazamiento de bits

- `sll rd, rt imm5` → Desplazamiento lógico (Se desplazan ceros) a la izquierda.
- `slr rd, rt, imm5` → Desplazamineto lógico (Se desplazan ceros) a la izquierda.
- `sra rd, rt, imm5` → Desplazamiento aritmético (Se desplazan unos) a la derecha.

El operador << se traduce como `sll` o `sllv`.

El operador >> se traduce como `sra` o `srav`.

- `sllv rd, rt, rs` → Realiza `sll` pero con los 5 bits de menos peso de `rs`.
- `srlv rd, rt, rs` → Realiza `srl` pero con los 5 bits de menos peso de `rs`.
- `srav rd, rt, rs` → Realiza `sra` pero con los 5 bits de menos peso de `rs`.

Desplazar hacia la izquierda equivale a multiplicar un entero/natural por 2^{imm5}.

Desplazar hacia la derecha equivale a dividir un entero/natural por 2^{imm5} (no se puede utilizar un desplazador hacia la derecha para dividir números negativos).

Operaciones lógicas bit a bit

- `and rd, rs, rt` o `andi rd, rs, rt` → Sirve para seleccionar bits poniendo a 1 los que queremos seleccionar y a 0 los que no. **(para hacer $rd\%(rt+1)$)**
- `or rd, rs, rt` o `or rd, rs, rt` → escribe unos en los bits que queramos.
- `xor rd, rs, rt` o `xor rd, rs, rt` → complementa bits **(sirve para elevar)**
- `nor rd, rs, rt` → Invierte los bits.

Comparaciones y operaciones booleanas

Pone `rd` a 1 si `rs < rt` o `rs < imm16` si no lo pone a 0:

- `slt rd, rs, rt → rd = rs < rt` Para enteros.
- `sltu rd, rs, rt → rd = rs < rt` Para naturales.
- `slti rd, rs, imm16 → rd = rs < SignExt(imm16)` Para enteros.
- `sltiu rd, rs, imm6 → rd = rs < SignExt(imm16)` Para naturales.

- `rc = (ra == 0); → sltiu rc, ra, 1`
- `rc = (ra != 0); → sltu rc, $zero, ra`
- `rc = ra <= rb → rc <= rb → rc != (ra > rb) →`
  `→ slt rc, rb, ra`
  `→ sltiu rc, rc, 1`

Saltos:

- `beq $t1, $t2, etiq` → Salta a etiq si `$t1 == $t2`.
- `bne $t1, $t2, etiq` → Salta a etiq si `$t1 != $t2`.
- `blt $t1, $t2, etiq` → Salta a etiq si `$t1 < $t2`.
- `bgt $t1, $t2, etiq` → Salta a etiq si `$t1 > $t2`.
- `ble $t1, $t2, etiq` → Salta a etiq si `$t1 <= $t2`.
- `ble $t1, $t2, etiq` → Salta a etiq si `$t1 >= $t2`.
- `b etiq` → Salta incondicionalmente.

Subrutinas → Subprograma con diferentes parámetros que permite devolver un resultado, y permite ser invocada des de múltiples puntos del programa.

```
main:
…
   jal suma
ret:
…
suma:
   …
   jr $ra
```

Regla 1: Paso de parámetros y resultados:

- Parámetros: `$a0, $a1, $a2, $a3 (floats a $f12 i $f14).`
- En orden.
- Resultados: `$v0 ($f0 los floats).`
- Extenderemos símbolo o ceros si el tipo < 32 bits.

Regla 2: Variables locales

- Libre elección: `$t0…$t9, $s0…$s7, $v0…$v1.`
- *Floats* simple precisión a `$f0…$f31.`
- Las excepciones irán a la pila:
    - Vectores y matrices.
    - Si la variable v aparece con &v.
    - I nos quedamos sin registros.

  Se puede usar `$a0` y `$v0` para almacenar una variable local.

La pila (*stack*) y el bloque de activación (*stack frame*)

- Subrutinas: hay que guardar en memoria y liberarla detrás.
- Estructura LIFO (*last in first out*).
- Stack pointer `$sp = 0x7FFFFFFC`.
- No se utilizará *PUSH/POP*. Se reservará el espacio al principio y liberará al final.

Regla 3: Bloque de activación (en el sp)

- Variables en orden de cómo están en el código.
- Alineadas como si estuvieran en el `.data`. (respetando la alineación de los tipos de variable).
- El espacio reservado ha de ser múltiple de 4.

Regla 4: Preservar el contexto

- Una subrutina ha de preservar los valores originales de `$s0…$s7`. (Se guardan en registros seguros aquellos datos que se usaran después de la llamada y que su valor a sido generado antes de la llamada).
- Protegeremos los valores que se necesiten en estos registros.
- La pila debe quedar tal y como estaba.
- Todo se hará al principio y al final de la subrutina.

Multiplicar 2 números enteros de n y m bits da un resultado potencial de n + m bits, excepto cuando n o m es 1, ya que no produce *carry*.

- `mult rs, rt` → multiplica 2 registros `rs` y `rt` (de 32 bits) y deja el resultado (de 64 bits) en los registros `$hi` y `$lo` (los 2 de 32 bits).
- `mflo rd` → carga `$lo` en `rd`.
- `mfhi rd` → carga `$hi` en `rd`.

Ejemplos:

```
int mat[NF][NC]
int mit[2][3] = {{-1,0,2},{1,2,3}}
```

```
.data
mat: .space NF * NC * 4
mit: .word -1,0,2,1,2,3
```

```
int mat [NF] [NC] ;
  void func () {
  int i , j , k ;
  . . .
  k = mat [ i ] [ j ] ;
}
```

```
la $t3, mat
li $t4, NC
mult $t4, $t0 # . . . = i*NC
mflo $t4
addu $t4, $t4, $t1 # $t4 = i*NC + j
sll $t4, $t4, 2 # $t4 = (i*NC +
j)*4
addu $t3, $t3, $t4 # $t3 = mat +
(i*NC + j)*4
lw $t2, 0($t3) # k = mat [i] [j]
```

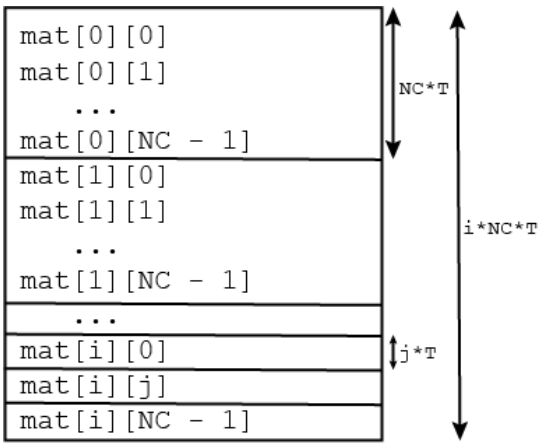
Para acceder a cualquier dirección de la matriz: `&mat[i][j] = mat + (i*NC + j)*T`.

Acceso secuencial → Cuando en vez de calcular cada vez la dirección a partir de la dirección inicial, se hace sumando un *stride* (constante entre los 2 elementos) a la dirección anterior. Se puede usar si las direcciones de cualesquiera 2 elementos consecutivos están separadas por una distancia constante.

Ejemplo:

```
void clear1 (int array[] , int
nelem) {
  int i;
  for (i = 0; i < nelem; i++)
  array [i] = 0;
}
```

```
clear1:
  move $t0, $zero #i =0
loop1:
  bge $t0, $a1, end1
  sll $t1, $t0, 2
  addu $t2, $a0, $t1
  sw $zero, 0($t2)
  addiu $t0, $t0, 1 #i++
  b loop1
end1:
```



Suma/Resta → Se produce *overflow* cuando el resultado exacto no pertenece al rango $[-2^{n-1}, 2^{n-1}-1]$ (números enteros en Ca2) i, por lo tanto, el resultado no es correcto.

- Si los signos NO son iguales el resultado siempre se podrá representar.
- Si los signos SÍ son iguales hay que mirar el resultado, si el bit de más peso es igual a la extensión de signo el resultado es correcto, en caso contrario no.

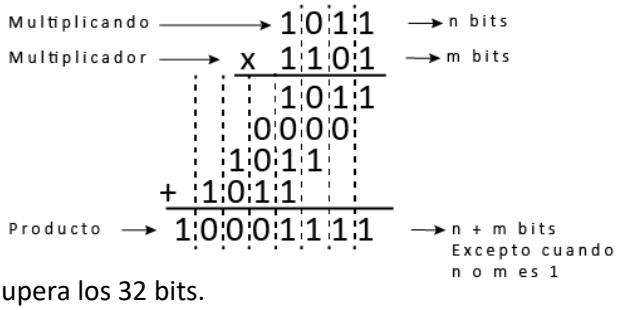
- `add, addi, sub` → causan excepción en caso de *overflow*.
- `addu, addiu, subu` → no causa excepción en caso de *overflow*.

Multiplicación

Multiplicación de naturales:

Multiplicación de enteros:

1- Cambiar signos a positivos
2- Multiplicar como natural
3- Cambiar el signo del resultado



Se produce *overflow* cuando el resultado supera los 32 bits.

Multiplicación *Hardware:*

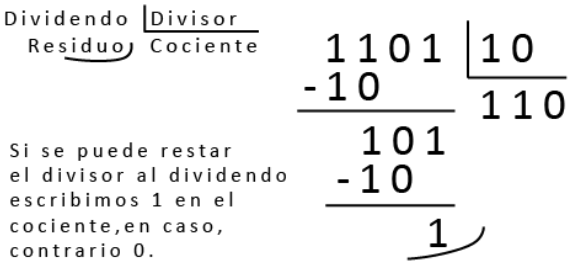
IT	MD (<<)	MR (>>)	P (+)
0	0000 XXXX	YYYY	0000 0000
…			Si el bit de menos peso de la iteración anterior (MR) es 1 se suma, si no, no se hace nada

División

Divisón de naturales:

División de enteros:

1- Cambiar signo
2- Dividir como naturales
3- Cambiar el signo del resultado



Solo existe overflow cuando dividimos $-2^{31}/-1 = 2^{31}$ → Esta fura de rango

Divisón Hardware:

IT	(divisor) D (>>)	(divdendo) R (-)	Q (<<)
0	XXXX 0000	0000 YYYY	0000
…		Si se puede restar (D < R) se resta la iteración posterior	Si se puede restar el divisor al dividendo escribimos 1, si no, 0.

Coma fija → De los 32 bits, se le asigna el bit de más peso al signo, unos bits a los enteros y otros bits a los fraccionarios

***Undeflow*:** Cuando el resultado de una operación es un valor absoluto más pequeño que el que la computadora realmente puede representar

Coma flotante (IEEE - 754)

	1	8	23
Precisión simple (32 bits):	Signo	Exponente	mantisa

	1	11	52
Precisión doble (64 bits):	Signo	Exponente	mantisa

Codificar:

1- Pasar a binario.
2- Normalizar (Escribir el valor en binario con la coma y signo).
3- Mover los bits.
4- Calcular el exceso del exponente $(x + 127)$.
5- Codificar.

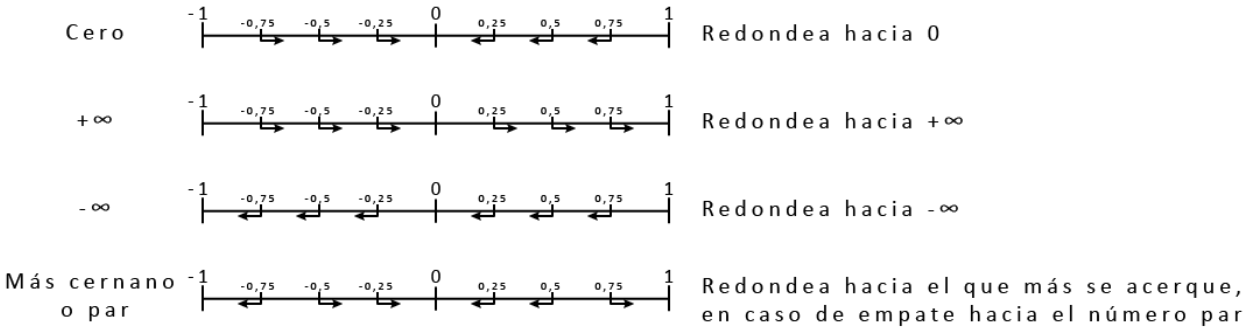
Descodificar

1- Pasar a binario.
2- Mirar el exceso $(x - 127)$.
3- Normalizar (pasar de normal a notación científica).
4- Mover los bits.
5- Pasar a decimal.

Calcular el error:

$$Error = |Valor\ real - valor\ representado|$$ (con infinitos decimales)

Redondear:



Codificaciones especiales:

Cero	$(+0) \rightarrow 0x0000\ 0000$ $(-0) \rightarrow 0x8000\ 0000$	Exponente y mantisa 0	Se pierde la posibilidad de usar el exponente -127
Infinito	$(+\infty) \rightarrow 0x7F80\ 0000$ $(-\infty) \rightarrow 0xFF80\ 0000$	Exponente 1 y mantisa 0	Se pierde la posibilidad de utilizar el exponente 128
Denormales	Exponente 0 y mantisa $\neq 0$		
NaN	Exponente 1 y mantisa $\neq 0$		

Rango / Precisión → Se obtiene una mayor precisión cuando el número esta más cerca del 0.

Rango de representación en formato normalizado en simple precisión → $[-126, 127]$

Denormales:

- Son números más cercanos al 0 que los números normalizados.
- No hay un bit oculto y el exponente es -126 (aunque lo codifiquemos como 0000 0000, que seria el -127 en exceso).
- El denormal más pequeño es $2^{-23} * 2^{-126} = 2^{-149}$.

Suma/Resta:

Para sumar y restar $x * b^n + y * b^n$ deben de tener los mismos exponentes $m = n \rightarrow (x + y) \cdot b^n$

1- Alinear exponentes (el de menos exponente alinearlo al de más).
2- Sumar/Restar mantisas. Si en la suma los 2 operandos tienen signos diferentes se hará una resta (el más pequeño al más grande, en valor absoluto).
3- Normalizar, si hace falta.
4- Redondear al más cercano o al par.
5- Ajustar el signo del resultado.
6- Codificar.

`add.s $f0, $f2, $f4` (fs pares porque al utilizar `add.d` guardan el resultado a `$f0 y $f1`).

Multiplicación:

$x * b^n * y * b^m = x * y * b^{n+m}$ No hace falta alinear.

1- Multiplicar mantisas (La coma se pone sumando el número de decimales del multiplicando y el multiplicador).
2- Sumar exponentes.
3- Normalizar, si hace falta.
4- Redondear al más cercano o par.
5- Ajustar el signo del resultado.
6- Codificar

```
mfc1 $f1, $t2 #Copia a $t2 el registro $f2
mtc1 $t2, $f2 #Copiar a $f2 el registro $t2
```

Con *float* la suma no es asociativa.

Terminología:

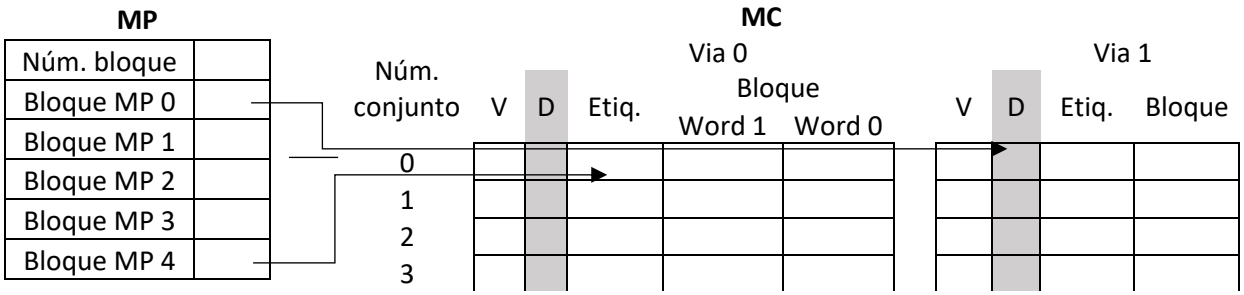
- **Fallada (miss):** Cuando la CPU intenta encontrar el dato en la cache (ya sea porque V = 0 o las etiquetas no coinciden) y este no está, entonces copiará de MP (memoria principal) a MC (memoria cache) en el bloque al que pertenece el dato.
- **Reemplazamiento:** Si en el sitio de la MC al que corresponde el bloque ya está ocupado por otro bloque se reemplazará (Si es escritura retardada D = 1).
- **Acierto (hit):** Si el dato está en la MC (cuando V = 1 y las etiquetas coinciden) se produce un acierto.
- **num_referencies:** Número de referencias en la memoria.
- **num_encerts:** Referencias que se han resuelto con aciertos en la cache.
- **Tasa de aciertos:** $h = \frac{num\_aciertos}{num\_referencias}$
- **Tasa de fallada:** $m = \frac{num\_falladas}{num\_referencias} = 1 - h$

Jerarquía de memoria:

Organización de la memoria en bloques:

$$número\ de\ bloque\ MP = \frac{A\ (número\ de\ bloque)}{TAMBLOC\ (Tamaño\ del\ bloque)}$$

Cache de correspondencia directa:



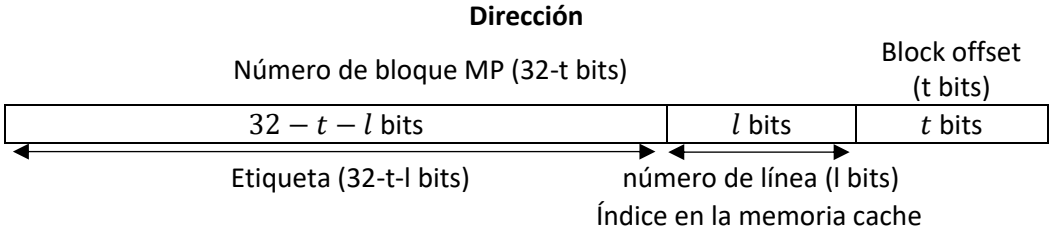
- Un cache de correspondencia directa consta de un número de líneas/bloques determinado.
- En cada línea puede haber un bloque de memoria principal.
- La línea de la MC dónde se ubican los MP se determina mediante

$$número\ de\ línea\ MC = número\ de\ bloque\ MP\ mod\ NUM\_LINEAS$$

- El bit V es el bit de validación sirve para saber si un bloque ya está ocupado (1) o no (0).
- La etiqueta sirve para poder distinguir unos bloques de otros en la memoria cache
- La etiqueta D (solo se usa con escritura retardada): indica si se ha modificado el bloque (1) o no (0)
- Cuando hay vías en la MC los bloques se ocupan en función de la que este vacío, no de la dirección, cuando todas están llenas se reemplaza mediante LRU (*Least Recently Used*).

$$Capacidad\ de\ la\ MC = palabras\_por\_bloque * tamaño\_palabra\_en\_bytes * num\_bloques$$

$$(\boldsymbol{TAMBLOC = 2^t}) \rightarrow ejemplo: 2\ palabras\ de\ 32\ bits = 64\ bits\ en\ el\ bloque = \frac{64}{8}\ bytes$$
$$= 8\ bytes = 2^3\ bytes \rightarrow t = 3\ bits$$
$$(\boldsymbol{NUM\_CONJUNTOS = 2^l}) \rightarrow ejemplo: 512\ bloques = 2^9\ núm.líneas \rightarrow l = 9\ bits$$



Gestión de escrituras

Escritura inmediata sin asignación:

- Lectura con fallada: Se remplaza el contenido del bloque.
- Lectura con acierto: No hace nada
- Escritura con acierto: Se remplaza el contenido en la MC y en la MP a la vez.
- Escritura con fallada: Se modifica solo la MP.

Escritura retardada con asignación:

- Lectura con fallada de un bloque no modificado: Se remplaza el contenido del bloque.
- Lectura con fallada de un bloque modificado: Hay que copiar el bloque modificado a la MP y luego reemplazar el contenido.
- Lectura con acierto: No hace nada.
- Escritura con acierto: Se modifica solo en la MC.
- Escritura con fallada de un bloque no modificado: Se remplaza el contenido del bloque.
- Escritura con fallada de un bloque modificado: Hay que copiar el bloque modificado a la MP y luego reemplazar el contenido.

Diseño de memoria:

Posibilidad 1: MC, MP y los buses de amplitud de una palabra.

- Tiempo que tarda el procesador en enviar una dirección = 1 ciclo.
- Tiempo que tarda el MP en leer **una palabra** = 15 ciclos (habrá que hacerlo 4 veces).
- Tiempo que tarda en enviar **una palabra** por el bus = 1 ciclo (habrá que hacerlo 4 veces).
- Magnitud del bloque: 4 palabras.

Tiempo necesario para copiar un bloque MP a MC: $t_p = 1 + 4 * 15 + 4 * 1 = 65\ ciclos$.

El número de bytes transferidos por ciclo serian: $4 * 4/65 = 0{,}25\ bytes/ciclo$.

Posibilidad 2: MC y MP de mayor amplitud, por ejemplo, que sean capaces de leer y escribir 2 palabras a la vez

- Tiempo que tarda el procesador en enviar una dirección = 1 ciclo.
- Tiempo que tarda el MP en leer **dos palabras** = 15 ciclos (habrá que hacerlo 2 veces).
- Tiempo que tarda en enviar **dos palabras** por el bus = 1 ciclo (habrá que hacerlo 4 veces).

Tiempo necesario para copiar un bloque MP a MC: $t_p = 1 + 2 * 15 + 2 * 1 = 33\ ciclos$.

El número de bytes transferidos por ciclo serian: $4 * 4/33 = 0{,}48\ bytes/ciclo$.

Posibilidad 2: Memoria entrelazada: b bancos de memoria, de amplitud p de palabras cada una, capaces de trabajar simultáneamente. La salida de menor amplitud.

- Tiempo que tarda el procesador en enviar una dirección = 1 ciclo.
- Tiempo que tarda el MP en leer **cuatro palabras** = 15 ciclos.
- Tiempo que tarda en enviar **una palabra** por el bus = 1 ciclo (habrá que hacerlo 4 veces).

Tiempo necesario para copiar un bloque MP a MC: $t_p = 1 + 1 * 15 + 4 * 1 = 20\ ciclos$.

El número de bytes transferidos por ciclo serian: $4 * 4/20\ = 0{,}80\ bytes/ciclo$.

Medidas de rendimiento

$$t_{acceso} = t_h + t_p$$

$t_{acceso} =$ Tiempo de servicio de una referencia en memoria.
$t_h =$ Tiempo para determinar si es un acierto o una fallada y usar la referencia en caso de acierto.
$t_p =$ Tiempo de penalización para resolver la referencia en acceder al siguiente nivel de jerarquía de memoria.

El buffer de escritura: Es donde quedan almacenadas las escrituras pendientes de llevarlo al MP, tiene una longitud ilimitada.

Escritura inmediata sin asignación: La única penalización será la penalización por falladas de lectura

$$t_p = (1 - p_e) * (t_{block} + t_h)$$

$p_e =$ proporción de escrituras.

Escritura retardada con asignación:

$$t_p = p_m * (2 * t_{block} + t_h) + (1 - p_m) * (t_{block} + t_h)$$

$p_m =$ proporción de bloques modificados.

Media de tiempo de acceso a memoria:

$$t_{ma} = t_h + m * t_p$$

Número de referencias en memoria por instrucción:

$$nr_{instr} = \frac{nr}{I}$$

Proporción referencias de lectura o escritura de datos $= nr_{instr} - 1$

$$t_{ma} = \frac{t_{ma:fecth} + (nr_{instr} - 1) * t_{ma:datos}}{nr_{instr}}$$

$$t_{ma:fetch} = t_{h:fetch} + m_{fetch} * t_{p:fetch}$$

$$t_{ma:datos} = t_{h:datos} + m_{datos} * \left(pm * t_{p:modif} + (1 - pm) * t_{p:no\ modif}\right)$$

$$CPI_{ideal} = \frac{n_{clicles\ ideal}}{n_{ins}}, \qquad CPI = CPI_{ideal} + falladas_{fetch} + falladas_{datos}$$

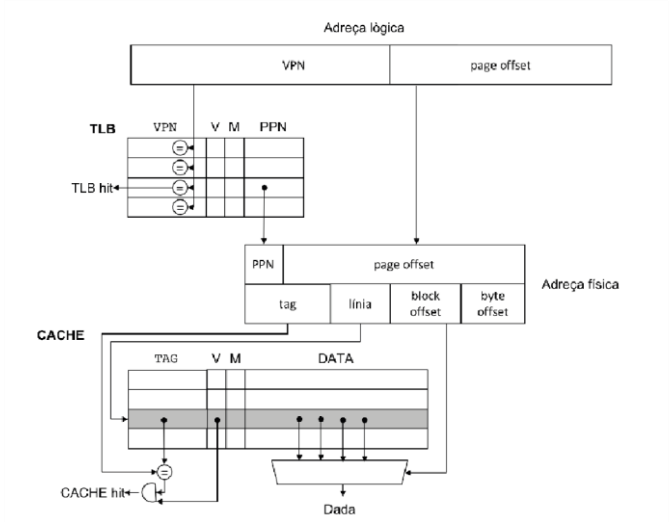
$$falladas_{fetch} = m_{fetch} * t_{p:fetch}$$

$$falladas_{datos} = (nr_{instr} - 1) * m_{datos} * (pm * t_{p:datos:modif} + (1 - pm) * t_{p:datos:nomodif}$$

$$t_{eje} = n_{ins} * CPI * t_c = \left(n_{ins} * CPI_{ideal} + n_{falladas} * t_p\right) * t_c$$

La MV usa dos espacios de direccionamiento:

- Espacio de direccionamiento lógico o virtual (MV)l:
    - Exclusivo de cada programa
    - El tamaño es la máxima permitida por el número de bits que usa el ordenador.
    - Da la impresión de que sólo hay un programa y memoria ilimitada.
    - Las direcciones del compilador y de la CPU son direcciones lógicas.
- Espacio de direccionamiento físico (MF):
    - "Direcciones reales"
    - Durante la ejecución del programa habrá que cargarlo a la MF.
    - La MV permite recordar cuales son las direcciones físicas que se han cargado en direcciones lógicas en un programa.
    - Cuando la CPU solicita un dato, la dirección lógica traduce la dirección física dónde está el dato.



Características:

- Los bloques de la MC son equivalentes en la MV, se llaman páginas.
- Las falladas en MV se llaman falladas de página.
- Una página virtual es un bloque de memoria contiguo y de tamaño fijo T de un programa.
- Cada página tendrá un numero de página virtual (*Virtual Page Number* VPN)
- Cada página tiene una dirección lógica A, $VPN = A/T = 32 - t$ bits altos de la dirección
- *Page offset* = t bits de menor peso de la dirección ($T = 2^t$ bits).

- Durante la ejecución hay que ir cargando las páginas que se vayan necesitando en la MF.
- Cada subdivisión contigua de tamaño T de la MF se llamará marco de página (*page frame*).
- Cada marco de página tendrá asociado un número de página física (*physical page number* PPN).
- Las páginas de un programa que, en un momento dado, no estén en la MF estarán en el disco.

- La asignación de páginas a marcos de página la realiza el sistema operativo en función del espacio disponible en cada momento.
- **El VPN no determina dónde van las páginas**, como pasaba en la MC completamente asociativa.

Traducción de direcciones:

- El procesador trabaja con direcciones lógicas.
- Cada vez que necesita leer o escribir un dato pregunta por la dirección lógica de este dato a la unidad de gestión de memoria (MMU)
- Traducción de direcciones: la MMU traduce la dirección lógica en la dirección física, dónde realmente está el dato.

- Dado el número de la página (VPN) a la que pertenece la dirección, el sistema de traducción determina en que marco de página (PPN) de la memoria física se encuentra.
- La selección de que marco de página se carga en una página dada la realiza el OS.
- Para poder recordar después a que marco pertenece la página cargada con un VPN determinado, se usa la tabla llamada tabla de páginas (hay una por programa).
- Cada fila de la tabla de páginas se llama entrada de la tabla de páginas (PTE, incluye la PPN, y los 2 bits V y D), hay tantas como páginas virtuales (VPNs) haya.

Fallada de página:

- Se produce una fallada de página (*page fault*) cuando la CPU referencia una dirección lógica que pertenece a una página que no se encuentra en la memoria física, es decir cuando el bit P de la entrada de la TP correspondiente a la VPN solicitada vale 0.
  Cuando esto sucede:
  1- Se lee del disco la página.
  2- Cegarla en un marco de página de la memoria física.
  3- Actualizar la información de la TP.
  4- Reintentar la operación.
- Escritura diferida con asignación:
  - Si hay que cargar una página en la MF y no queda ningún marco de la página libre se reemplaza, el algoritmo de reemplazamiento de páginas es el LRU.
  - Al usar escritura retardada hay un bit M de página modificada.
  - Antes de reemplazar una página con bit $M = 1$ hay que escribir en el disco la página modificada.

Traducción rápida con TLB, cache de traducciones (*translation-lookaside buffer*):

- Para leer o escribir un dato hay que acceder antes a la TP

Tabla de páginas

VPN	P	M	PPN
0x00000	1	0	0x01
0x00001	0	0	
0x00002	1	0	0x03
0x00003	1	1	0x00
0x00004	1	0	0x02
0x00005	1	0	0x04
0x00006	0	0	
…	…		
0xFFFFF	0	0	

TLB

VPN	V	M	PPN
0x00002	1	0	0x03
0x00000	1	0	0x01
0x00004	1	0	0x02
0x00003	1	1	0x00

- La entrada del TLB = VPN + copia de una entrada de la TP (P, M y PPN).
- Las búsquedas se hacen por la VPN.
- Para saber si una entrada esta validada se usa el mismo bit P, pero se llama V (Validación).

- Si la VPN corresponde con la VPN de la TLB es un acierto (TLB hit)
    - Si V = 0 después del acierto, se producirá una fallada de página, una vez resuelta la fallada se reescribirá la entrada a la TLB
- Si la VPN no corresponde con ninguna VPN de la TLB es una fallada de TLB (TLB miss)
    - Se copia la entrada TP a la TLB (sin mirar si P es 0 o 1)
        - Reemplazamos primero las entradas TLB con el bit V a 0.

Protección y compartición

La MV permite compartir memoria con múltiples procesos del ordenador de manera segura, un proceso no puede acceder al espacio de otro proceso, gracias a la TP.

El modo usuario no puede modificar ni la TP ni la TLB, solo el modo sistema.

Existe una protección contra escritura en determinadas páginas, el permiso de escritura se incluye con el bit E en cada entrada de la TP y la TLB. Esto ocurre cuando un proceso quiere acceder al espacio de direccionamiento de otro, el sistema operativo asigna una página lógica de la pagina física del proceso que se desea compartir, esta pagina puede tener permisos de escritura o no.

Cálculos

$$tamaño\ página = 2^t \rightarrow bits\ offset = t$$

$$Bits\ de\ la\ VPN\ =\ bits\ del\ procesador\ -\ bits\ offset$$

$$Páginas\ virtuales = \frac{espacio\ direcciones\ de\ los\ programas}{tamaño\ página}$$

$$Marcos\ de\ página\ /\ Páginas\ físicas = \frac{espacio\ de\ la\ memoria\ física}{tamaño\ página}$$

$$Tamaño\ de\ entrada\ TP = bit\ V + bit\ D + bits\ PPN$$

$$Tamaño\ memoria\ virtual = páginas * tamaño\ páginas$$