Memoria Cache en Ripes

Versión 1.0.0

Memoria Cache en Ripes

En la clase de Memoria Cache hemos aprendido

- Su principio de funcionamiento
- Métodos de asignación
- Ventajas y desventajas de cada método

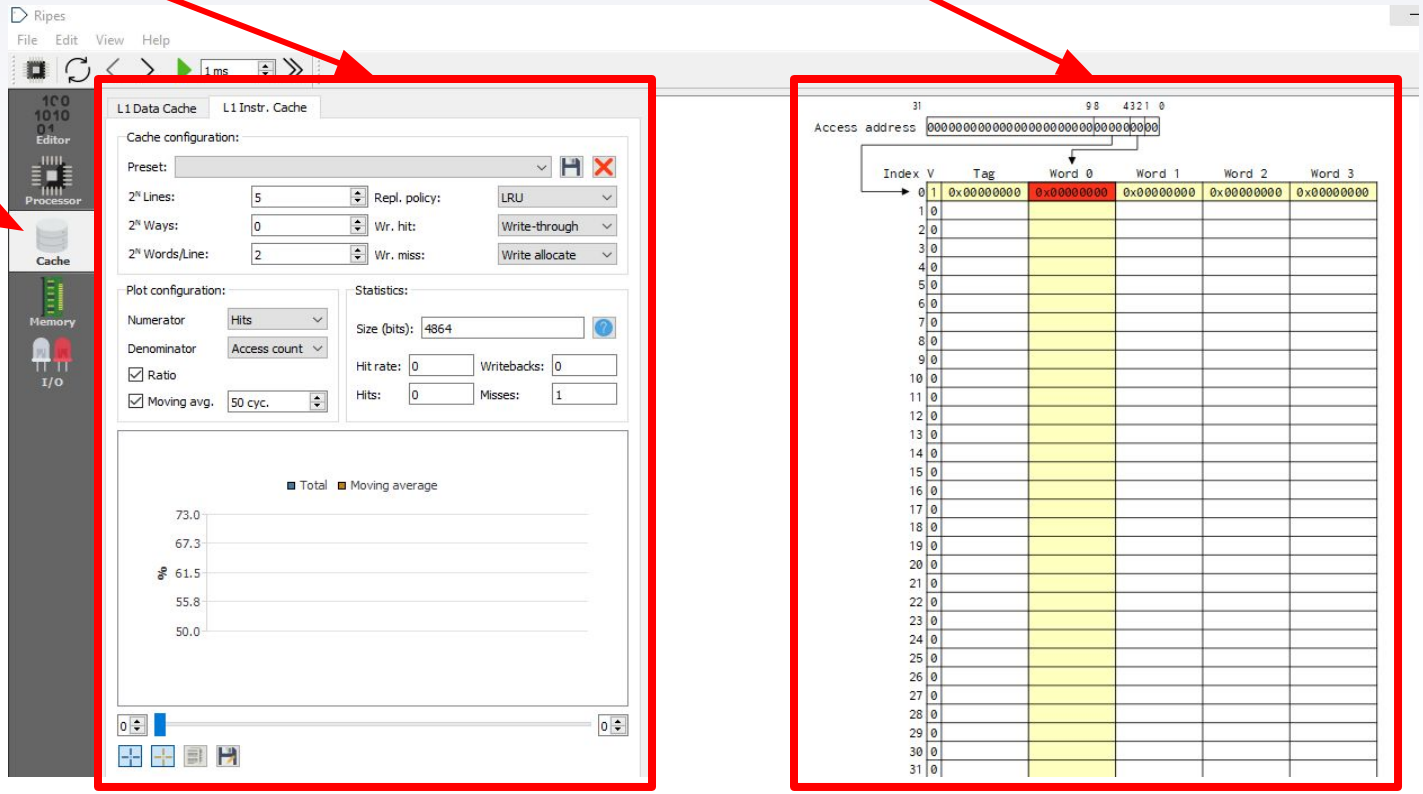
En esta clase mediante algunos ejemplos veremos cómo aprovechar Ripes para comprender cómo esto se ve reflejado en la ejecución de un programa.

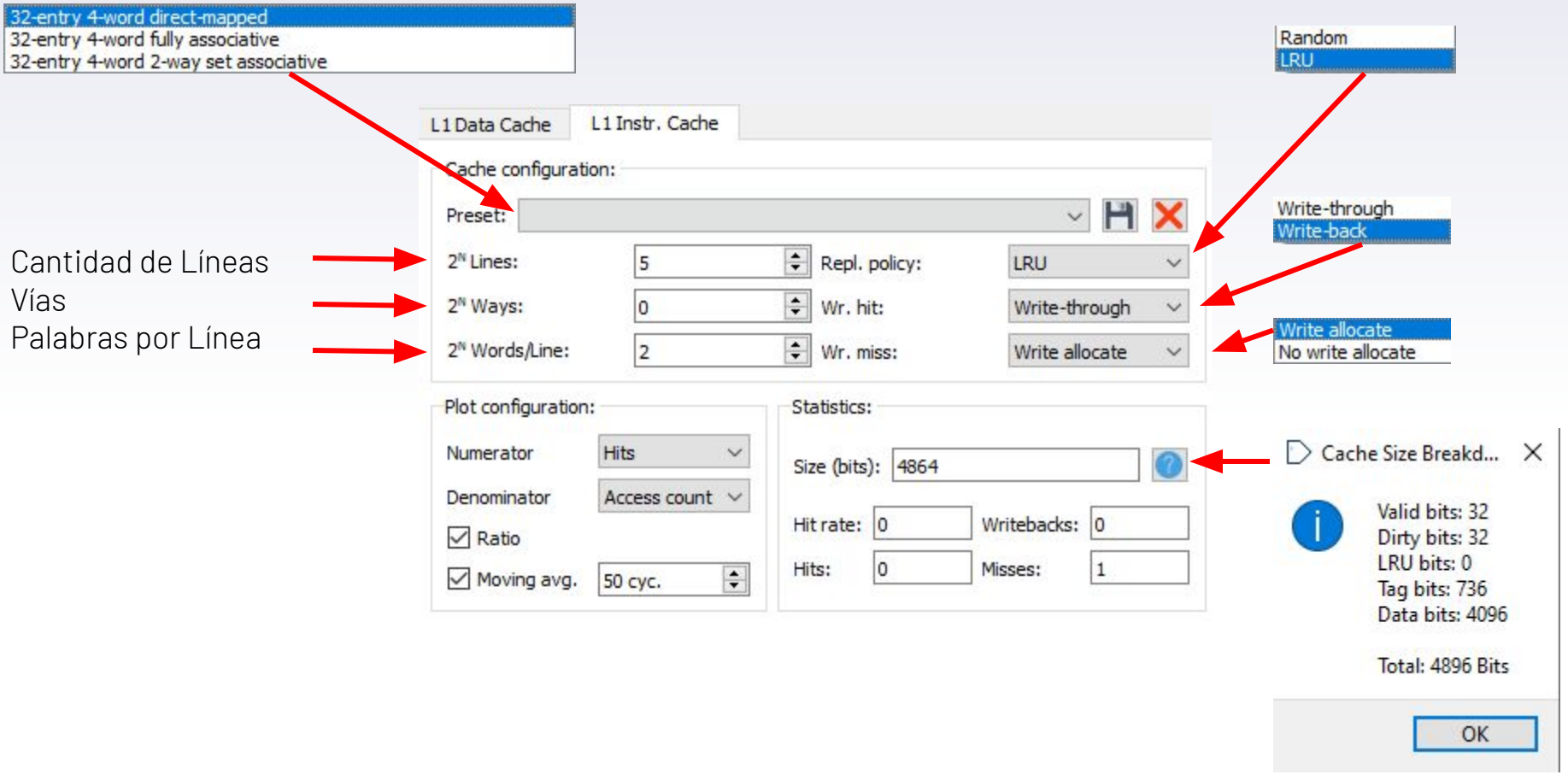
Al seleccionar cache aparecerá una pantalla como la siguiente

Configuración y estadística

Mapeo de la Cache configurada

Seleccionar Cache





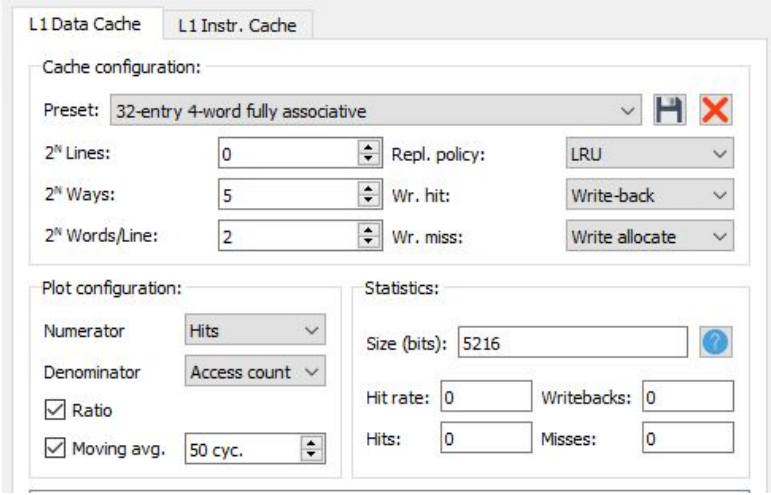
Cantidad de Líneas
Vías
Palabras por Línea

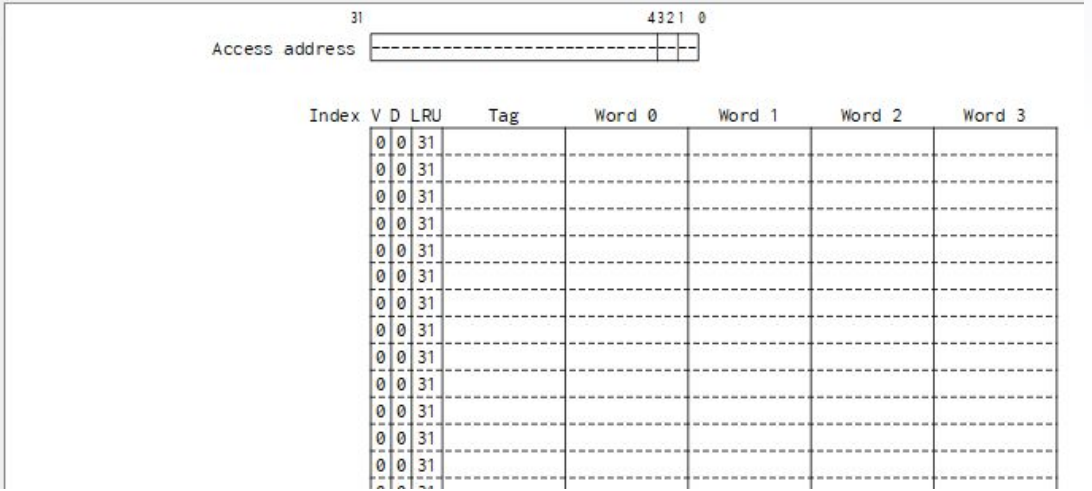
Cantidad de Líneas
Vías
Palabras por Línea

En la arquitectura RISC-V al igual que la memoria, la cache se encuentra dividida entre cache de datos y cache de instrucciones

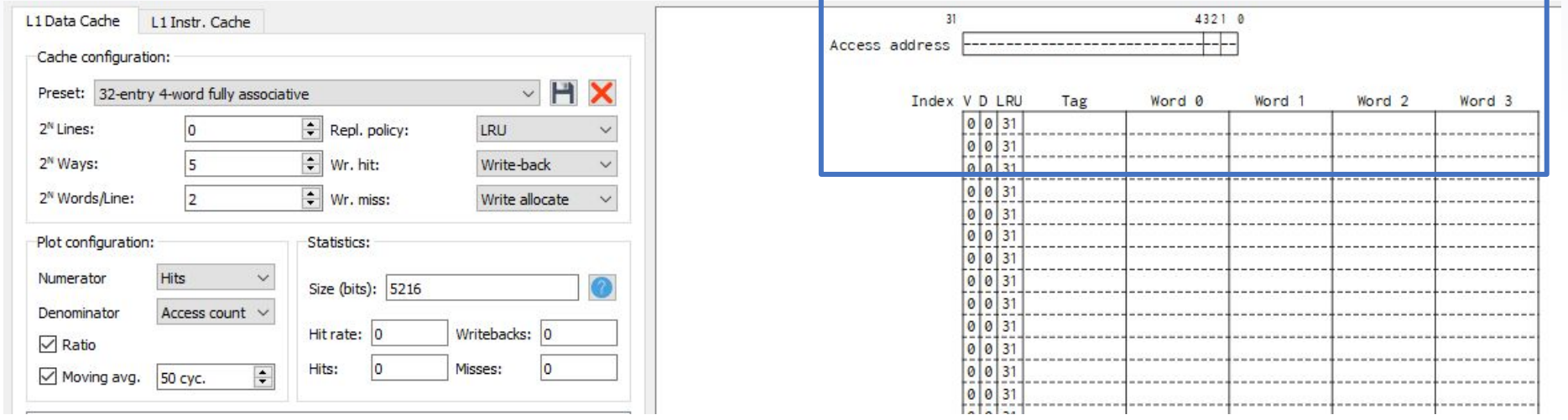
Asignación Asociativa

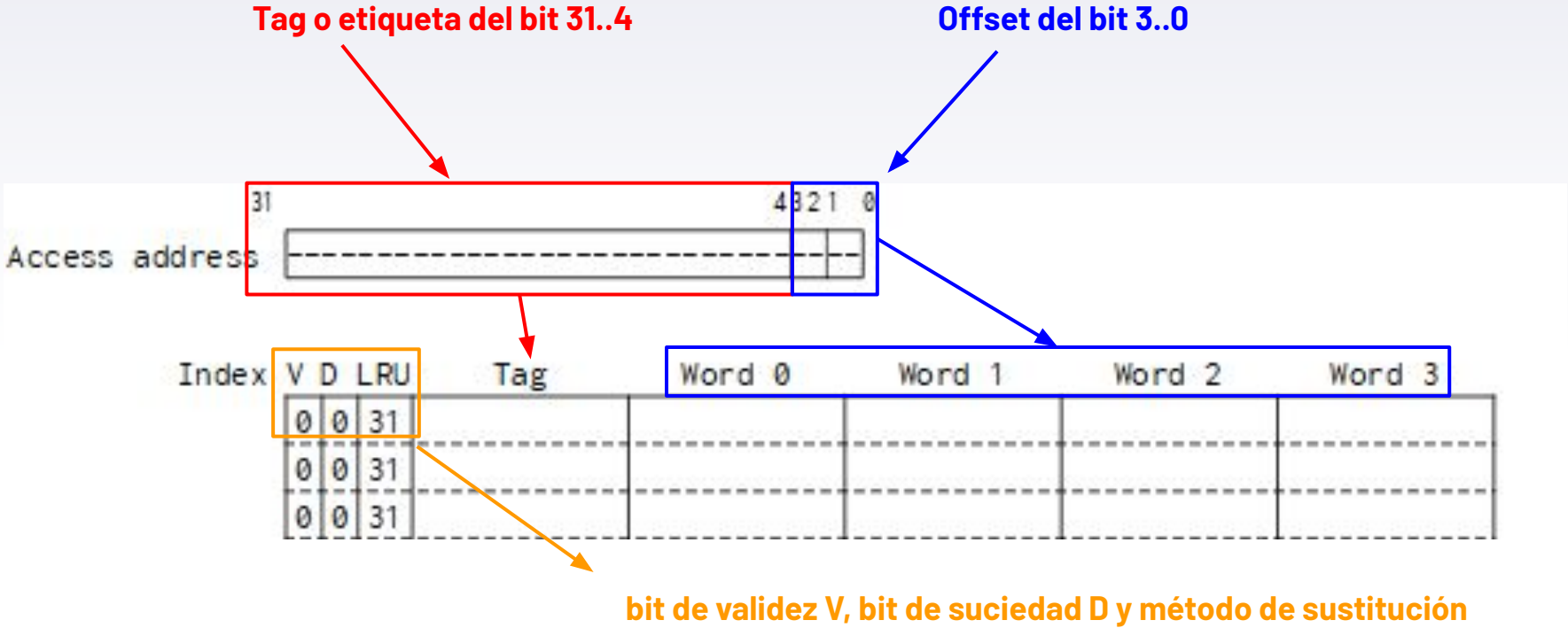
Veamos cómo queda configurada en Ripes una memoria cache asociativa de 32 líneas y 4 palabra por línea que es una de las opciones que esta pre-seteada





Entendamos el detalle





Tener en cuenta que los bit 0 y 1 del **Offset** corresponden a cada byte dentro de la palabra **Word**

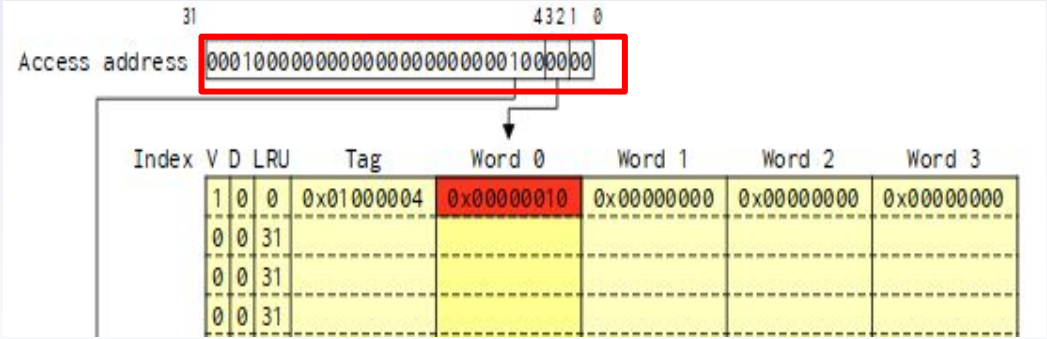
Analicemos un ejemplo

```
1  .data
2  vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3  vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4  cont: .word 16
5  .text
6  lui x7,0x10000
7  la x13,cont
8  lw x5, 0,x13
9  loop:
10     lw x10, 0,x7
11     addi x10,x10,1
12     sw x10, 0,x7
13     addi x5,x5,-1
14     addi x7,x7,+4
15     bne x0,x5,loop
16 fin:
17     beq x0,x0,fin
```

Un vector de 16 posiciones (o dos de 8 contiguos) al cual le vamos a sumar 1 (uno) a cada posición.

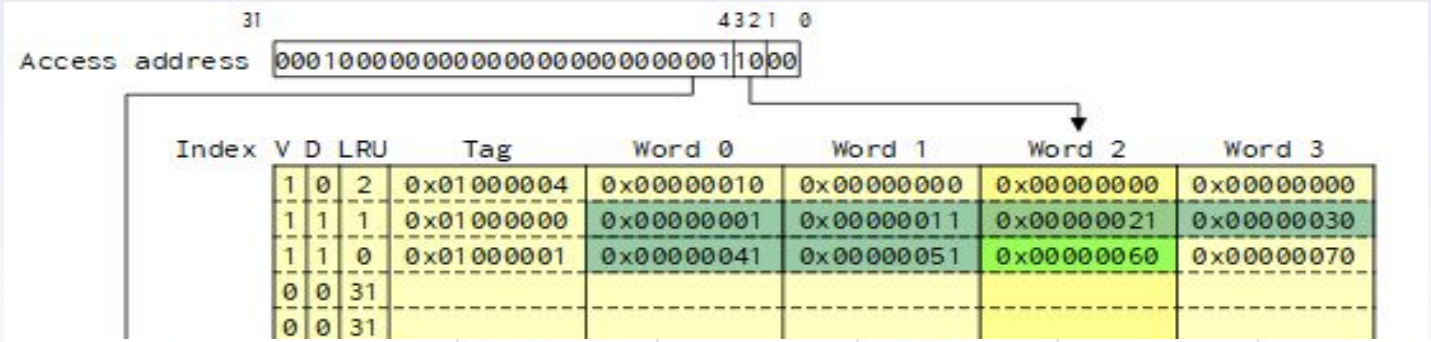
Vamos analizar la memoria cache de datos, tener en cuenta que esta comienza en 0x10000000

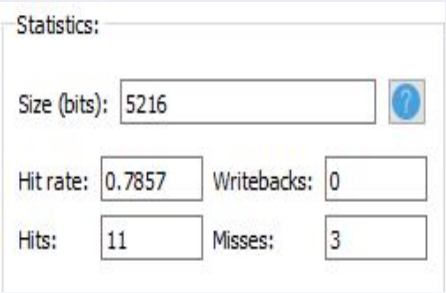
```
1 .data
2 vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3 vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4 cont: .word 16
5 .text
6 lui x7,0x10000
7 la x13,cont
8 lw x5, 0,x13
9 loop:
```



Vemos que el primer fallo se produce al cargar x5 con el valor contenido por cont que se encuentra en la posición de memoria 0x10000040, TAG 0x0100004, Offset 0x0. Vemos que el bit de validez es **V=1** indicando que esa línea de cache corresponde al entorno actual

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000048	X	X	X	X	X
0x10000044	X	X	X	X	X
0x10000040	0x00000010	0x10	0x00	0x00	0x00
0x1000003c	0x000000f0	0xf0	0x00	0x00	0x00
0x10000038	0x000000e0	0xe0	0x00	0x00	0x00

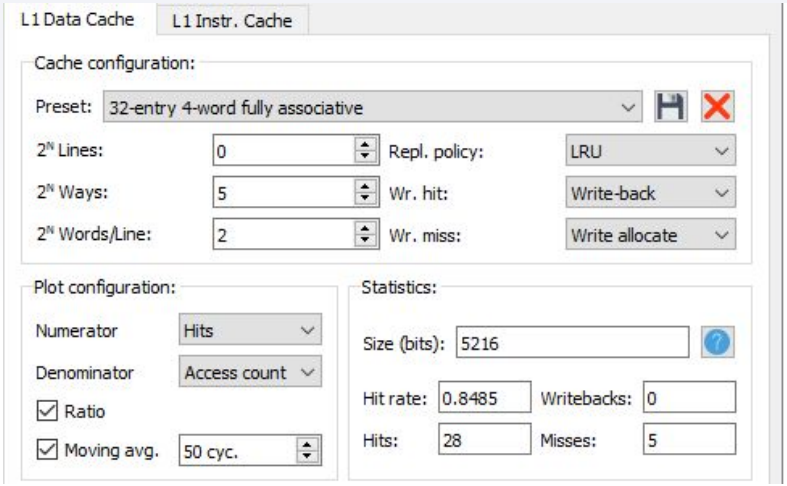


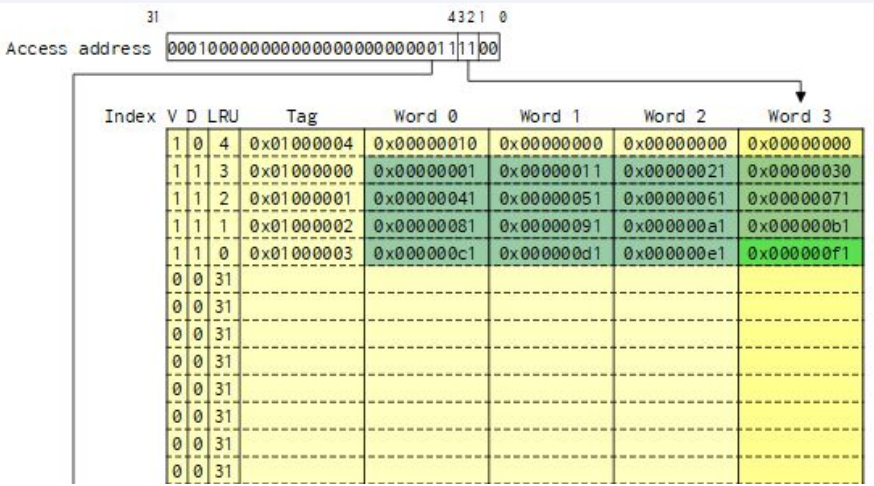


0x1000001c	0x00000070	0x70	0x00	0x00	0x00
0x10000018	0x00000060	0x60	0x00	0x00	0x00
0x10000014	0x00000051	0x51	0x00	0x00	0x00
0x10000010	0x00000041	0x41	0x00	0x00	0x00
0x1000000c	0x00000031	0x31	0x00	0x00	0x00
0x10000008	0x00000021	0x21	0x00	0x00	0x00
0x10000004	0x00000011	0x11	0x00	0x00	0x00
0x10000000	0x00000001	0x01	0x00	0x00	0x00

Se puede apreciar que a medida que el programa se ejecuta, los bit de validez y suciedad se modifican como así también el bit de LRU (Least Recently Used) indicando cual es la última línea a la que se accedió. En el cuadro de estadísticas se ve el Hit Rate y la cantidad de Hits y Misses al momento de la ejecución

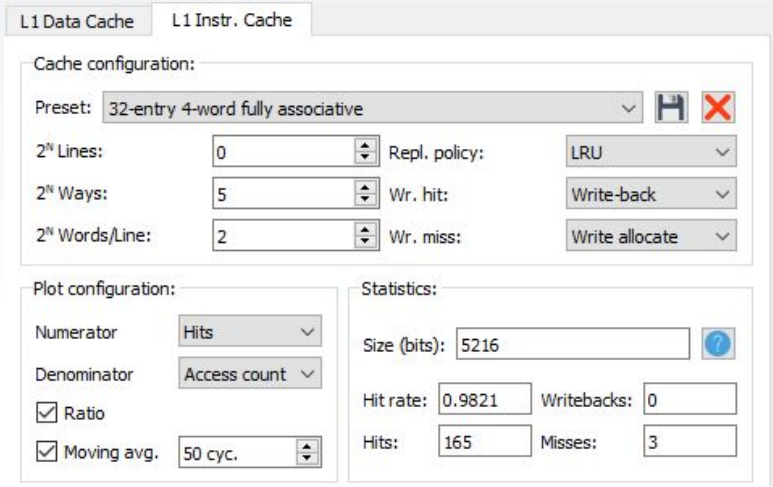
Al finalizar la ejecución del programa podremos evaluar la eficiencia de la configuración de la cache ayudándonos del cuadro de estadísticas

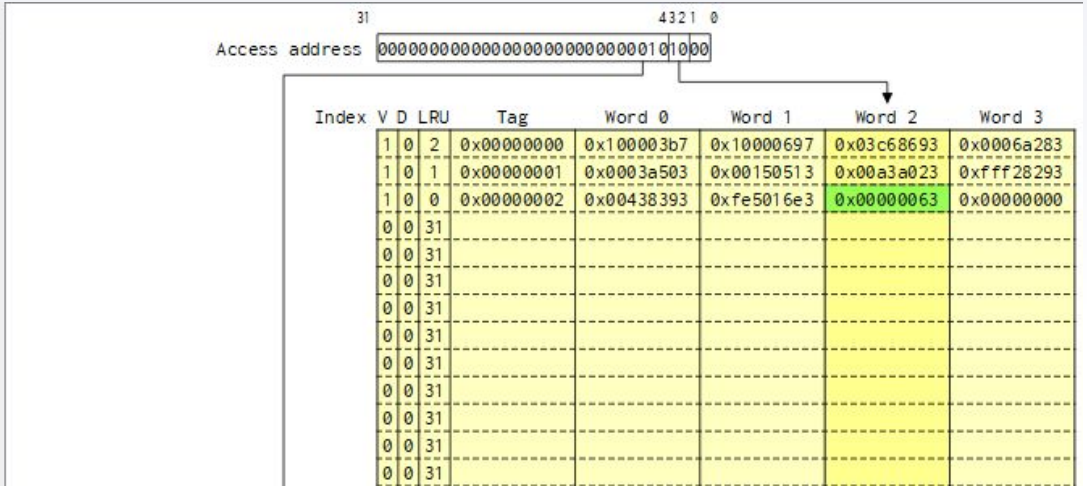




Podemos cambiar la cantidad de líneas y palabras por línea para analizar las diferencias en su funcionamiento y eficiencia

De la misma forma se puede analizar el comportamiento de la cache de instrucciones, y al finalizar la ejecución del programa podremos evaluar cómo se comportó la misma

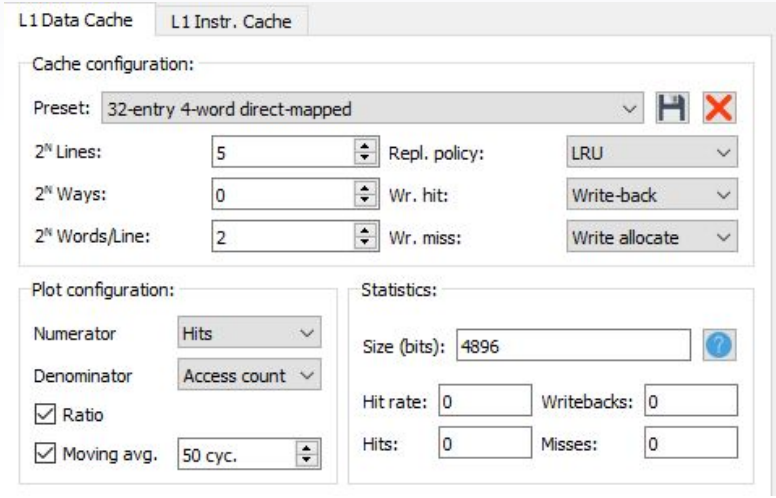


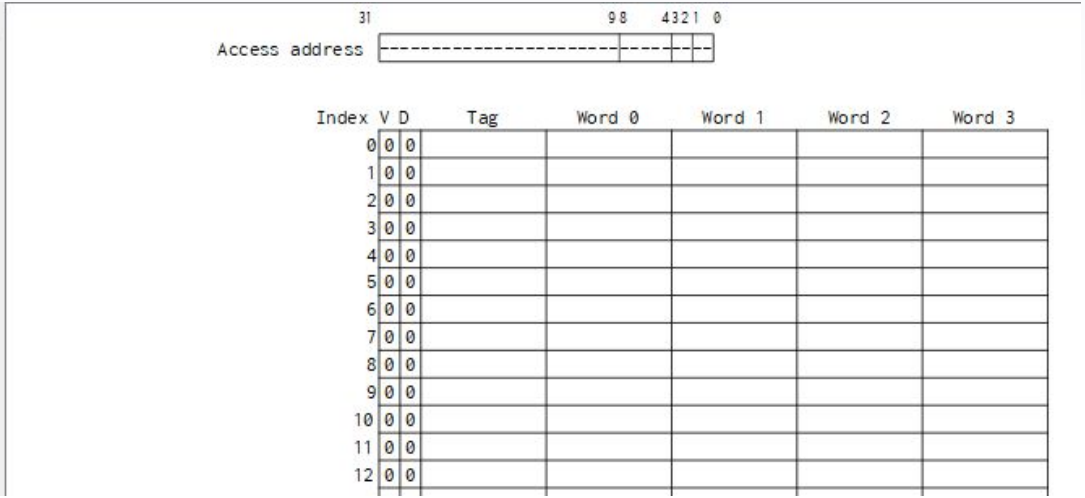


También podemos apreciar que el bit de suciedad **D** siempre se mantuvo en cero al no actualizarse ningún valor en la memoria de instrucciones

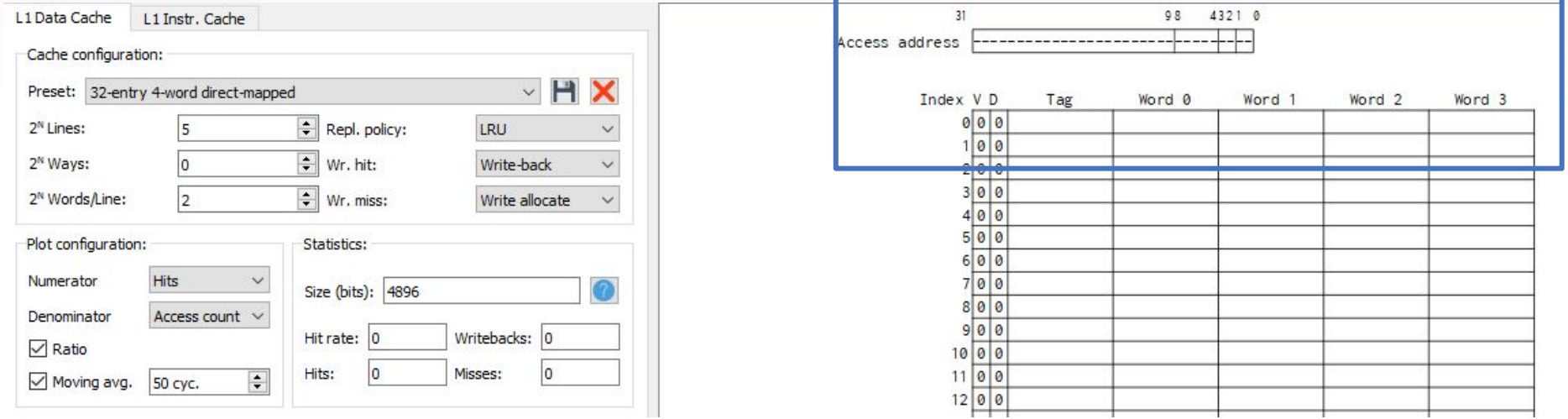
Asignación Directa

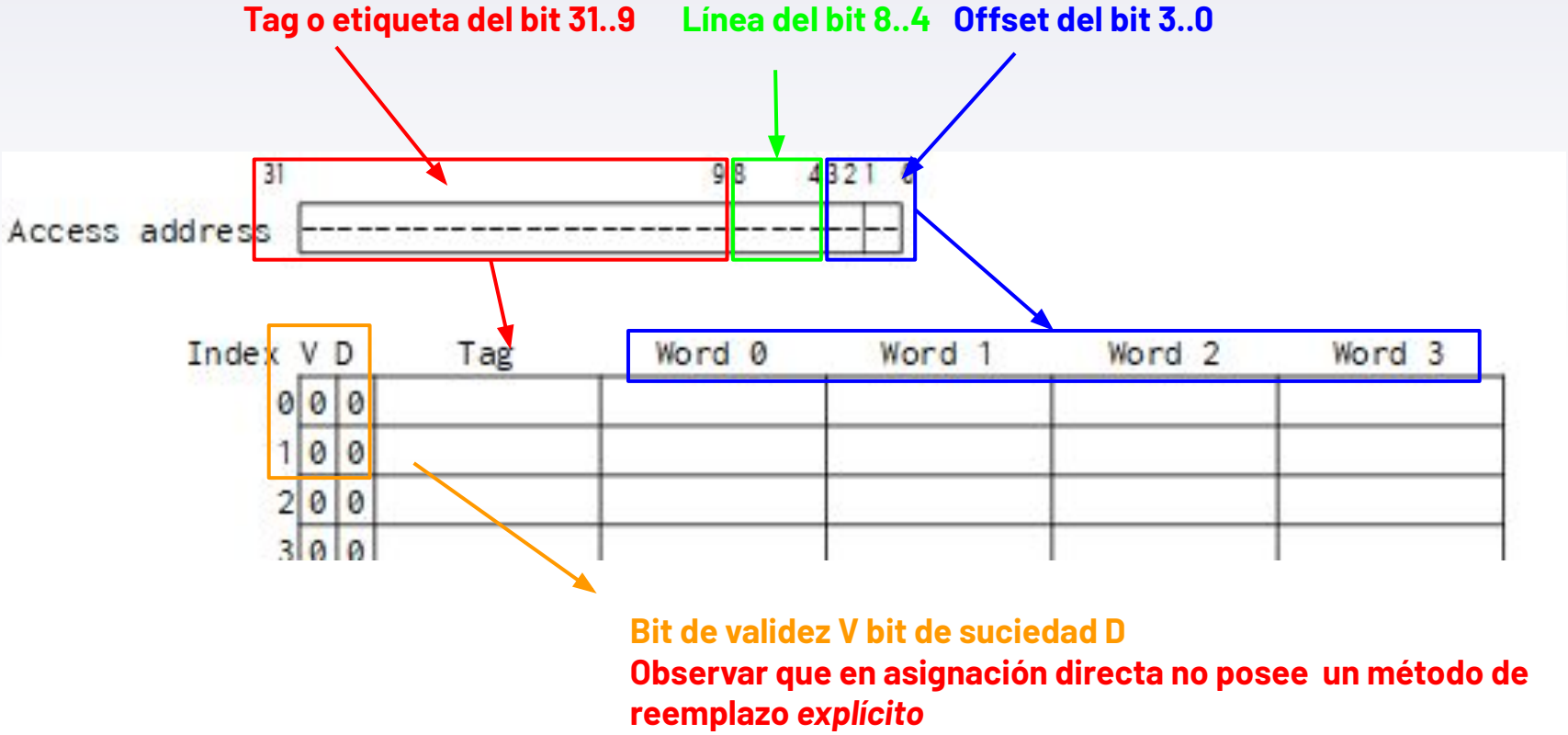
Veamos cómo queda configurada en Ripes una memoria cache con asignación directa de 32 líneas y 4 palabra por línea otra de las opciones pre-seteada





Entendamos el detalle





En asignación directa recordemos que se reduce el tamaño de la etiqueta y además tenemos el valor de la Línea. Tener en cuenta que los bit 0 y 1 del **Offset** corresponden a cada byte dentro de la palabra **Word**

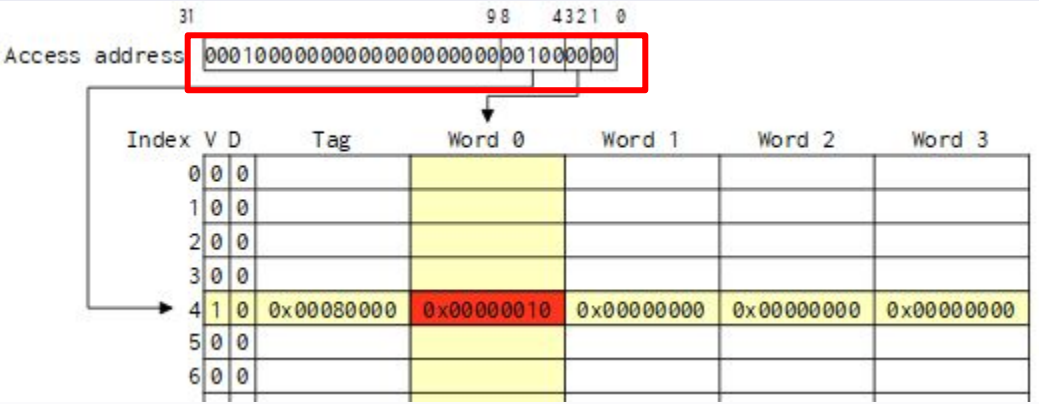
Analicemos el mismo ejemplo

```
1  .data
2  vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3  vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4  cont: .word 16
5  .text
6  lui x7,0x10000
7  la x13,cont
8  lw x5, 0,x13
9  loop:
10     lw x10, 0,x7
11     addi x10,x10,1
12     sw x10, 0,x7
13     addi x5,x5,-1
14     addi x7,x7,+4
15     bne x0,x5,loop
16 fin:
17     beq x0,x0,fin
```

Un vector de 16 posiciones (o dos de 8 contiguos) al cual le vamos a sumar 1 (uno) a cada posición.

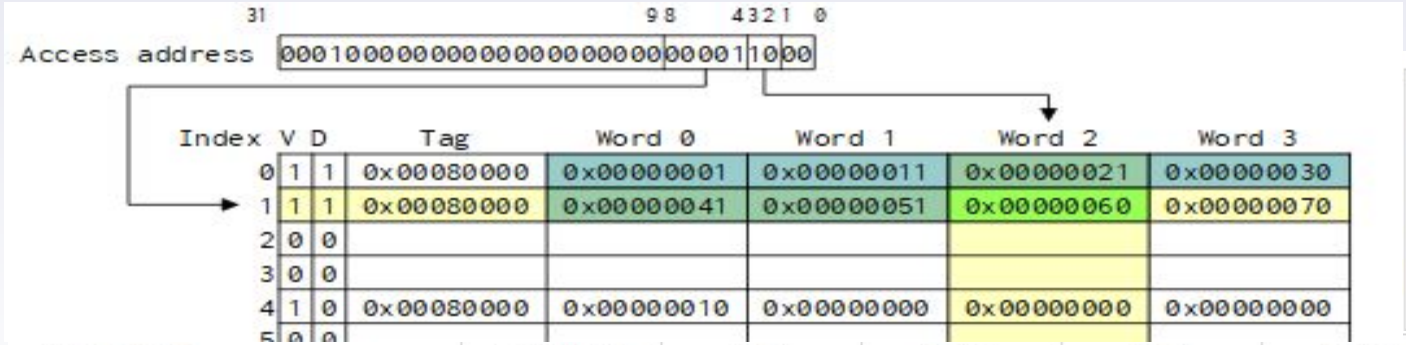
Vamos analizar la memoria cache de datos, tener en cuenta que esta comienza en 0x10000000

```
1 .data
2 vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3 vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4 cont: .word 16
5 .text
6 lui x7,0x10000
7 la x13,cont
8 lw x5, 0,x13
9 loop:
```



Vemos que el primer fallo, al igual que antes, se produce al cargar x5 con el valor que apunta cont en la posición de memoria 0x10000040, TAG 0x00080000, Línea 0x04, Offset 0x0. Observemos que el bit de validez es **V=1** indica que esa línea de cache corresponde al entorno actual.

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000048	X	X	X	X	X
0x10000044	X	X	X	X	X
0x10000040	0x00000010	0x10	0x00	0x00	0x00
0x1000003c	0x000000f0	0xf0	0x00	0x00	0x00
0x10000038	0x000000e0	0xe0	0x00	0x00	0x00



Statistics:

Size (bits): 4896

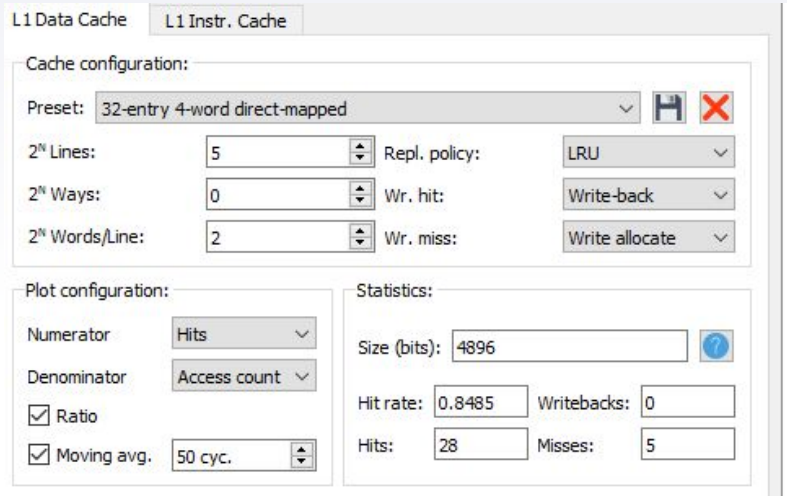
Hit rate: 0.7857 Writebacks: 0

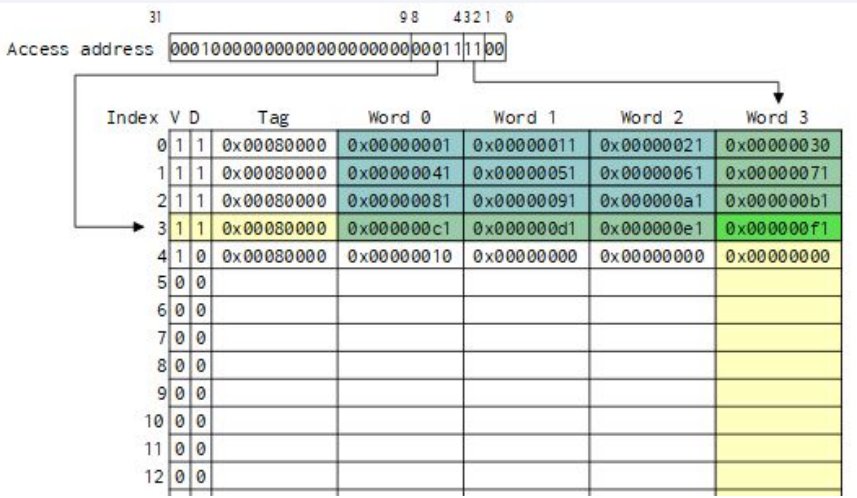
Hits: 11 Misses: 3

0x1000001c	0x00000070	0x70	0x00	0x00	0x00
0x10000018	0x00000060	0x60	0x00	0x00	0x00
0x10000014	0x00000051	0x51	0x00	0x00	0x00
0x10000010	0x00000041	0x41	0x00	0x00	0x00
0x1000000c	0x00000031	0x31	0x00	0x00	0x00
0x10000008	0x00000021	0x21	0x00	0x00	0x00
0x10000004	0x00000011	0x11	0x00	0x00	0x00
0x10000000	0x00000001	0x01	0x00	0x00	0x00

Se puede apreciar que a medida que el programa se ejecuta los bit de validez y suciedad se modifican. En el cuadro de estadísticas se ve el Hit Rate y la cantidad de Hits y Misses al momento de la ejecución

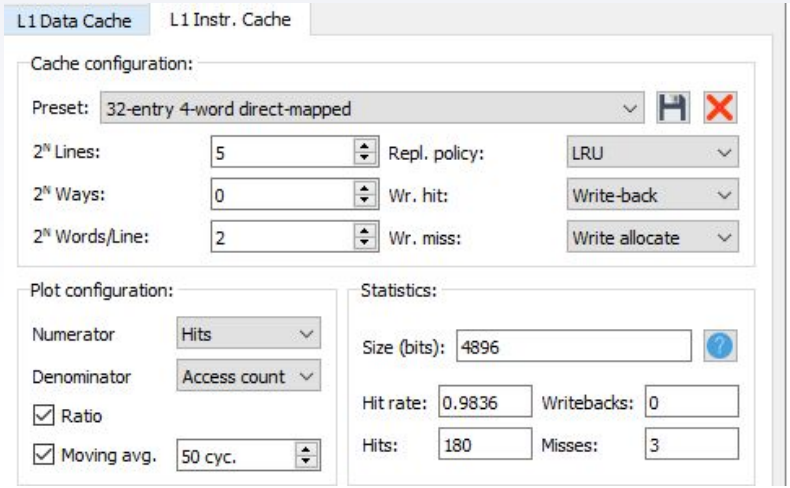
Al finalizar la ejecución del programa podremos evaluar la eficiencia de la configuración de la cache ayudándonos del cuadro de estadísticas

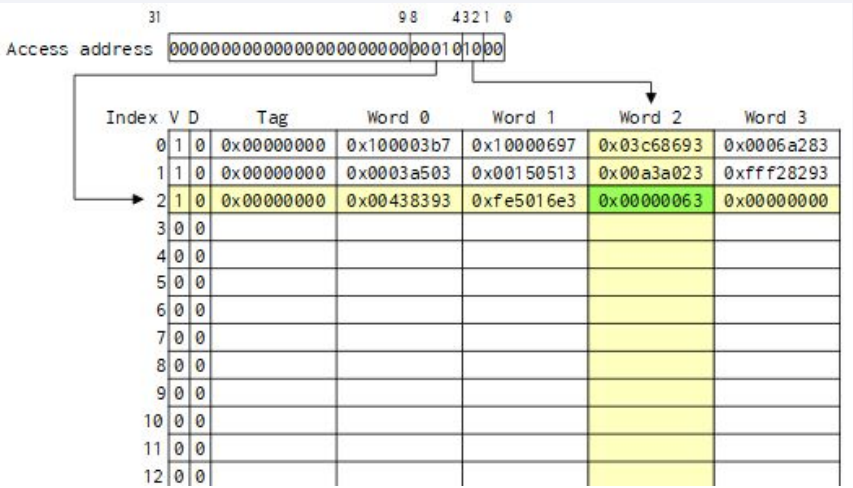




Podemos analizar la diferencia entre un método y otro comparando los resultados, pudiendo apreciar que no existen diferencias en este programa.
Se puede cambiar la cantidad de líneas y palabra por línea para analizar las diferencias en su funcionamiento y eficiencia

De la misma forma se puede analizar el comportamiento de la cache de instrucciones, y al finalizar la ejecución del programa podremos evaluar cómo se comportó la misma

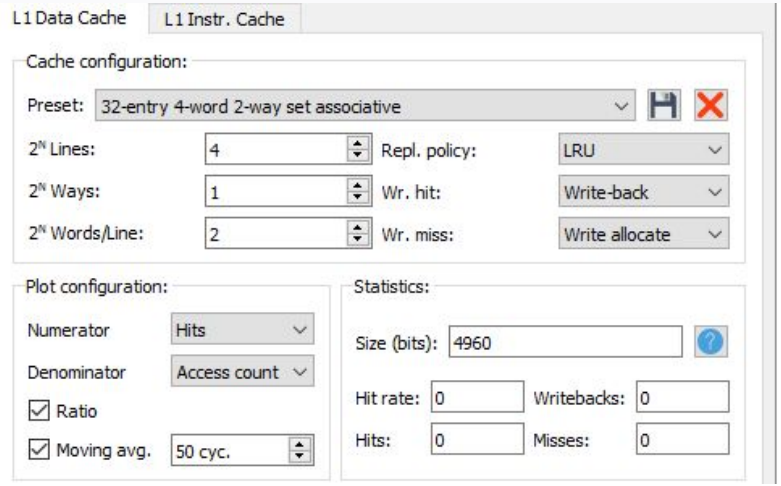


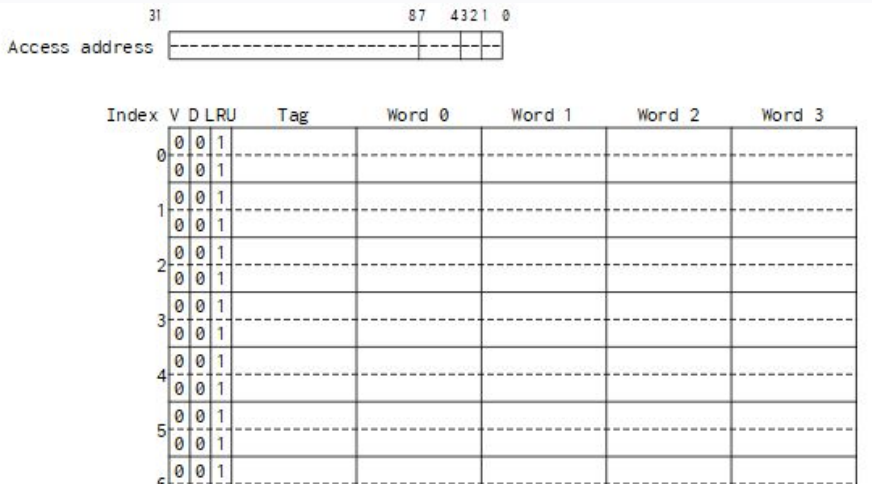


Podemos apreciar que el bit de suciedad **D** siempre se mantuvo en cero al no actualizarse ningún valor en la memoria de instrucciones

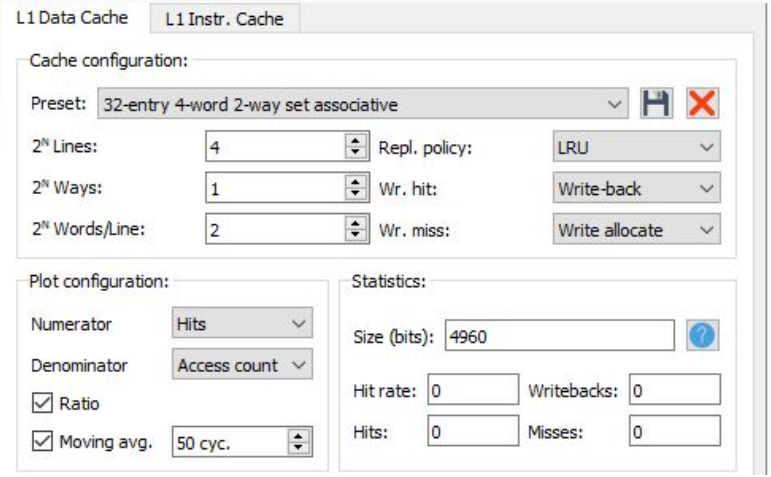
Asignación Asociativa por Conjuntos

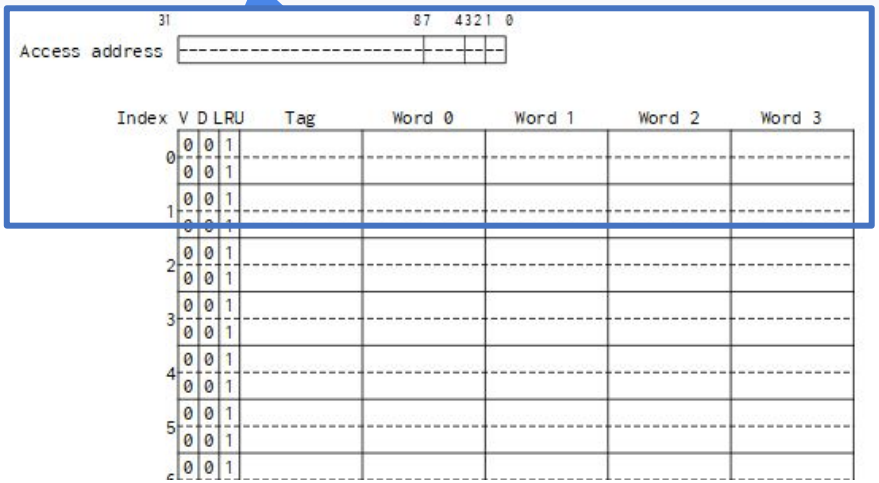
Veamos cómo queda configurada en Ripes una memoria cache con asignación asociativa por conjuntos de 2 vías y 4 palabra por línea, otra de las opciones pre-seteadas

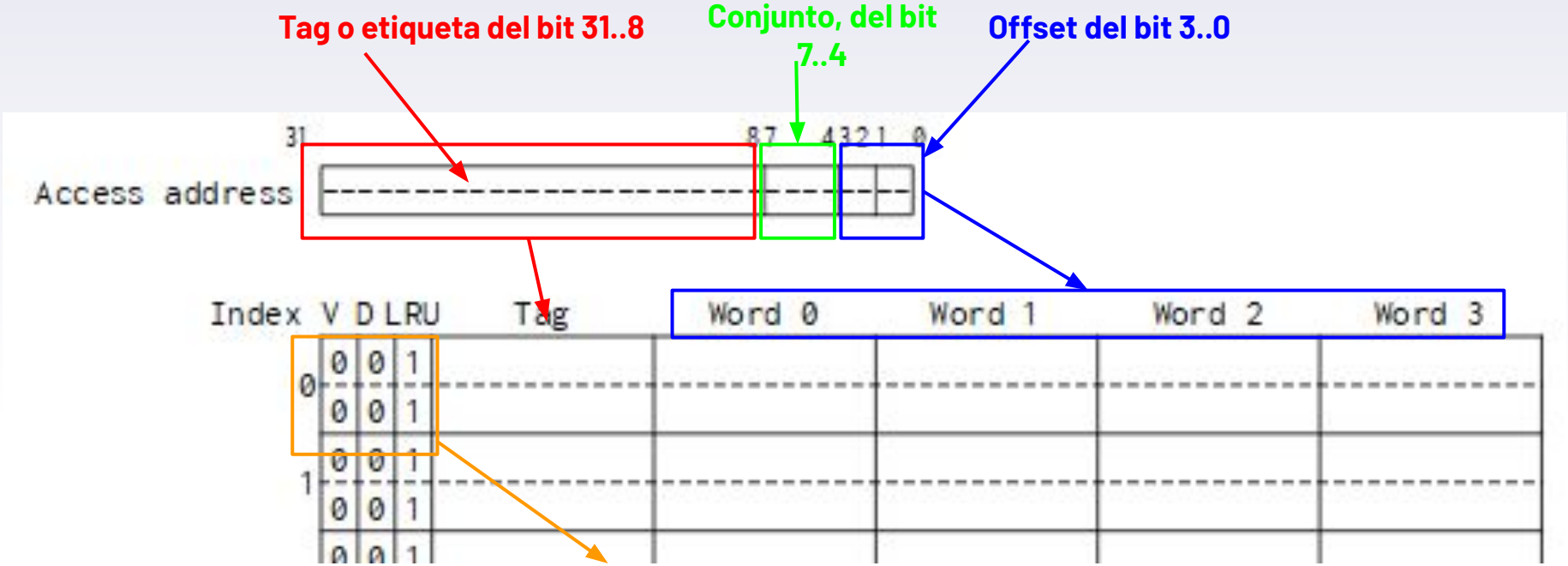




Entendamos el detalle







En asignación asociativa por conjunto recordemos que se incrementa el tamaño de la etiqueta dependiendo de la cantidad de vías y por ende el tamaño de la Línea se reduce.
Tener en cuenta que los bit 0 y 1 del **Offset** corresponden a cada byte dentro de la palabra **Word**

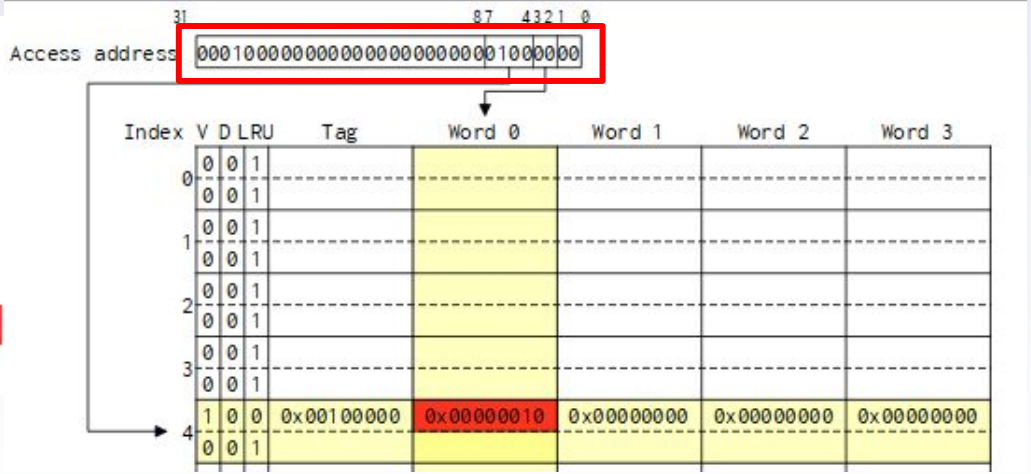
Analicemos el mismo ejemplo

```
1  .data
2  vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3  vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4  cont: .word 16
5  .text
6  lui x7,0x10000
7  la x13,cont
8  lw x5, 0,x13
9  loop:
10     lw x10, 0,x7
11     addi x10,x10,1
12     sw x10, 0,x7
13     addi x5,x5,-1
14     addi x7,x7,+4
15     bne x0,x5,loop
16 fin:
17     beq x0,x0,fin
```

Un vector de 16 posiciones (o dos de 8 contiguos) al cual le vamos a sumar 1 (uno) a cada posición.

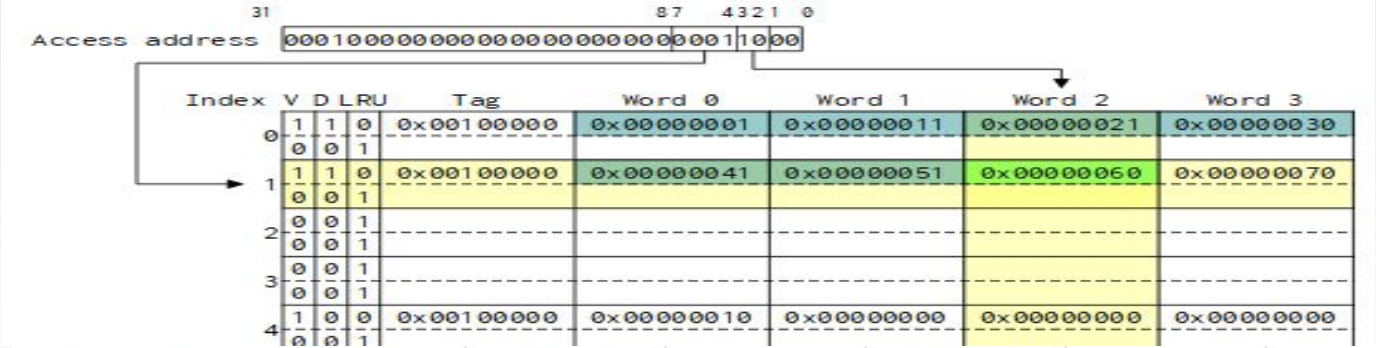
Vamos analizar la memoria cache de datos, tener en cuenta que esta comienza en 0x10000000

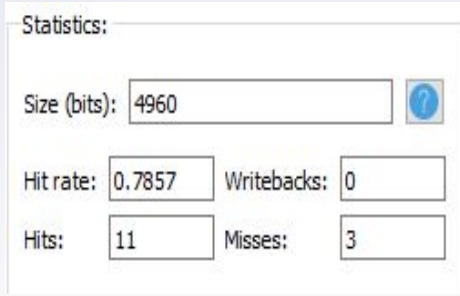
```
1 .data
2 vector1: .word 0x0,0x10,0x20,0x30,0x40,0x50,0x60,0x70,
3 vector2: .word 0x80,0x90,0xa0,0xb0,0xc0,0xd0,0xe0,0xf0
4 cont: .word 16
5 .text
6 lui x7,0x10000
7 la x13,cont
8 lw x5, 0,x13
9 loop:
```



Vemos que el primer fallo, al igual que antes, se produce al cargar x5 con el valor del contador en la posición de memoria 0x10000040, TAG 0x00100000, Línea 0x4, Offset 0x0 vemos que el bit de validez es **V=1** indicando que esa línea de cache corresponde al entorno actual

Address	Word	Byte 0	Byte 1	Byte 2	Byte 3
0x10000048	X	X	X	X	X
0x10000044	X	X	X	X	X
0x10000040	0x00000010	0x10	0x00	0x00	0x00
0x1000003c	0x000000f0	0xf0	0x00	0x00	0x00
0x10000038	0x000000e0	0xe0	0x00	0x00	0x00

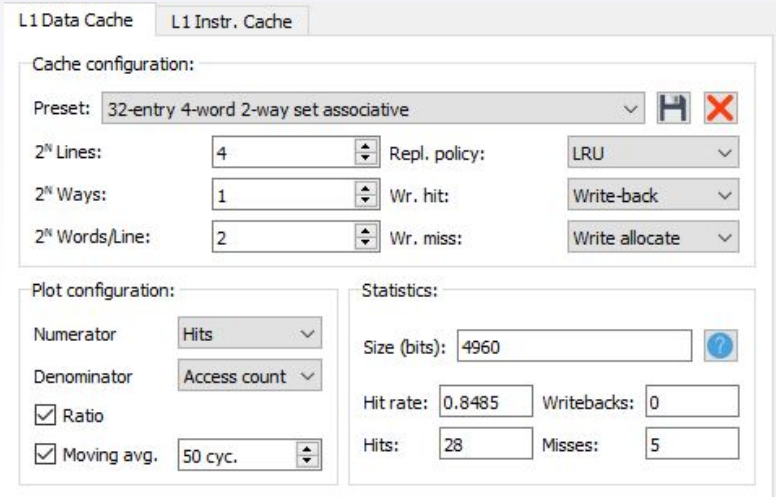


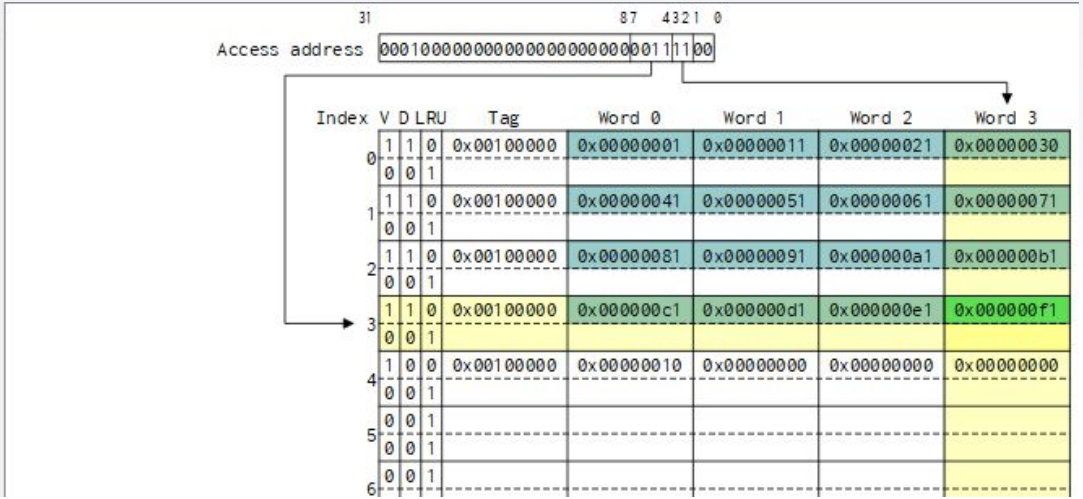


0x1000001c	0x00000070	0x70	0x00	0x00	0x00
0x10000018	0x00000060	0x60	0x00	0x00	0x00
0x10000014	0x00000051	0x51	0x00	0x00	0x00
0x10000010	0x00000041	0x41	0x00	0x00	0x00
0x1000000c	0x00000031	0x31	0x00	0x00	0x00
0x10000008	0x00000021	0x21	0x00	0x00	0x00
0x10000004	0x00000011	0x11	0x00	0x00	0x00
0x10000000	0x00000001	0x01	0x00	0x00	0x00

Se puede apreciar que a medida que el programa se ejecuta los bit de validez y suciedad se modifican como así también el bit LRU. En el cuadro de estadísticas se ve el Hit Rate y la cantidad de Hits y Misses al momento de la ejecución.

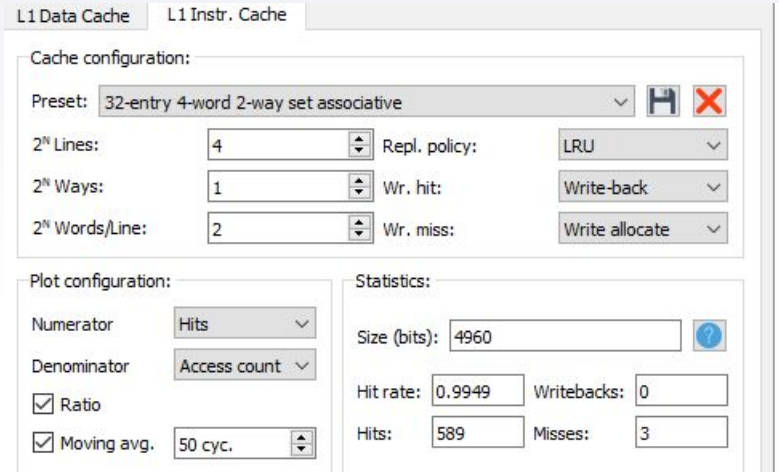
Al finalizar la ejecución del programa podremos evaluar la eficiencia de la configuración de la cache ayudándonos del cuadro de estadísticas

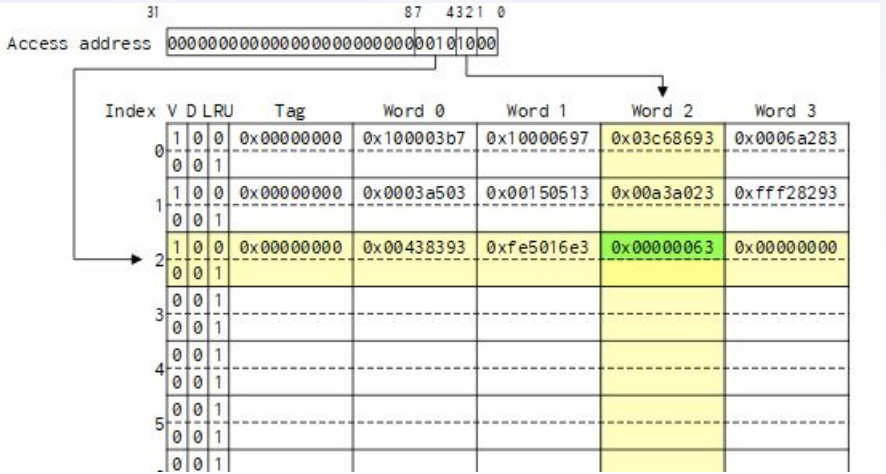




Podemos analizar las diferencias entre los distintos métodos de asignación comparando los resultados, pudiéndose apreciar que no existen diferencias en el ejemplo dado.
Se pueden cambiar la cantidad de líneas y palabras por línea y vías para analizar las diferencias en su funcionamiento y eficiencia
En este caso podemos apreciar la diferencia existente respecto a la asignación directa de la forma que se ubican los datos en la cache.

De la misma forma se puede analizar el comportamiento de la cache de instrucciones, y al finalizar la ejecución del programa podremos evaluar cómo se comportó la misma





Podemos observar que el bit de suciedad **D** siempre se mantuvo en cero al no actualizarse ningún valor en la memoria de instrucciones, como así también la disposición de los datos.

¿ Cuál es el mejor método de asignación ?

A esta altura ya podrán inferir que no existe un método mejor que otro.

Solo podemos afirmar que cada uno tiene sus ventajas y/o desventajas.

Dependerá no solo del tamaño de la cache y su método de asignación sino también de la optimización en la utilización de la memoria. Es decir donde se almacenan los datos y cómo se acceden a ellos tiene un impacto en la eficiencia del uso de la cache.

Veamos qué sucede si.....

Tenemos dos vectores de 32 words cada uno

V1 - ubicado a partir de la Dirección 0x10000000
V2 - ubicado a partir de la Dirección 0x20000000

Y queremos intercambiar los valores de V1 en V2 y los de V2 en V1

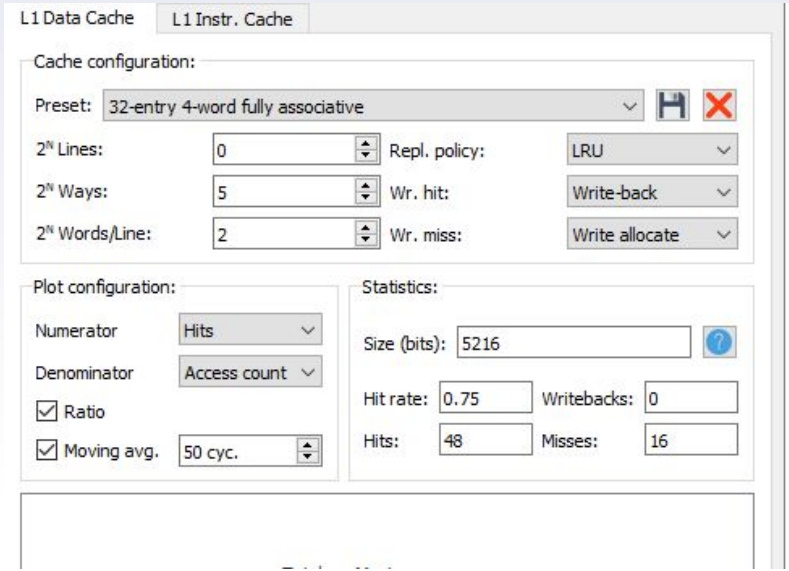
Configurando la memoria cache en cada uno de los formatos pre-seteados en Ripes

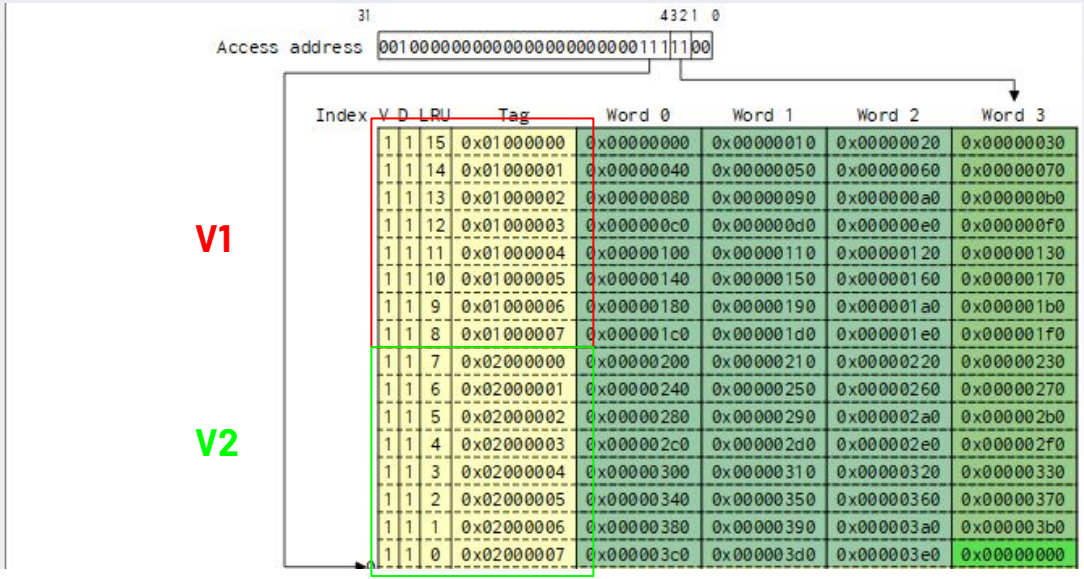
Generamos los vectores de 32 posiciones
uno apuntado por x7 y otro por x8

```
1  lui x7,0x10000
2  addi x10,x10,32
3  add x6,x6,x0
4  carga1:
5      sw x6, 0,x7
6      addi x7,x7,4
7      addi x6,x6,0x10
8      addi x10,x10,-1
9      bne x0,x10,carga1
10 lui x8,0x20000
11 addi x10,x10,32
12 carga2:
13       sw x6, 0,x8
14     addi x8,x8,4
15     addi x6,x6,0x10
16     addi x10,x10,-1
17     bne x0,x10,carga2
18
```

y realizamos el intercambio de sus valores

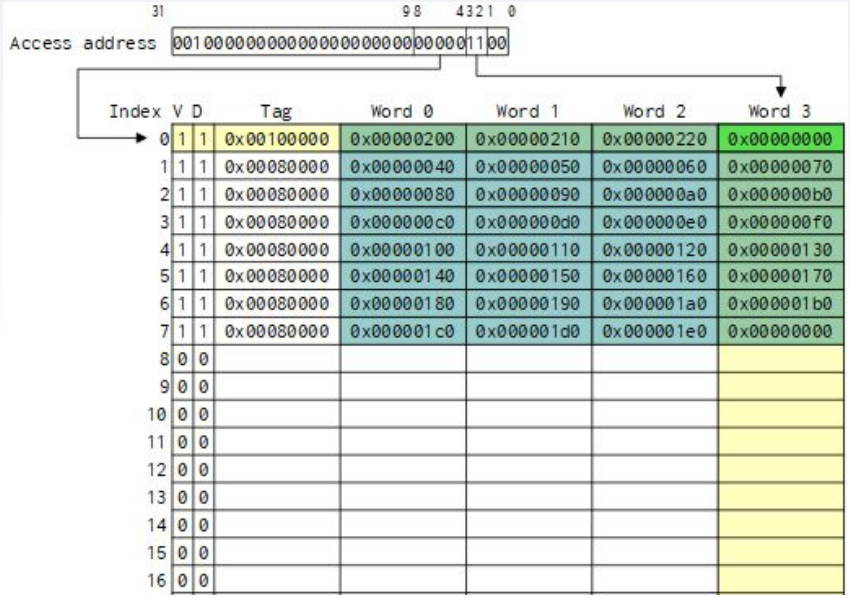
```
18
19 addi x7,x7,-128
20 addi x8,x8,-128
21 addi x10,x0,32
22
23 intercambio:
24     lw x11,0,x7
25     lw x12,0,x8
26     sw x12,0,x7
27     sw x11,0,x8
28     addi x7,x7,4
29     addi x8,x8,4
30     addi x10,x10,-1
31     bne x0,x10,intercambio
32 fin:
33     beq x0,x0,fin
34
```





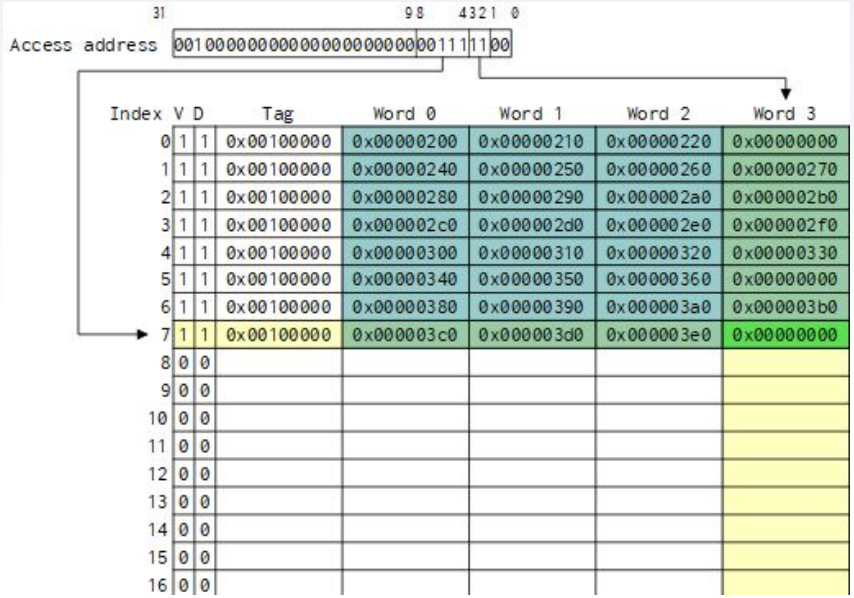
Vemos que al finalizar la carga, ambos vectores se encuentran cargados en la memoria cache si la asignación es asociativa

Que sucede en asignación directa



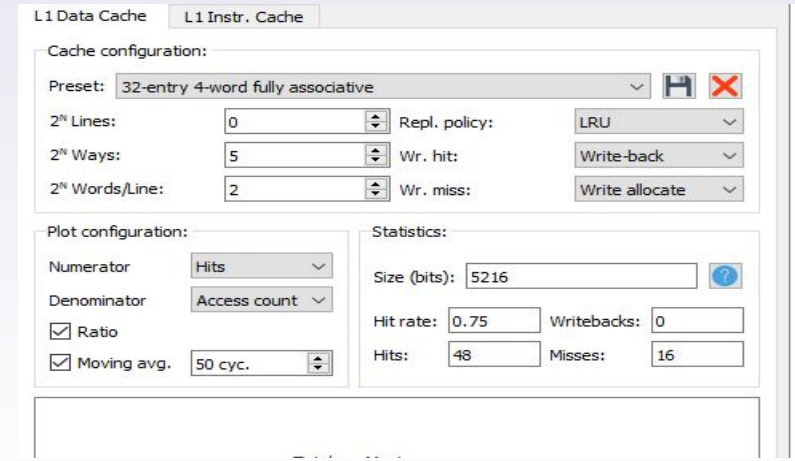
Index	V	D	Tag	Word 0	Word 1	Word 2	Word 3
0	1	1	0x00100000	0x00000200	0x00000210	0x00000220	0x00000000
1	1	1	0x00080000	0x00000040	0x00000050	0x00000060	0x00000070
2	1	1	0x00080000	0x00000080	0x00000090	0x000000a0	0x000000b0
3	1	1	0x00080000	0x000000c0	0x000000d0	0x000000e0	0x000000f0
4	1	1	0x00080000	0x00000100	0x00000110	0x00000120	0x00000130
5	1	1	0x00080000	0x00000140	0x00000150	0x00000160	0x00000170
6	1	1	0x00080000	0x00000180	0x00000190	0x000001a0	0x000001b0
7	1	1	0x00080000	0x000001c0	0x000001d0	0x000001e0	0x00000000
8	0	0					
9	0	0					
10	0	0					
11	0	0					
12	0	0					
13	0	0					
14	0	0					
15	0	0					
16	0	0					

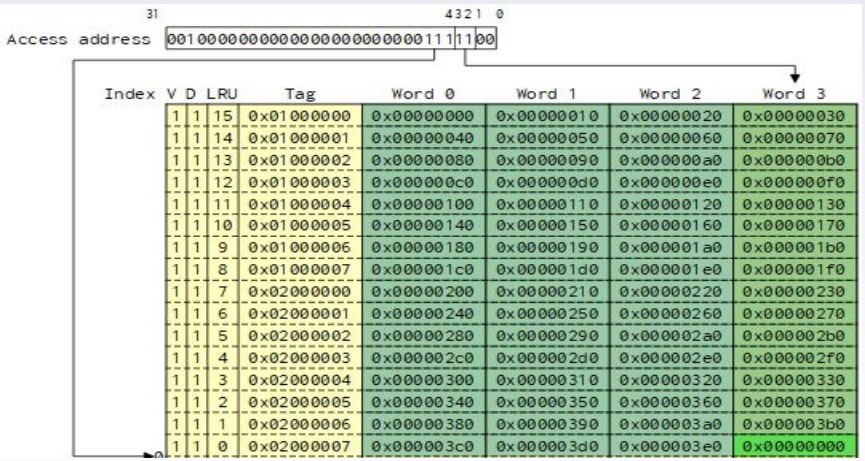
Vemos que al comenzar a cargar el segundo vector este corresponde a la misma línea por lo que se reemplaza el contenido anterior provocando un fallo

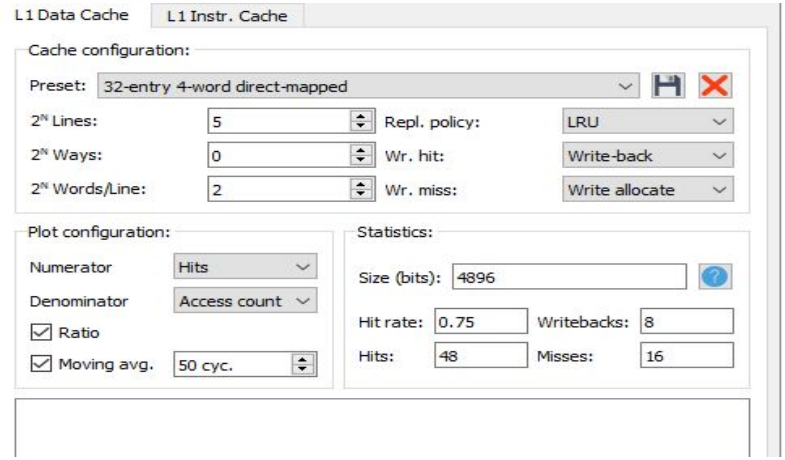


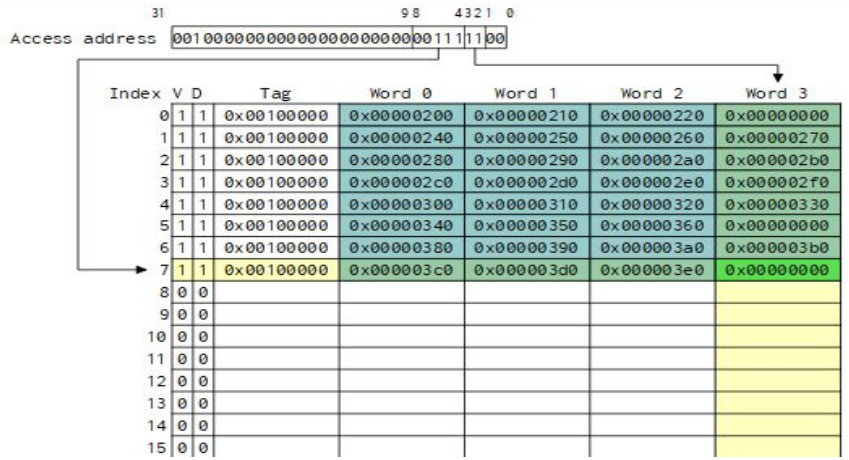
Index	V	D	Tag	Word 0	Word 1	Word 2	Word 3
0	1	1	0x00100000	0x00000200	0x00000210	0x00000220	0x00000000
1	1	1	0x00100000	0x00000240	0x00000250	0x00000260	0x00000270
2	1	1	0x00100000	0x00000280	0x00000290	0x000002a0	0x000002b0
3	1	1	0x00100000	0x000002c0	0x000002d0	0x000002e0	0x000002f0
4	1	1	0x00100000	0x00000300	0x00000310	0x00000320	0x00000330
5	1	1	0x00100000	0x00000340	0x00000350	0x00000360	0x00000000
6	1	1	0x00100000	0x00000380	0x00000390	0x000003a0	0x000003b0
7	1	1	0x00100000	0x000003c0	0x000003d0	0x000003e0	0x00000000
8	0	0					
9	0	0					
10	0	0					
11	0	0					
12	0	0					
13	0	0					
14	0	0					
15	0	0					
16	0	0					

Al finalizar la carga solo los valores correspondientes al vector 2 permanecen en la memoria cache









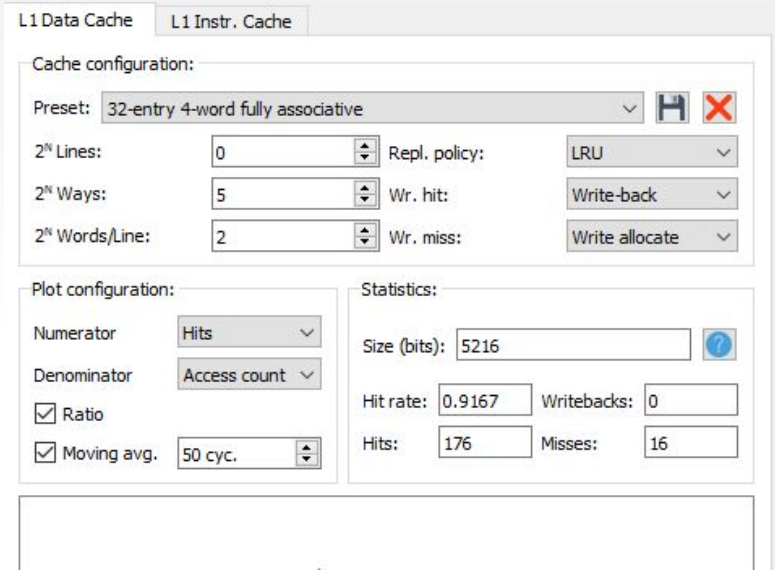
Podemos apreciar que la cantidad de fallos en esta etapa fue la misma en ambos casos, pero

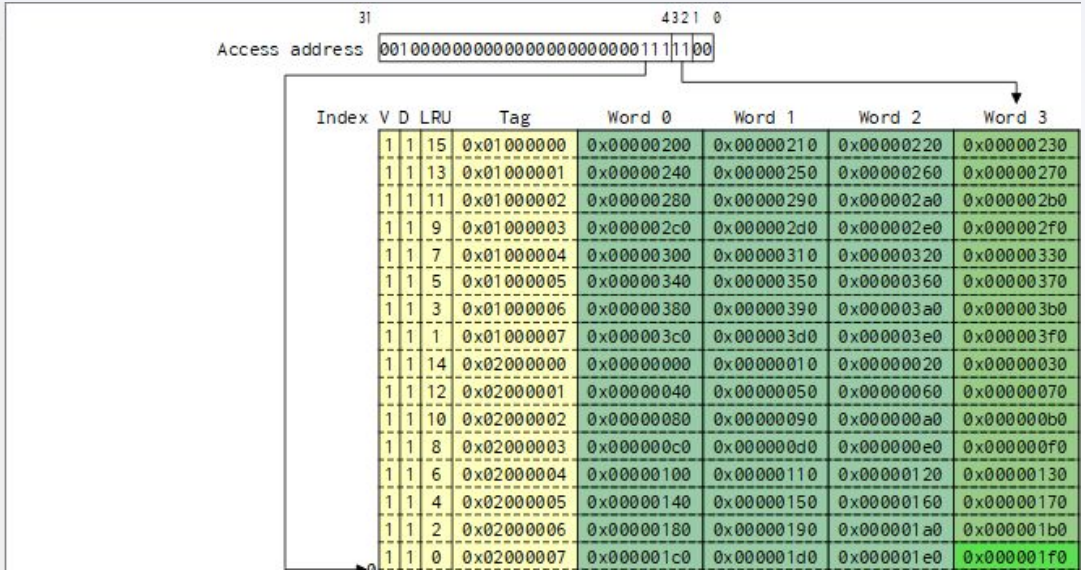
¿Cuál es el mayor inconveniente a partir de ahora?

Veamos que sucede al intercambiar los valores de los vectores

- ¿Cuál cree Ud que será el método de asignación que mejor se comporte en este caso?

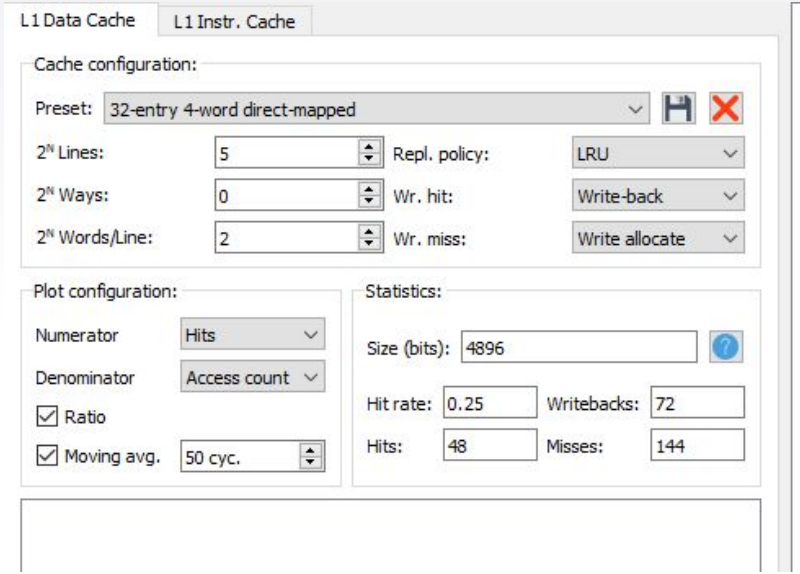
Asignación Asociativa

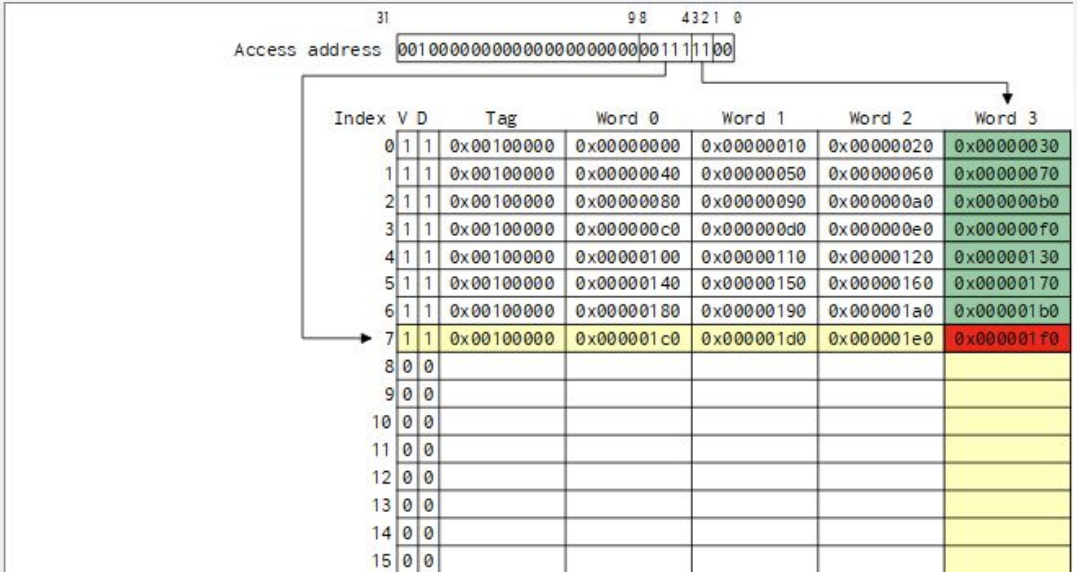




Podemos observar una tasa de aciertos del 91,67% con 176 aciertos y 16 fallos

Asignación Directa





Podemos observar una tasa de aciertos del 25,00% con 48 aciertos y 144 fallos

Address - Dirección																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1				**0**				**0**				**0**				**0**				**0**				**0**				**0**			

Asignacion Asociativa																													
Etiqueta - TAG																												Offset	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1				0				0				0				0				0				0				0			

Asignación Directa																													
Etiqueta - TAG																					Línea					Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0				8				0				0				0				0				0				0			

Asignación Asociativa por conjunto de 2 Vias																													
Etiqueta - TAG																								Conjunto				Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1				0				0				0				0				0				0				0			

Address - Dirección																													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
2				**0**				**0**				**0**				**0**				**0**				**0**				**0**			

Asignacion Asociativa																													
Etiqueta - TAG																												Offset	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2				0				0				0				0				0				0				0			

Asignación Directa																													
Etiqueta - TAG																					Línea					Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1				0				0				0				0				0				0				0			

Asignación Asociativa por conjunto de 2 Vias																													
Etiqueta - TAG																								Conjunto				Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2				0				0				0				0				0				0				0			

Podemos ver que en asignación Directa, ambas direcciones corresponden a la misma Línea dentro de la memoria Cache

Esto produce un fallo en cada reemplazo de los valores de los vectores

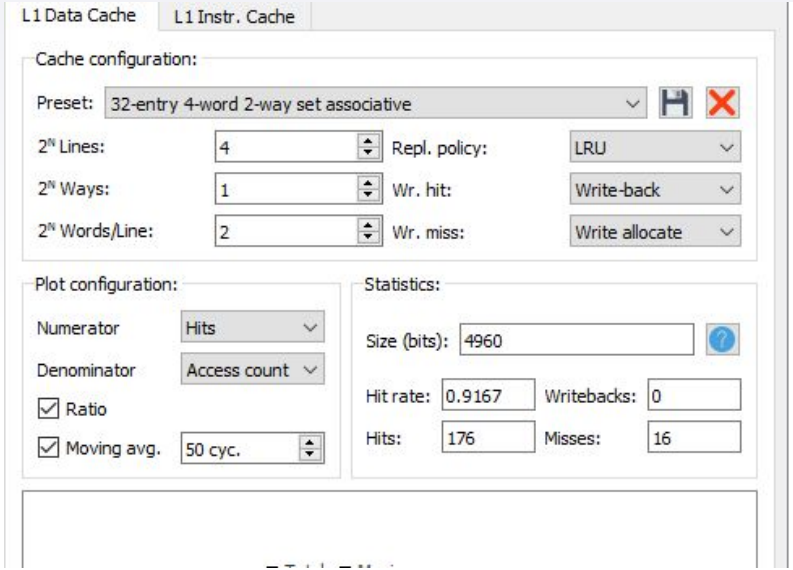
Con lo que acabamos de ver:

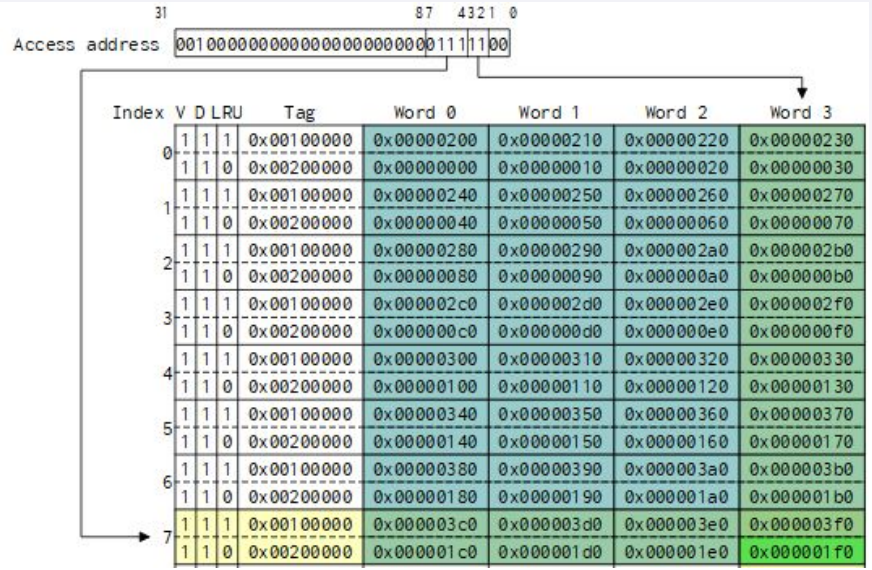
- **¿Podemos decir que un método es mejor que el otro?**
- **¿depende solamente del tamaño de la memoria cache?**
- **¿Cuál piensa usted que sería la solución si no podemos cambiar el programa?**

Con lo que acabamos de ver:

- ¿Podemos decir que un método es mejor que el otro?
- ¿depende solamente del tamaño de la memoria cache?
- ¿Cuál piensa usted que sería la solución si no podemos cambiar el programa?
- ¿Podría solucionarse con asignación asociativa por conjunto de 2 vías?

Asignación Asociativa por conjuntos 2 vías





Podemos observar una tasa de aciertos del 91,67% con 176 aciertos y 16 fallos
Vemos que para esta asignación también ambos vectores permanecen en memoria cache aunque ahora se ven intercalados en su posición

¿Qué pasa con la Cache de Instrucciones?

- **¿Cree Ud que tendremos el mismo inconveniente?**
- **¿Es importante la manera de organizar nuestro algoritmo?**

Veamos este fragmento de programa

```
21 addi x10,x0,32
22 intercambio:        #PC 0x00000048
23     lw x11,0,x7
24     lw x12,0,x8
25     sw x12,0,x7
26     jal salto #PC 0x00000054 salto a 0x0000026c
27     sw x11,0,x8
28     addi x7,x7,4
29     addi x8,x8,4
30     addi x10,x10,-1
31 vuelve:
32     bne x0,x10,intercambio #PC 0x00000068
33 addi x7,x7,-128
34 addi x8,x8,-128
```

```
137
138 salto:
139     sw x11,0,x8
140     addi x7,x7,4
141     addi x8,x8,4
142     addi x10,x10,-1
143     jal vuelve              #PC 0000027C  salta a 00000068
144     bne x0,x10,intercambio
145
146
```

```
54:        218000ef        jal x1 536 <salto>
```

```
27c:       dedff0ef        jal x1 -532 <vuelve>
```

Analicemos que sucede con la memoria cache de instrucciones en el momento que se produce **jal salto** y **jal vuelve**

Asignación Asociativa

jal salto

Asignacion Asociativa

Etiqueta - TAG																								Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
0				0				0				0				0				0				5				4			

Etiqueta salto

Etiqueta - TAG																								Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0
0				0				0				0				0				2				6				C			

jal vuelve

Etiqueta - TAG																								Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
0				0				0				0				0				0				6				8			

Podemos ver que dependiendo del tamaño de la memoria cache no tendríamos conflicto con este salto, en este ejemplo usamos la cache de 32 líneas 4 palabras por línea

Asignación Directa

jal salto

Asignación Directa																													
Etiqueta - TAG																					Línea					Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
0			8				0				0				0				0				0	5			4				

Etiqueta salto

Etiqueta - TAG																					Línea					Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0
0			0				0				0				0				1				0	6			C				

jal vuelve

Etiqueta - TAG																					Línea					Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
0			0				0				0				0				0				0	6			8				

Aquí podemos ver el conflicto que aparece dado que tanto la etiqueta salto con la instrucción **jal vuelve** pertenecen a la misma línea de cache al ser esto un salto repetitivo generará tantos fallos como iteraciones haya, en este ejemplo usamos la cache de 32 líneas 4 palabras por línea

Asignación Asociativa por conjuntos 2 vías

jal salto

Etiqueta salto

jal vuelve

Asignación Asociativa por conjunto de 2 Vias

Etiqueta - TAG																						Conjunto				Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0
0				0				0				0				0				0				5				4			

Etiqueta - TAG																						Conjunto				Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0
0				0				0				0				0				2				6				C			

Etiqueta - TAG																						Conjunto				Offset			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
0				0				0				0				0				0				6				8			

Podemos ver que a pesar de pertenecer al mismo conjunto al ser de dos vias no presentará inconvenientes como en asignación directa, en este ejemplo usamos la cache de 32 líneas 4 palabras por línea asociativa por conjunto en 2 vías

Hasta aquí hemos visto el comportamiento de la cache en Ripes con las configuraciones pre-establecidas en el simulador de la arquitectura Risc-V

Los invitamos a que realicen diferentes pruebas modificando las configuraciones de la memoria cache y analicen cómo se comporta ante diferentes situaciones

Esperamos que con lo que han aprendido se comprenda porque la memoria cache colabora a mejorar el funcionamiento de todo el sistema