**UNIVERSIDAD CARLOS III DE MADRID. DEPARTAMENTO DE INFORMÁTICA**
**GRADO EN INGENIERÍA INFORMÁTICA. DOBLE GRADO ADE E INGENIERÍA INFORMÁTICA**
**ESTRUCTURA DE COMPUTADORES**
**11 de enero de 2019** **Examen final**

Para la realización del presente examen se dispondrá de **2:30 horas.**
**NO** se podrán utilizar libros, apuntes **ni** calculadoras (o dispositivos electrónicos) de ningún tipo.

---

**Ejercicio 1 (1puntos).** Indique el valor decimal del número no normalizado más grande que se puede representar utilizando el estándar IEEE 754 de simple precisión. Indique también el valor decimal del siguiente número (mayor que el no normalizado más grande) representable. ¿Qué diferencia hay entre estos dos números consecutivos representables? ¿Se mantiene constante esta diferencia para todos los números representables en este estándar?

**Ejercicio 2 (3 puntos).** Considere la rutina `Contabilizar`. Esta rutina acepta **cinco** parámetros de entrada:

- La dirección de inicio de una matriz (se almacena por filas) de números de tipo `int`, de dimensión `MxN`
- El número de filas de la matriz (`M`).
- El número de columnas de la matriz (`N`).
- Un número de fila `i`.
- Un número de columna `j`.

La función devuelve dos valores. El primero es el valor del elemento almacenado en la posición (`i, j`) de la matriz. En el segundo valor se devuelve el número de elementos de la matriz cuyo valor es igual al situado en la posición (`i,j`). Para este segundo valor también se tiene que tener en cuenta el elemento situado en la posición (`i,j`). Es decir, el segundo valor devuelto por la función siempre tendrá un valor mayor o igual que 1.

Se pide:

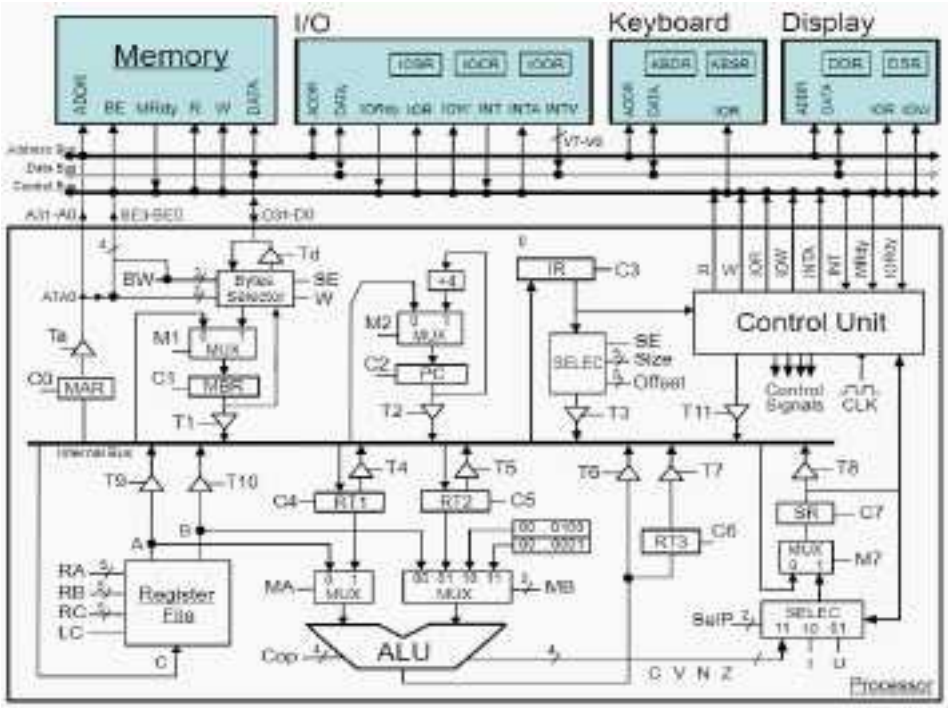
a) Codifique correctamente la rutina `Contabilizar` anteriormente descrita. Ha de seguirse estrictamente el convenio de paso de parámetros del MIPS visto en clase. Se valorará positivamente que dicha función invoque a otras funciones.

b) Dada el siguiente segmento de datos:

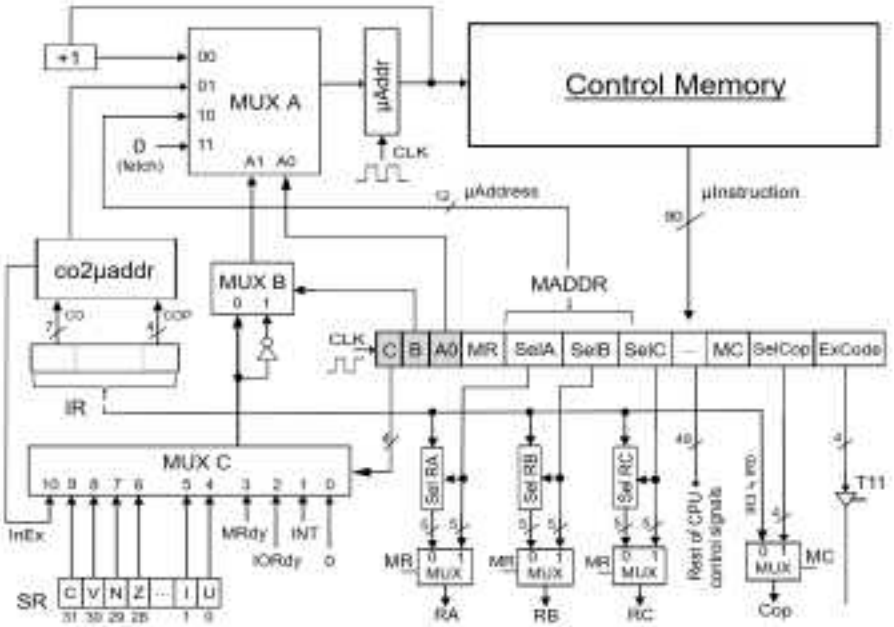
```
.data
  A: .word      8, 4, 3, 3, 5,
                2, 3, 0, 4, 5,
                0, 0 ,1, 2, 3
  M: .word 3
  N: .word 5
```

Codifique el fragmento de código que permite invocar correctamente a la función `Contabilizar` para la matriz `A` de dimensión `MxN` y los valores de `i=2` y `j=3`. A continuación se deben imprimir los dos valores devueltos por la función.

**Ejercicio 3 (3 puntos).** Dado el procesador WepSIM con la siguiente estructura:




Este procesador dispone de una Unidad de control representada por la siguiente figura:



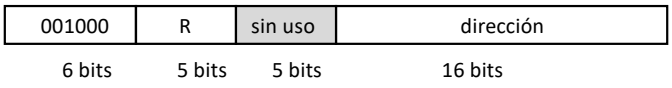

Especifique las operaciones elementales y señales de control necesarias para ejecutar la instrucción máquina `ADDM (Ra) (Rb) direccion` (incluya el ciclo de *fetch*). Esta instrucción suma el contenido de las posiciones de memoria almacenadas en los registros `Ra` y `Rb`. El resultado se almacena en la dirección de memoria `dirección`. La instrucción anterior ocupa dos palabras y el código de operación 6 bits. Los campos para los registros Ra y Rb se codifican a continuación del código de operación. El campo dirección se codifica en la segunda palabra de la instrucción.

NOTA: Asuma que `R29` actúa como puntero de pila, que el puntero de pila apunta a cima de pila y que la pila crece hacia direcciones decrecientes de memoria. Considere también que el registro `R0` es de solo lectura y almacena el valor 0.

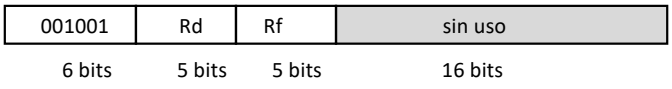
**Ejercicio 4 (1 puntos).** Sobre el procesador WepSim se quiere ejecutar un juego de instrucciones que incluye entre otras las siguientes:

li R, valor (similar a la pseudoinstrucción li del MIPS)

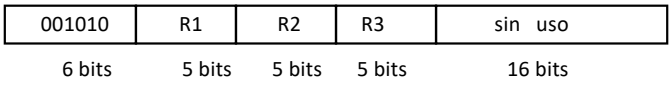

lw R, dirección (similar a la instrucción lw del MIPS)



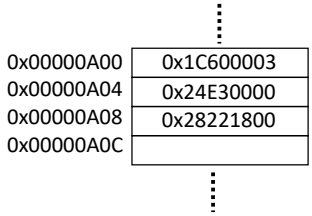

lw Rd, (Rf) (similar a la instrucción lw del MIPS)




add R1, R2, R3 (similar a la instrucción lw del MIPS)




Considere la siguiente porción de la memoria principal de WepSim:




Si el valor de del contador de programa es 0x00000A04, indique los valores de los registros PC, MAR, MBR e IR una vez ejecutada la instrucción almacenada en la dirección anterior. Indique, asimismo, a qué instrucción en ensamblador (código de operación y operandos) se corresponde la instrucción máquina almacenada en la posición de memoria 0x00000A04.

**Ejercicio 5 (2 puntos).** Considere el siguiente fragmento de código del ensamblador MIPS 32:

```
.data:
            v1: .space 1024
            v2: .space 4096
.text:
            li   $t0, 0
            li   $t1, 0
            li   $t2, 0
            li   $t3, 1024
bucle:      bge  $t0,  $t3, fin
            lb   $t4, v1($t1)
            lw   $t5, v2($t2)
            add  $t4, $t4, $t5
            sw   $t4, v2($t2)
            addi $t0, $t0, 1
            addi $t1, $t1, 1
            addi $t2, $t2, 4
            b    bucle
fin:
```

Dicho código se ejecuta en una arquitectura con un ancho de palabra de 32 bits, que incluye una memoria caché de instrucciones totalmente asociativa de 64 KB y una caché de datos de 128KB, asociativa por conjuntos de 8 vías. Ambas cachés tienen líneas de 64 bytes. Asumiendo que la caché está inicialmente vacía y que el programa comienza en la dirección de memoria 0x00000000. Se pide:

a) Indique el número de líneas y conjuntos de la memoria caché de datos.
b) Indique la tasa de aciertos que produce la ejecución del fragmento anterior en la caché de instrucciones.
c) Indique la tasa de aciertos que produce la ejecución del fragmento anterior en la caché de datos.
d) Si se cambia la caché de datos por una totalmente asociativa, qué repercusiones tendría sobre la tasa de aciertos calculada y sobre el rendimiento global de la memoria caché. Justifique su respuesta.

# *Guía rápida del ensamblador MIPS32*

| | |
|---|---|
| abs Rdest, Rsrc | Valor absoluto |
| add Rdest, Rsrc1, Src2 | Suma con desbordamiento |
| addu Rdest, Rsrc1, Src2 | Suma sin desbordamiento |
| and Rdest, Rsrc1, Src2 | Operación lógica AND |
| div Rsrc1, Rsrc2 | Divide con desbordamiento. Deja el cociente en el registro *lo* y el resto en el registro *hi* |
| divu Rsrc1, Rsrc2 | Divide sin desbordamiento. Deja el cociente en el registro *lo* y el resto en el registro *hi* |
| div Rdest, Rsrc1, Src2 | Divide con desbordamiento |
| div Rdest, Rsrc1, Src2 | Divide sin desbordamiento |
| mul Rdest, Rsrc1, Src2 | Multiplica sin desbordamiento |
| mulo Rdest, Rsrc1, Src2 | Multiplica con desbordamiento |
| mulou Rdest, Rsrc1, Src2 | Multiplicación con signo y con desbordamiento |
| mult Rsrc1, Rsrc2 | Multiplica, la parte baja del resultado se deja en el registro *lo* y la parte alta en el registro *hi* |
| mult Rsrc1, Rsrc2 | Multiplica con signo, la parte baja del resultado se deja en el registro *lo* y la parte alta en el registro *hi* |
| neg Rdest, Rsrc | Niega el valor (detecta desbordamiento) |
| negu Rdest, Rsrc | Niega el valor (sin desbordamiento) |
| nor Rdest, Rsrc1, Src2 | Operación Lógica NOR |
| not Rdest, Rsrc | Operación Lógica NOT |
| or Rdest, Rsrc1, Src2 | Operación Lógica OR |
| rem Rdest, Rsrc1, Src2 | Resto (Módulo), pone el resto de dividir Rsrc1 por Src2 en el registro Rdest |
| rol Rdest, Rsrc1, Src2 | Rotar a la izquierda |
| ror Rdest, Rsrc1, Src2 | Rotar a la derecha |
| sll Rdest, Rsrc1, Src2 | Desplazamiento lógico de bits a la izquierda |
| srl Rdest, Rsrc1, Src2 | Desplazamiento lógico de bits a la derecha |
| sra Rdest, Rsrc1, Rsrc2 | Desplazamiento aritmético de bits a la derecha |
| sub Rdest, Rsrc1, Src2 | Resta (con desbordamiento) |
| subu Rdest, Rsrc1, Src2 | Resta (sin desbordamiento) |
| xor Rdest, Rsrc1, Src2 | Operación Lógica XOR |

| | |
|---|---|
| b etiqueta | Bif. incondicional a la instrucción que está en etiqueta. |
| beq Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 es igual a Src2. |
| beqz Rsrc, etiqueta | Bif. condicional si el registro Rsrc es igual a 0. |
| bge Rsrc1, Src2, etiqueta | Bif. condicional si el registro Rsrc1 es mayor o igual a Src2 (con signo). |
| bgeu Rsrc1, Src2, etiq | Bif.. condicional si el registro Rsrc1 es mayor o igual a Src2 (sin signo). |
| bgez Rsrc, etiqueta | Bif. condicional si el registro Rsrc es mayor o igual a 0. |
| bgezal Rsrc, etiqueta | Bif. condicional si el registro Rsrc es mayor o igual a 0. Guarda la dirección actual en el registro $ra ($31) |
| bgt Rsrc1, Src2, etiqueta | Bif. condicional si el registro Rsrc1 es mayor que Src2 (con signo). |
| bgtu Rsrc1, Src2, etiqueta | Bif. condicional si el registro Rsrc1 es mayor que Src2 (sin signo. |
| bgtz Rsrc, etiqueta | Bif. condicional si Rsrc es mayor que 0. |
| ble Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 es menor o igual a Src2 (con signo). |
| bleu Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 es menor o igual a Src2 (sin signo). |
| blez Rsrc, etiqueta | Bif. condicional si Rsrc es menor o igual a 0. |
| bltzal Rsrc, etiqueta | Bif. condicional si Rsrc es menor que 0. Guarda la dirección actual en el registro $ra ($31). |
| blt Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 es menor que Src2 (con signo). |
| bltu Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 es menor que Src2 (sin signo). |
| bltz Rsrc, etiqueta | Bif. condicional si Rsrc es menor que 0. |
| bne Rsrc1, Src2, etiqueta | Bif. condicional si Rsrc1 no es igual a Src2. |
| bnez Rsrc, etiqueta | Bif. condicional si Rsrc no es igual a 0. |
| j etiqueta | Salto incondicional. |
| jal etiqueta | Salto incondicional, almacena la dirección actual en $ra ($31). |
| jalr Rsrc | Salto incondicional, almacena la dirección actual en $ra ($31). |
| jr Rsrc | Salto incondicional. |

| | |
|---|---|
| la Rdest, dirección | Carga dirección en Rdest (el valor de dirección, no el contenido) |
| lb Rdest, dirección | Carga el byte de la dirección especificada y extiende el signo |
| lbu Rdest, dirección | Carga el byte de la dirección especificada, no extiende el signo |
| ld Rdest, dirección | Carga Rdest y Rdest + 1 con el valor del double (64 bits) que se encuentra a partir de la dirección especificada. |
| lh Rdest, dirección | Carga 16 bits de la dirección especificada, se extiende el signo |
| lhu Rdest, dirección | Carga 16 bits de la dirección especificada, no se extiende signo |
| lw Rdest, dirección | Carga una palabra de la dirección especificada. |

# Soluciones

## Ejercicio 1.

- El número no normalizado más grande utilizando el estándar IEEE 754 de simple precisión es aquel que tiene todo ceros en el exponente y la mantisa más grande. Es decir:

  `0  00000000  11111111111111111111111`

  Su valor es: $0,11111111111111111111111_{(2} \times 2^{-126} = (1-2^{-23}) \times 2^{-126}$ en decimal.

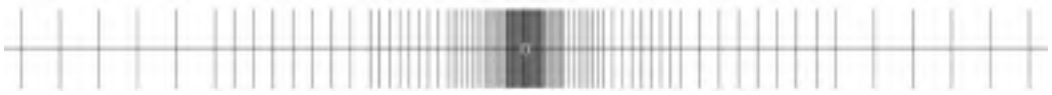
- El siguiente número representable es el número normalizado:

  `0  00000001  00000000000000000000000`

  Su valor decimal es $1,0 \times 2^{-126}$ en decimal.

- Como se puede observar la diferencia entre ambos números consecutivos es, cuantitativamente de $(2^{-23}) \times 2^{-126}$ y cualitativamente de ser uno el menor normalizado representable y el otro el mayor no normalizado representable.

- Esta diferencia no se mantiene constante para todos los números representables en este estándar porque la densidad de números representables disminuye a medida que nos alejamos del 0.



**Ejercicio 2.**

a) A esta función se le pasan 5 argumentos, los cuatro primeros se pasan en `$a0`, `$a1`, `$a2` y `$a3` y el quinto se pasa en la pila. La función devuelve dos valores, el primero se devuelve en $v0 y el segundo en `$v1`.

```
Contabilizar:   addi  $sp, $sp, -20
                sw    $a0, 0($sp)
                sw    $a1, 4($sp)
                sw    $a2, 8($sp)
                sw    $a3, 12($sp)
                sw    $ra, 16($sp)

                ; la función ObtenerNumero devuelve
                ; el elemento almacenado en la posición i, j.
                ; Acepta los mismo 5 argumentos
                ; hay que volver a dejar el quinto en la pila
                lw    $t0, 24($sp)
                addi  $sp, $sp, -4
                sw    $t0, ($sp)
                jal   ObtenerNumero  ; obtiene el número almacenado
                                     ; (i,j)
                addi  $sp, $sp, 4
                move  $v1, $v0

                lw    $a0,  ($sp)
                lw    $a1,  4($sp)
                lw    $a2,  8($sp)
                move  $a3, $v1

                jal ContarElementos   ; devuelve el número de
                                      ; elementos igual al almacenado
                                      ; en (i,j) que está en $v1

                move  $t1, $v1
                move  $v1, $v0
                move  $v0, $t1

                lw    $a0, ($sp)
                lw    $a1, 4($sp)
                lw    $a2, 8($sp)
                lw    $a3, 12($sp)
                lw    $ra, 16($sp)

                addi  $sp, $sp, 20
                jr    $ra
```

```
ObtenerNumero:  ; se calcula la dirección del elemento (i,j)
                ; esta función acepta los mismo 5 parámetros que la
                ; función anterior

                lw   $t0, ($sp) ; se accede al quinto elemento
                                ; está en la cima de la pila

                li   $t1, 4
                mul  $t2, $t1, $t0
                mul  $t3, $a2, $t1
                mul  $t3, $t3, $a3
                add  $t3, $t3, $a0
                add  $t3, $t3, $t2
                lw   $v0, ($t3)
                jr   $ra

ContarElementos:li   $t0, 0
                li   $v0, 0
                mul  $t1, $a1, $a2 ; número de elementos de la matriz
     bucle:     bge  $t0, $t1, fin
                lw   $t2, ($a0)
                bneq $t2, $a3, noigual
                addi $v0, $v0, 1
     noigual:   addi $t0, $t0, 1
                addi $a0, $a0, 4
                b    bucle
          fin:  jr   $ra
```

b) Para invocar a la función se necesita el siguiente fragmento de código:

```
la   $a0, A
lw   $a1, M
lw   $a2, N
li   $a3, 2
li   $t0, 3
addi $sp, $sp, -4
sw   $t0, ($sp)
jal  Contabilizar
addi $sp, $sp, 4

move $a0, $v0
li   $v0, 1
syscall

move $a0, $v1
syscall
```

**Ejercicio 3.**

El registro Ra se codifica en los bits 25-**21** de la primera palabra de la instrucción y el registro Rb se codifica en los bits 20-**16** de dicha palabra.

- *Fetch*:

| Ciclo | Operaciones elementales | Señales de control |
|---|---|---|
| C1 | MAR ← PC | T2, C0 |
| C2 | MBR ←MP[MAR]<br>PC ← PC+4 | Ta, R, BW=11, M1=1, C1,<br>M2=1, C2 |
| C3 | IR ← MBR | T1, C3 |
| C4 | Decodificar y salto a Código de operación. | C=0, B=0, A0=1 |

- Código del microprograma pedido. Dado que la instrucción es aritmética, se considera que se modifica el registro de estado

| Ciclo | Operaciones elementales | Señales de control |
|---|---|---|
| C1 | MAR ← Ra | C0, T11, MR=0, SelA=**21** (10101) |
| C2 | MBR ← MP[MAR] | Ta, R, BW=11, M1=1, C1 |
| C3 | RT1 ← MBR | T1, C4 |
| C4 | MAR ← Rb | C0, T11, MR=0, SelA=**16** (10000) |
| C5 | MBR ← MP[MAR] | Ta, R, BW=11, M1=1, C1 |
| C6 | RT2 ← MBR | T1, C5 |
| C7 | RT3 ← RT1 + RT2<br>Actualizar SR | MA=1, MB=1, MC=1, Cop=+, C6<br>SelP=11, M7, C7 |
| C8 | MAR ← PC | T2, C0 |
| C9 | MBR ←MP[MAR]<br>PC ← PC+4 | Ta, R, BW=11, M1=1, C1,<br>M2=1, C2 |
| C10 | MAR ← MBR | T1, C0 |
| C11 | MBR ← RT3 | T7, C1 |
| C12 | MP[MAR] ← MBR | Ta, Td, W, BW=11 |
| C13 | Salto a fetch | C=0, B=1, A0=0, MADDR=fetch |

**Ejercicio 4.**

- La instrucción es 0x24E30000  que en binario es:

  0010 0100 1110 0011 0000 0000 0000 0000

  Los primeros 6 bits son el código de operación: 001001 que se corresponde con la instrucción lw Rd, (Rf)

  El campo $Rd es 00111 (R7)
  El campo $Rf es 00011 (R3)

  Por tanto, la instrucción en ensamblador es lw R7, (R3)

- Una vez ejecutada la instrucción:

  PC = 0x00000A08
  MAR = <contenido de R3>
  MBR = <Contenido de la dirección de memoria indicada en R3 y que se encuentra en MAR>
  IR = 0x24E30000

**Ejercicio 5.**

a) La caché de datos tiene un tamaño de 128KB = $2^{17}$ bytes. Las líneas son de 64 bytes.
El número de líneas es de $2^{17} / 2^6 = 2^{11} = 2048$.
El número de conjuntos es de $2^{11} / 2^3 = 2^8 = 256$

b) Cada línea de la caché de instrucciones tiene 64 bytes y permite almacenar 16 instrucciones máquina. Todas las instrucciones del MIPS ocupan una palabra.

Cuando se accede a la primera instrucción se produce un fallo y se trae a la memoria caché de instrucciones un bloque de 16 instrucciones máquina (64 bytes). Este bloque incluye a todas las instrucciones del fragmento de código de enunciado, por tanto, no se vuelven a producir más fallos.

El número de accesos a instrucciones que se produce en la ejecución del fragmento anterior es:
4 + 1024 x 9 + 1 = 4 + 9216 + 1 = 9221.

Por tanto, la tasa de aciertos es de: (9221 – 1) / 9221

c) En cada vuelta del bucle se produce los siguientes accesos a datos:
- Una lectura de un byte de un vector de bytes.
- Una lectura de 4 bytes de un vector de enteros.
- Una escritura de 4 bytes en un vector de enteros.

Cada 64 iteraciones se producen:
- Un fallo en v1.
- Cuatro fallos en v2.
- El número de accesos a memoria es de 64 x 3 = 192

Como el número total de iteraciones (1024) es múltiplo de 64, la proporción se mantiene y, por tanto, la tasa de aciertos es de:
(192-5) / 192 = 187 / 192 = (3*1024) – 5*(1024/64) / (3*1024)

d) Cada conjunto de 8 vías supone 8*64 = 512 bytes, por lo que v1 precisa 2 conjuntos y v2 precisa 8 conjuntos, sin haber solapamiento entre dichos conjuntos (al tenerse 256 conjuntos distintos). Además, los datos no se reutilizan. Por tanto, no hay expulsiones y el comportamiento es similar a una caché totalmente asociativa (en cuanto a número de fallos). Una caché asociativa requeriría más bits para la etiqueta. Si la búsqueda de las etiquetas se hace en paralelo, el rendimiento sería similar, pero se necesitaría una compleja circuitería para examinar en paralelo las etiquetas de todas las líneas de la caché. Si la búsqueda no se hace en paralelo, el rendimiento sería peor.