

# PRÁCTICA 4

# MEMORIAS SICAM

[i72cascm@uco.es](mailto:i72cascm@uco.es)  
Manuel Casas Castro

## Introducción

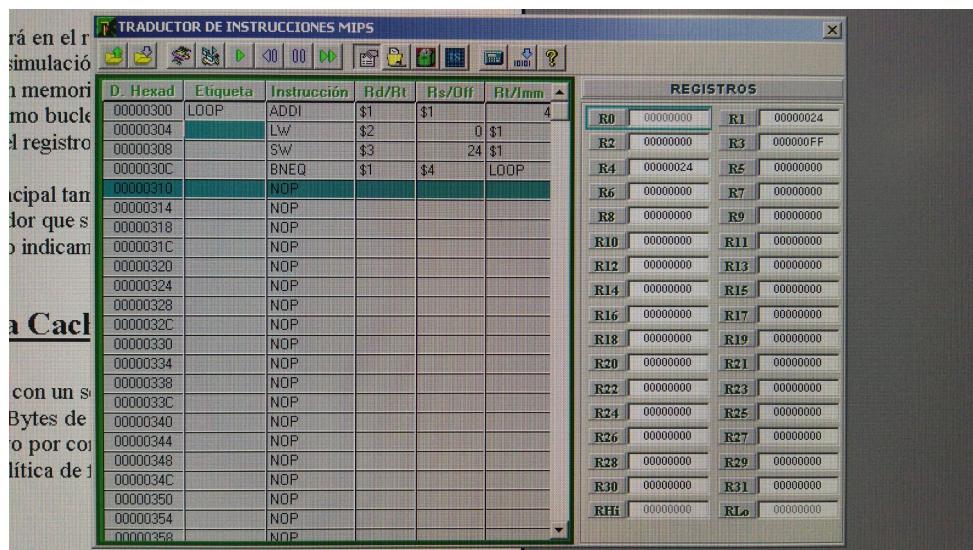
En estas memorias de la práctica 4 de Arquitectura de Computadores, se tiene como objetivo ayudar a la asimilación de conocimientos de la relación entre memoria caché y memoria principal y otros factores determinantes como la transferencia entre estos bloques, los tipos de caché o la latencia de acceso.

Para ello, en la herramienta de simulación SICAM introduciremos una serie de instrucciones en lenguaje MIPS para comprender el funcionamiento real de una memoria caché y su relación con memoria principal.

En primer lugar, realizaremos una simulación con un código ejemplo descrito en el enunciado de la práctica para de esta forma analizar los resultados obtenidos y compararlos con posteriores simulaciones.

## Simulación 1 (Código Ensamblador)

Para realizar esta simulación, indicaremos al simulador que nuestra posición inicial de instrucciones es 300 (Hexadecimal) ya que por defecto es la posición que el simulador sitúa. Tras meter las instrucciones en el código ensamblador debemos inicializar los registros \$3=FF y \$4=24 para los primeros análisis.



En este código podemos encontrar una serie de instrucciones contenidas en un ciclo loop cuya finalidad es mantener el bucle activo hasta que el registro \$4 y \$1 tengan el mismo contenido, para ello, la instrucción ADDI-\$1,\$1,4 sumará 4 al contenido \$1 por cada bucle realizado dando un total de 9 bucles hasta finalizar el programa ya que inicialmente en el registro \$4 inicializamos el contenido a 24. Esto se logra gracias a la instrucción BNEQ-\$1, \$4, LOOP hacia el loop inicializado en la instrucción primera (ADDI).

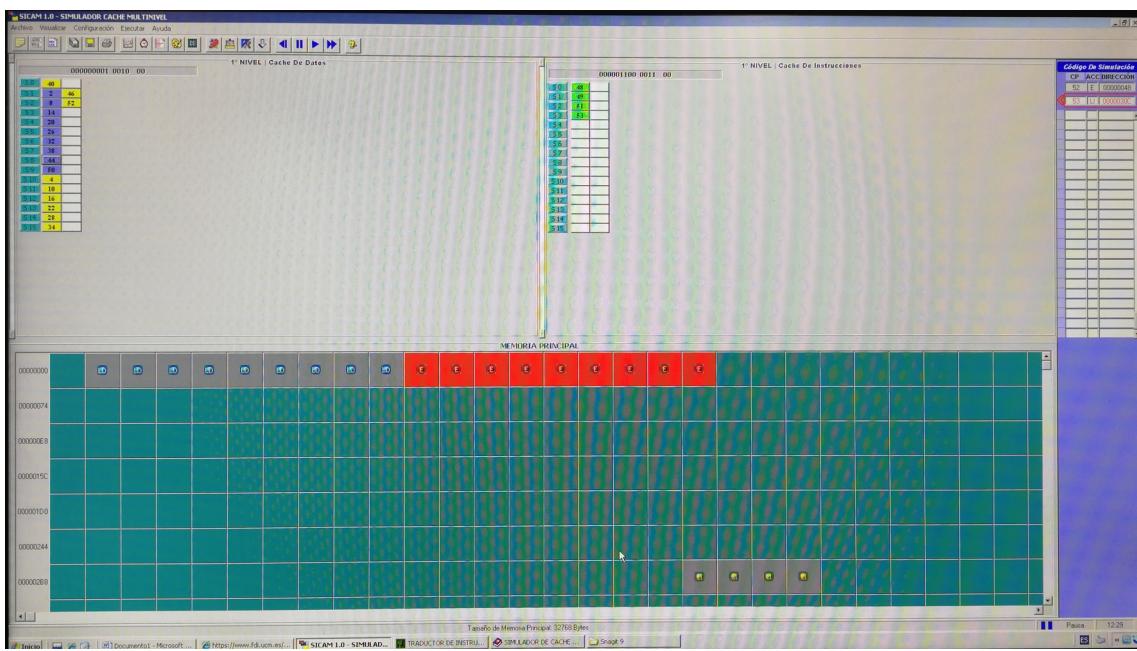
La instrucción SW-\$3, 24(\$1) guardará en memoria principal el registro \$3 (en este caso FF) en la posición 24+\$1, de esta manera, tras 9 bucles se guardarán en las posiciones de memoria 28, 2C, 30, 34, 38, 3C, 40, 44 y 48 el registro \$3.

La instrucción LW-\$2, 0(\$1) cargará en el registro \$2 la posición de memoria 0+\$1, por lo que en esta simulación esta instrucción no cargará nada en \$2 ya que el primer dato en memoria principal será guardado en el registro 28 ( $24+(\$1=4)$ ) y en el último bucle  $\$1=24$ , si se realizase un bucle más FF será cargado en el registro \$2.

Cabe destacar que en memoria principal también son guardadas las instrucciones del código ensamblador que se van ejecutando una a una a partir de la posición 300 tal y como indicamos al simulador al inicializarlo.

## **Simulación 1 (Memoria Caché)**

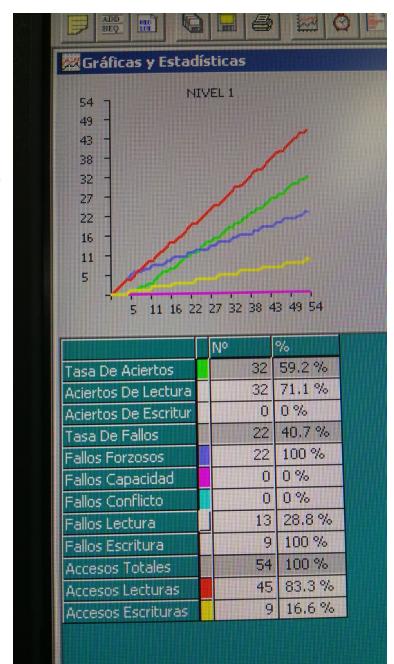
La primera simulación se realizará con un solo nivel de memoria caché con 128 bytes de tamaño de caché y 32kBytes de memoria principal. Tipo de reemplazo FIFO y mapeo asociativo por conjuntos. El tipo de caché será mixta, con escritura directa y la política de fallo de escritura será: ubicar en escritura. Una palabra por bloque.



Al realizar la simulación podemos observar que en el caché de instrucciones se escriben las cuatro instrucciones que introducimos en el código ensamblador y se mantienen durante toda la ejecución, haciendo posible que se acceda a ellas directamente desde la caché. Siendo la primera vez un fallo forzoso, ya que es la primera vez que se accede al bloque y el resto de veces aciertos.

De igual manera, en la caché de datos podemos observar que existen 9 fallos forzados debidos a la instrucción LW que busca cargar datos en el registro \$2, incrementándose en cada bucle y a su vez otras 9 escrituras en bloque debido a la instrucción SW que escribe en memoria principal el contenido FF en las posiciones indicadas anteriormente y que quedan registradas en caché.

Revisando las estadísticas de esta simulación, observamos que se ha realizado un total de 54 accesos de los cuales 32 han sido aciertos de lectura gracias a la caché de instrucciones. Estos aciertos son el número de veces que se ejecutan las instrucciones ( $9 \times 4 = 36$ ) menos el primer loop donde se realiza el fallo forzoso ( $36 - 4 = 32$ ).



Tiempos de Acceso por instrucciones										
CP	Ac	Dirección	TL n1	TT n1	TE n1	TPE n1	TL Mp	TT Mp	TE MP	T. Ej.
0	LI	00000300	4	8	0	7	3	4	0	0,19
1	LI	00000304	4	8	0	7	3	4	0	0,19
2	LD	00000004	4	8	0	7	3	4	0	0,19
3	LI	00000308	4	8	0	7	3	4	0	0,19
4	E	00000028	4	0	12	33	3	4	30	0,49
5	LI	0000030C	4	8	0	7	3	4	0	0,19
6	LI	00000300	4	8	0	0	0	0	0	0,12
7	LI	00000304	4	8	0	0	0	0	0	0,12
8	LD	00000008	4	8	0	7	3	4	0	0,19
9	LI	00000308	4	8	0	0	0	0	0	0,12
10	E	0000002C	4	0	12	33	3	4	30	0,49
11	LI	0000030C	4	8	0	0	0	0	0	0,12
12	LI	00000300	4	8	0	0	0	0	0	0,12
13	LI	00000304	4	8	0	0	0	0	0	0,12
14	LD	0000000C	4	8	0	7	3	4	0	0,19
15	LI	00000308	4	8	0	0	0	0	0	0,12
16	E	00000030	4	0	12	33	3	4	30	0,49
17	LI	0000030C	4	8	0	0	0	0	0	0,12

TIEMPO TOTAL: 10,71999 Micro Seg.

También observamos que la tasa de escrituras es 9, como indicamos anteriormente y el número de accesos a lectura 45 ( $45+9=54$  accesos totales). Tenemos un total de 22 fallos forzados entre lectura y escritura, de los cuales 9 son de escritura y 13 de lectura que se componen de los 9 de lectura de dato y de los 4 primeros de lectura de instrucción.

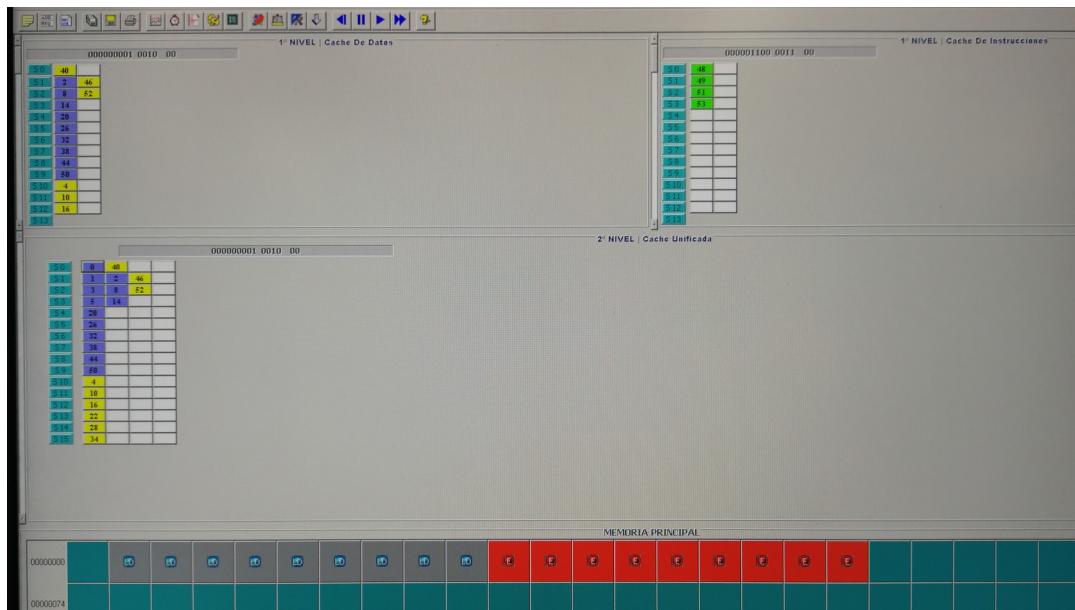
En los tiempos empleados para la ejecución del programa, podemos observar una variedad de tres tiempos distintos entre las instrucciones del programa. Las lecturas, tanto de datos como de instrucciones, podemos notar que cuando se realiza un fallo forzoso, el tiempo de ejecución es mayor que en los aciertos y esto se debe a que el dato debe ser buscado en memoria principal.

Lo mismo ocurre con todas las escrituras producidas, ya que todas se ven obligadas a acceder a memoria principal.

Sumando todos los tiempos de ejecución tenemos un total de 10,71999 microsegundos.

## Simulación 2

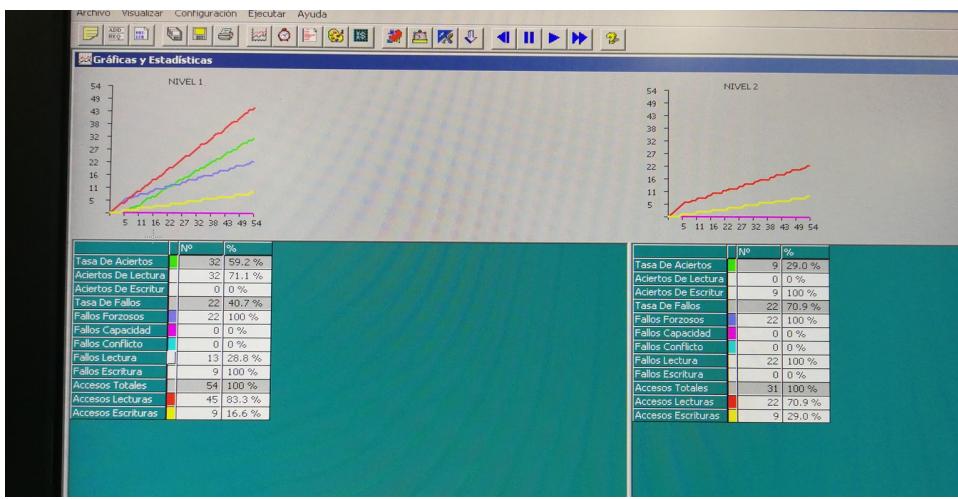
Realizando una segunda simulación con el mismo código ensamblador, pero esta vez con dos niveles de caché y dejando el resto de configuraciones intactas. Podemos observar que el nivel 1 de caché queda de manera similar a la simulación realizada con un solo nivel de caché:



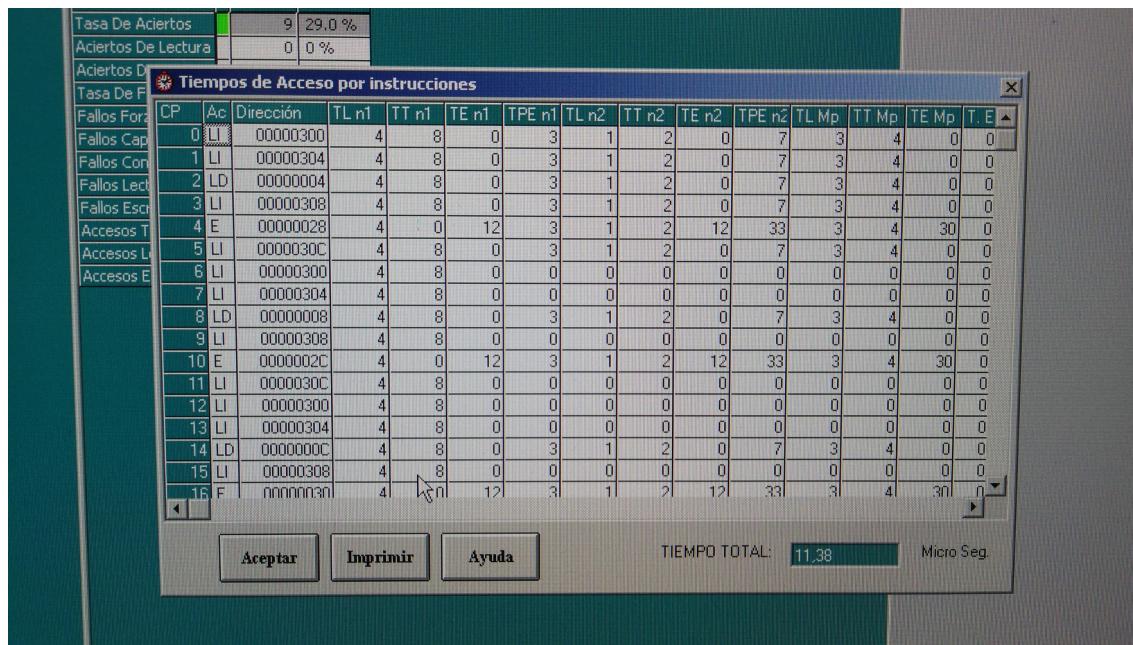
El contenido en memoria principal es igual al de la primera simulación, pero esta vez al existir dos niveles de caché el nivel L2 realiza copias de los datos que se alojan en el nivel uno en el momento que estos se guardan en el nivel 1.

Al haber configurado el segundo nivel de memoria con la misma capacidad que el nivel 1, podemos observar que existen el mismo número de bloques tanto en el nivel 1 (Caché de datos + Caché de instrucciones) que en el nivel 2.

Debemos destacar de esta simulación, que en el segundo nivel, al ser unificada, también se guardan las direcciones de las instrucciones del código ensamblador.



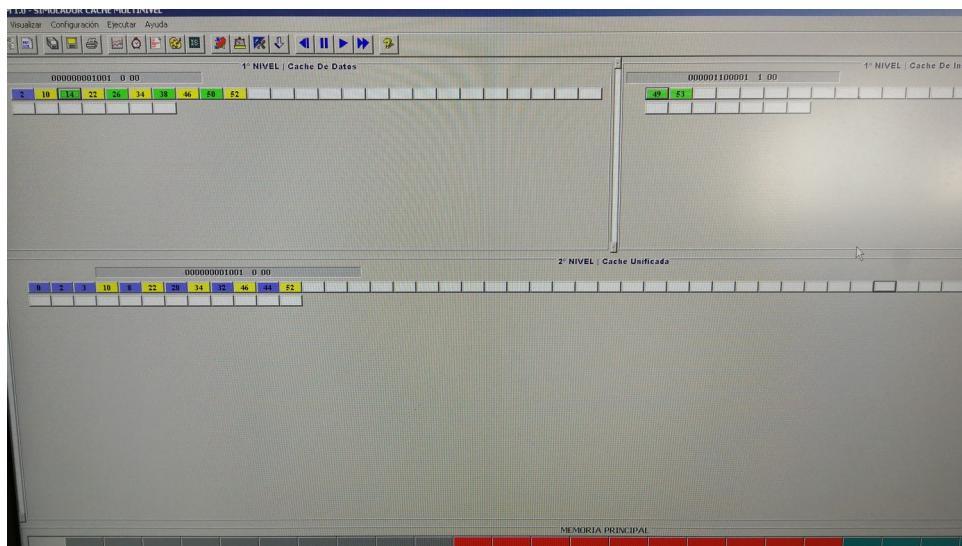
En las estadísticas de la simulación podemos observar que existen el mismo número de fallos en ambos niveles de caché y un número menor de accesos a lecturas. Podemos encontrar una diferencia notable en los aciertos, ya que en el nivel 2 de caché, encontramos que las 9 escrituras realizadas son aciertos. Esto se debe a que para guardar la información en el nivel dos, se realiza directamente desde el nivel 1, evitando así accesos innecesarios a memoria principal.



En los tiempos de ejecución del programa podemos notar que esta segunda simulación es ligeramente más lenta que la simulación primera, esto se debe al tiempo que se pierde en acceder al siguiente nivel de memoria aunque realmente el programa tarda aproximadamente medio microsegundo más en completarse.

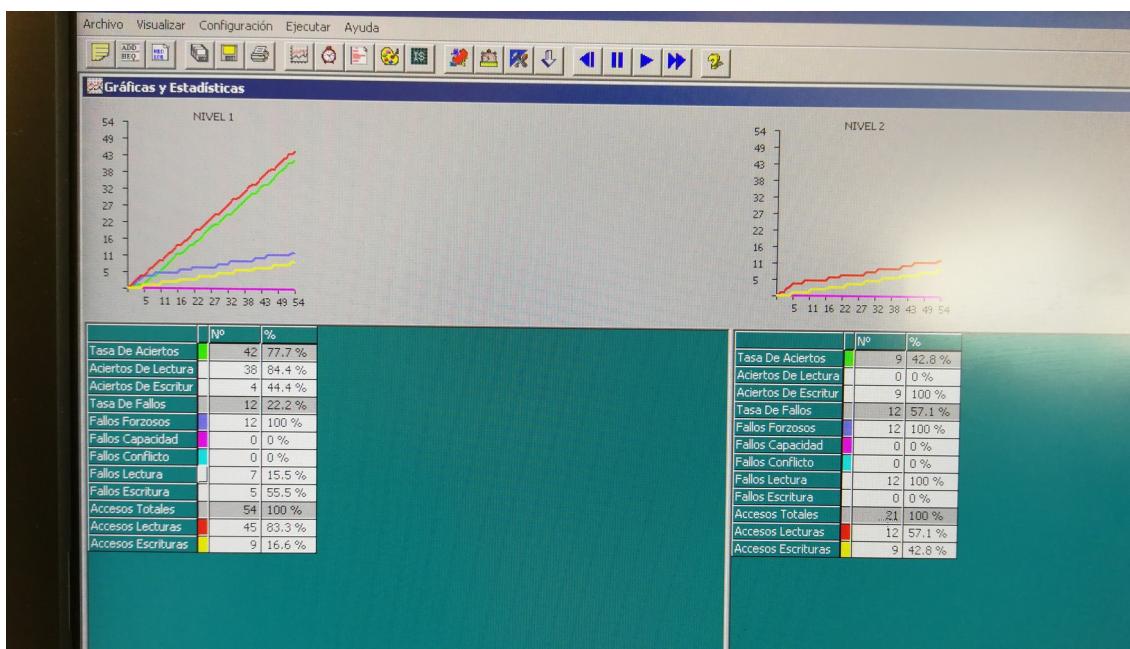
## Simulación 3

Siguiendo con el código ensamblador inicial, probamos la configuración de memorias de tipo asociativo y con dos palabras por bloque:



Podemos ver que en esta política de ubicación las palabras se guardan en cualquier bloque de memoria caché, comprobándose solo las etiquetas a la hora de buscar algún dato en caché.

Podemos observar que hemos obtenido ciertos aciertos en lectura, esto se debe a tener varias palabras por bloque ya que cuando se accede a memoria por algún dato, la caché trae tantos datos como palabras pueda almacenar en el bloque adicionalmente al dato el cual se está buscando.



En las estadísticas podemos observar una tasa de aciertos considerablemente mayor a las otras simulaciones, esto se debe a lo explicado anteriormente ya que al número de aciertos que obteníamos en las otras simulaciones, hay que sumarle el número de aciertos obtenidos por los accesos de memoria a las palabras que se guardaron en bloques de manera adicional a la información que se buscaba.

Con el incremento en el número de aciertos, se ve reducido el número de fallos que ha experimentado el simulador para llevar la ejecución a cabo y así mismo los accesos de memoria utilizados.

Tiempos de Acceso por instrucciones														
CP	Ac	Dirección	TL n1	TT n1	TE n1	TPE n1	TL n2	TT n2	TE n2	TPE n2	TL Mp	TT Mp	TE Mp	T. E ▲
0	LI	00000300	4	8	0	6	2	4	0	14	6	8	0	0
1	LI	00000304	4	8	0	0	0	0	0	0	0	0	0	0
2	LD	00000004	4	8	0	6	2	4	0	14	6	8	0	0
3	LI	00000308	4	8	0	6	2	4	0	14	6	8	0	0
4	E	00000028	4	0	12	6	2	4	12	36	6	8	30	0
5	LI	0000030C	4	8	0	0	0	0	0	0	0	0	0	0
6	LI	00000300	4	8	0	0	0	0	0	0	0	0	0	0
7	LI	00000304	4	8	0	0	0	0	0	0	0	0	0	0
8	LD	00000008	4	8	0	6	2	4	0	14	6	8	0	0
9	LI	00000308	4	8	0	0	0	0	0	0	0	0	0	0
10	E	0000002C	4	0	12	14	2	0	12	36	6	0	30	0
11	LI	0000030C	4	8	0	0	0	0	0	0	0	0	0	0
12	LI	00000300	4	8	0	0	0	0	0	0	0	0	0	0
13	LI	00000304	4	8	0	0	0	0	0	0	0	0	0	0
14	LD	0000000C	4	8	0	0	0	0	0	0	0	0	0	0
15	LI	00000308	4	8	0	0	0	0	0	0	0	0	0	0
16	F	00000000	4	0	12	6	2	4	12	36	6	8	30	0

TIEMPO TOTAL: 12.34 Micro Seg.

Con 12,34 microsegundos, podemos observar que la simulación ha tardado más en realizarse que las anteriores debido a que se necesita un poco más de tiempo en transmitir 2 palabras a un bloque que solo una.

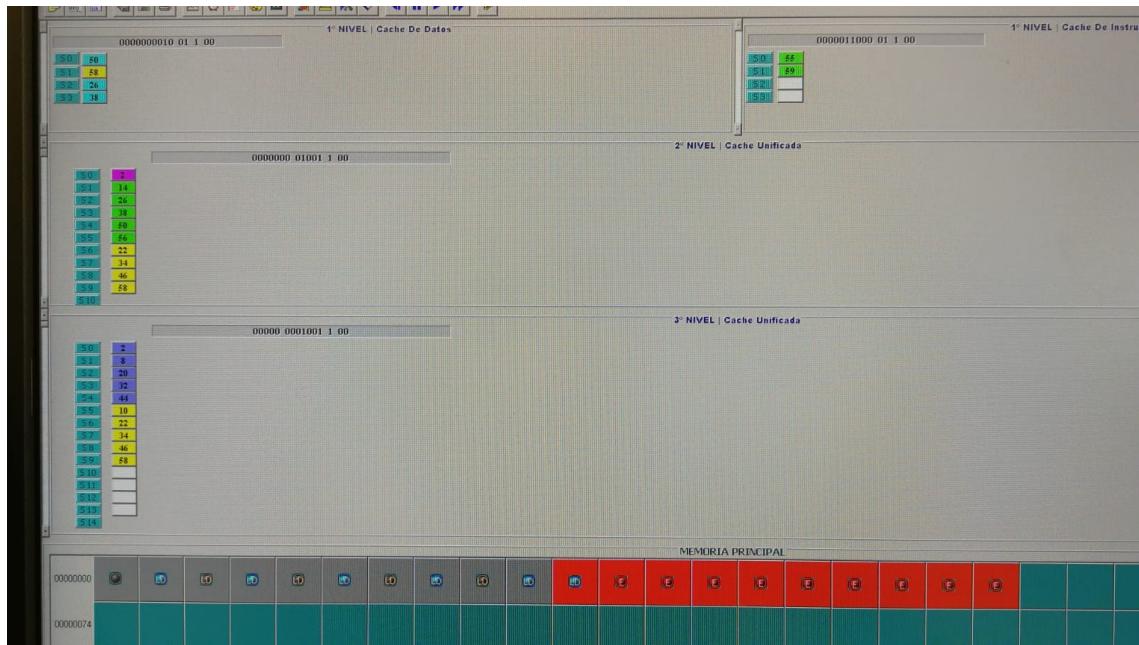
## Simulación 4 (Código ensamblador)

Vamos a realizar una nueva simulación con las mismas instrucciones y registros, pero esta vez el registro \$4 será inicializado a 28, para de esta forma dejar que el programa realice un bucle más y la instrucción LW cargue en el registro \$2 el valor FF que se aloja en la posición 28 de memoria principal.

D. Hexad	Etiqueta	Instrucción	Rd/Rt	Rs/Off	Rt/Imm	REGISTROS
00000300	LLOOP	ADDI	\$1	\$1	4	R0 00000000 R1 00000028
00000304		LW	\$2		0 \$1	R2 000000FF R3 000000FF
00000308		SW	\$3		24 \$1	R4 00000028 R5 00000000
0000030C		BNEQ	\$1	\$4	LOOP	R6 00000000 R7 00000000
00000310		NOP				R8 00000000 R9 00000000
00000314		NOP				R10 00000000 R11 00000000
00000318		NOP				R12 00000000 R13 00000000
0000031C		NOP				R14 00000000 R15 00000000
00000320		NOP				R16 00000000 R17 00000000
00000324		NOP				R18 00000000 R19 00000000
00000328		NOP				R20 00000000 R21 00000000
0000032C		NOP				R22 00000000 R23 00000000
00000330		NOP				R24 00000000 R25 00000000
00000334		NOP				R26 00000000 R27 00000000
00000338		NOP				R28 00000000 R29 00000000
0000033C		NOP				R30 00000000 R31 00000000
00000340		NOP				R32 00000000 RL0 00000000
00000344		NOP				
00000348		NOP				
0000034C		NOP				
00000350		NOP				
00000354		NOP				
00000358		NOP				

## Simulación 4 (Memoria Caché)

La simulación de éste nuevo código ensamblador se realizará con una jerarquía de tres niveles de caché con 32 bytes en L1, 256 bytes en L2 y 1kBytes en L3. Con mapeo de tipo directo:

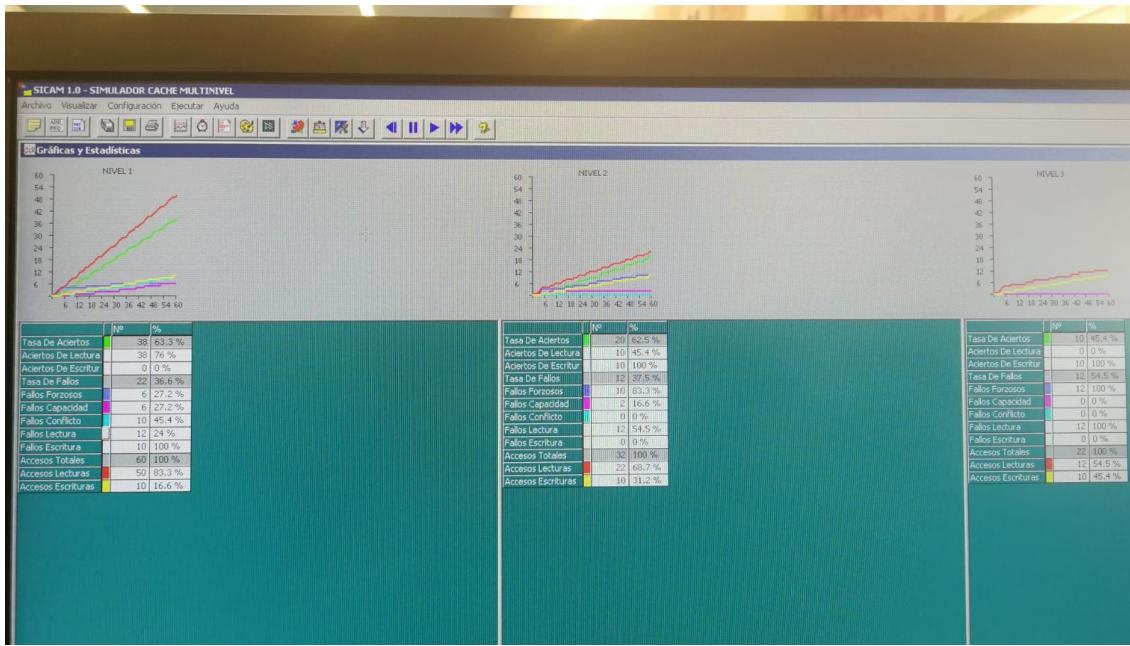


En esta nueva simulación podemos encontrar aparte del fallo forzoso visto en las anteriores simulaciones dos nuevos tipos de fallos, los cuales son el fallo de capacidad y el fallo de conflicto:

-El fallo de capacidad es el error que nos da la caché cuando no puede contener todos los bloques del programa en ejecución.

-El fallo de conflicto es el error que nos indica que se está sobrescribiendo un bloque de memoria caché con un dato relacionado directamente con ese bloque cuando aún la memoria caché no esta completamente llena. Este error se debe a que la ejecución es por mapeo directo (también podría ocurrir con mapeo asociativo por conjuntos).

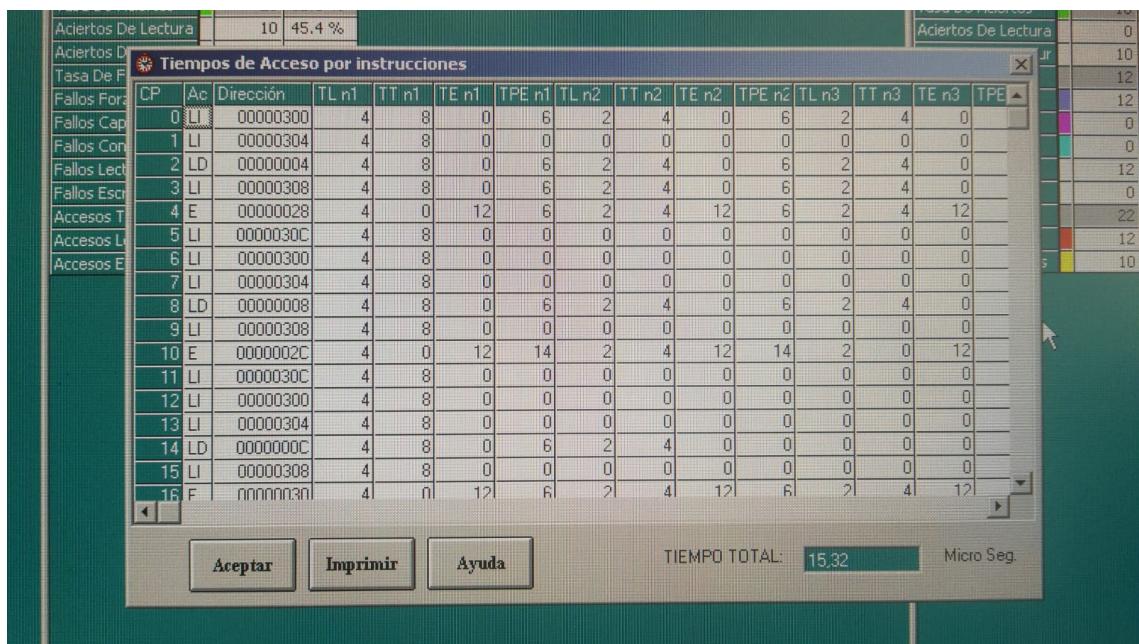
Durante la ejecución de este programa, podemos observar que L3 no ha obtenido ninguno de los fallos descritos anteriormente, mientras que L2 y L3 si, esto nos muestra que en una jerarquía de memoria el nivel de memoria caché es más eficiente si trabaja con varios niveles de memoria ya que de esta forma se puede retener más palabras de información a costa de una latencia minúscula en la ejecución de programas.



En las estadísticas de simulación, podemos destacar que tal y como pasó en los accesos a escritura en la L2 en la simulación anterior, en esta pasa lo mismo con L3, los aciertos en escritura se debe a que se recogen directamente desde el L1 de memoria.

Podemos observar también que la tasa de aciertos en L3 es notablemente inferior a la tasas de aciertos en L1 y L2, esto se debe a que mientras que L3 solo a conseguido aciertos en escritura, L1 y L2 también han recibido aciertos en lectura.

Los fallos de conflicto que hemos recibido exclusivamente de L1 se deben a que en esta simulación hemos configurado la memoria de L1 con muy poca capacidad para que de esta forma, el tipo de mapeo directo tenga que situar palabras en bloques de caché ya escritos y provocar estos fallos de lectura.



Como último análisis de esta simulación, destacaremos que está a sido la simulación con más latencia de todas las probadas. Esto se debe a los tamaños tan reducidos de memoria que se han configurado ya que provocan que se deba acceder a memoria principal múltiples veces para recuperar datos que la memoria caché ya había desechado anteriormente mediante la política de reemplazamiento FIFO en la que fue configurada.

## **Conclusiones:**

-Podemos concluir con esta serie de simulaciones que, la memoria caché es más eficiente si por cada bloque de palabras, se le otorga la capacidad de almacenar varias palabras como se pudo observar en las simulaciones dos y tres, donde el porcentaje de acierto es mucho mayor en la simulación tres que en la dos.

-La utilización de los distintos de niveles de caché no sería eficiente cuantos más niveles posee nuestra jerarquía de caché, ya que la eficiencia de estos se deben más a la cantidad de datos que está manejando el programa y a la política de reemplazamiento de las palabras en los bloques. Ya que por ejemplo un pequeño programa que un solo de nivel pueda almacenar va a ser más eficiente con ese solo nivel que con varios niveles.

-Asociativo por conjuntos es en los casos simulados, la manera de mapeo más eficiente ya que no provoca tantos fallos de conflicto como el mapeo directo ni gasta tanto tiempo en realizar comparaciones como el mapeo asociativo, sino que trabaja como el mapeo directo, pero en conjuntos de bloques para no realizar muchas comparaciones.

-La latencia se incrementa con el número de accesos a memoria principal que se realicen, por lo que es necesario adecuar la cantidad de niveles de caché empleados.

Tampoco es eficiente poner más de los necesarios, ya que los accesos a estos bloques secundarios innecesariamente también incrementa la latencia.