



**FACULTAD  
DE INGENIERIA**  
Universidad de Buenos Aires

## [66.20] ORGANIZACIÓN DE COMPUTADORAS

1º CUATRIMESTRE 2020

CURSO 1 - MARTES

---

## TP2 - Introducción a Caches

---

### AUTORES:

Carbón Posse, Ana Sofía - #101 187  
<scarbon@fi.uba.ar>

Fandos, Nicolás Gabriel - #101 018  
<nfandos@fi.uba.ar>

Torraca, Facundo - #101 046  
<ftorraca@fi.uba.ar>

### DOCENTES

Santi, Leandro

Natale, Luciano César

Perez Masci, Hernan

# Índice

<b>1. Enunciado</b>	<b>3</b>
1.1. Objetivos . . . . .	3
1.2. Alcance . . . . .	3
1.3. Requisitos . . . . .	3
1.4. Descripción . . . . .	3
1.4.1. Instalación de cachegrind . . . . .	4
1.4.2. Funcionamiento de cachegrind . . . . .	4
1.5. Desarrollo . . . . .	6
1.6. Informe . . . . .	6
1.7. Entrega de TPs . . . . .	6
<b>2. Introducción</b>	<b>7</b>
<b>3. Código fuente utilizado</b>	<b>8</b>
3.1. Benchmark 0 . . . . .	8
3.2. Benchmark 1 . . . . .	10
3.3. Benchmark 2 . . . . .	11
3.4. Benchmark 3 . . . . .	12
3.5. MakeFile . . . . .	14
<b>4. Análisis de Benchmarks</b>	<b>15</b>
4.1. ¿Que es una Memoria Cache? . . . . .	15
4.2. Cache 1 - Direct Map . . . . .	16
4.2.1. Benchmark 0 . . . . .	17
4.2.2. Benchmark 1 . . . . .	21
4.2.3. Benchmark 2 . . . . .	25
4.2.4. Benchmark 3 . . . . .	28
4.3. Cache 2 - 2 Way Set Associative . . . . .	35
4.3.1. Benchmark 0 . . . . .	36
4.3.2. Benchmark 1 . . . . .	39
4.3.3. Benchmark 2 . . . . .	42
4.3.4. Benchmark 3 . . . . .	45
4.4. Cache 3 - 4 Way Set Associative . . . . .	49
4.4.1. Benchmark 0 . . . . .	50
4.4.2. Benchmark 1 . . . . .	53
4.4.3. Benchmark 2 . . . . .	57
4.4.4. Benchmark 3 . . . . .	61
<b>5. Compilación y Ejecución</b>	<b>67</b>
5.1. Cache 1 . . . . .	68
5.1.1. Benchmark 0 . . . . .	68
5.1.2. Benchmark 1 . . . . .	68
5.1.3. Benchmark 2 . . . . .	68

5.1.4. Benchmark 3 . . . . .	68
5.2. Cache 2 . . . . .	69
5.2.1. Benchmark 0 . . . . .	69
5.2.2. Benchmark 1 . . . . .	69
5.2.3. Benchmark 2 . . . . .	69
5.2.4. Benchmark 3 . . . . .	69
5.3. Cache 3 . . . . .	70
5.3.1. Benchmark 1 . . . . .	70
5.3.2. Benchmark 1 . . . . .	70
5.3.3. Benchmark 2 . . . . .	70
5.3.4. Benchmark 3 . . . . .	70
<b>6. Cálculos de Misses y Miss rate</b>	<b>71</b>
6.1. Cache 1 . . . . .	71
6.1.1. Benchmark 0 . . . . .	71
6.1.2. Benchmark 1 . . . . .	71
6.1.3. Benchmark 2 . . . . .	73
6.1.4. Benchmark 3 . . . . .	73
6.2. Cache 2 . . . . .	75
6.2.1. Benchmark 0 . . . . .	75
6.2.2. Benchmark 1 . . . . .	75
6.2.3. Benchmark 2 . . . . .	76
6.2.4. Benchmark 3 . . . . .	76
6.3. Cache 3 . . . . .	78
6.3.1. Benchmark 0 . . . . .	78
6.3.2. Benchmark 1 . . . . .	78
6.3.3. Benchmark 2 . . . . .	79
6.3.4. Benchmark 3 . . . . .	80
<b>7. Conclusión</b>	<b>81</b>

# 1. Enunciado

## 1.1. Objetivos

Estudiar el comportamiento de diversos sistemas de memoria cache utilizando una serie de escenarios de análisis o benchmarks descriptos a continuación.

## 1.2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

## 1.3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la sección 6, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

## 1.4. Descripción

En este trabajo estudiaremos el comportamiento de una serie de configuraciones de sistemas de memoria cache, analizando la ejecución de programas de prueba o benchmarks en forma similar a lo estudiado en la práctica del día Martes 19/5 [1].

Nombre	Tipo	Comentario	Parámetros cachegrind
C1	DM	asociatividad trivial	--I1=64,1,16 --D1=64,1,16 --LL=64,1,16
C2	2WSA	2-way set-associative	--I1=64,2,16 --D1=64,2,16 --LL=64,2,16
C3	4WSA	4-way set-associative	--I1=64,4,16 --D1=64,4,16 --LL=64,4,16

Cuadro 1: configuraciones para el sistema de memoria.

A lo largo de este TP, adoptaremos las 3 configuraciones para el nivel 1 del sistema de memoria cache indicadas en el cuadro 1. En todos los casos la capacidad total será de 64 bytes, y el tamaño de líneas 16 bytes. Para cada una de estas configuraciones, deberá estudiarse el comportamiento de las mismas al ejecutar una serie de 4 programas MIPS32 disponibles en [2]:

- **benchmark-b0**
- **benchmark-b1**
- **benchmark-b2**
- **benchmark-b3**

Para ello, deberá usarse el entorno QEMU del trabajo anterior [3], y la el programa cachegrind [4], una herramienta de profiling y simulación de sistemas de memoria cache multinivel que forma parte de la suite de software Valgrind [5].

### 1.4.1. Instalación de cachegrind

Debido a fallas en la versión de cachegrind suministrada dentro de la distribución de Linux que usamos en el TP, en este trabajo será necesario instalar una versión más reciente de esta herramienta en forma manual. Para ello basta ejecutar el comando en la consola MIPS32:

```
1 $ gzip -dc valgrind-mips32-debian-stretch.tar.gz | (cd /opt/; tar -xvf -)
```

### 1.4.2. Funcionamiento de cachegrind

Para validar que la herramienta está funcionando, podemos compilar y ejecutar el ejemplo suministrado en

```
1 /opt/valgrind/share/fiuba/01-holamundo.S:
2
3 $ cc -Wall -g -o /tmp/01-holamundo /opt/valgrind/share/fiuba/01-holamundo.S
4
5 $ /opt/valgrind/bin/valgrind --tool=cachegrind /tmp/01-holamundo
6
7 Hola mundo.
```

(Notar que en el ejemplo de arriba sólo hemos mostrado la salida propia del programa, y hemos suprimido las líneas generadas por el propio cachegrind). Este último comando ejecuta el binario 01 - holamundo dentro de la herramienta de profiling del sistema de memoria, y toma nota de la actividad realizada por el cache en el archivo *cachegrind.out.pid*, donde *pid* representa el número de proceso UNIX que tenía el proceso en el momento de realizar la simulación (en este caso *\$pid* vale 3470).

Para acceder a la información de profiling del sistema de memoria, basta con ejecutar el programa *cg\_annotate*, indicando la ubicación del archivo con el código fuente del programa, y de esa manera poder acceder a las anotaciones línea por línea de la actividad del sistema de cache en nuestros programas MIPS32:

```
1
2 $ /opt/valgrind/bin/cg_annotate cachegrind.out.3470 \
3 /opt/valgrind/share/fiuba/01-holamundo.S
```

```
1  
2  
3  
4 2  
5  
6 --- User-annotated source: /opt/valgrind/share/fiuba/01-holamundo.S  
7  
8 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw  
9 --- line 4 ---  
10 . . . . . . .  
11 . . . . . . . . text  
12 . . . . . . . . align 2  
13 . . . . . . .  
14 . . . . . . . . globl main  
15 . . . . . . . . ent main  
16 . . . . . . . . main:  
17 . . . . . . . . set noreorder  
18 3 1 1 0 0 0 0 0 0 .cupload t9  
19 . . . . . . . . set nomacro  
20 1 0 0 0 0 0 0 0 0 addiu sp, sp, -24  
21 1 1 1 0 0 0 1 0 0 sw fp, 20(sp)  
22 1 0 0 0 0 0 0 0 0 move fp, sp  
23 1 0 0 0 0 0 1 0 0 .cprestore 0  
24 . . . . . . .  
25 1 0 0 0 0 0 0 0 0 1i a0, 1  
26 2 0 0 1 0 0 0 0 0 1a a1, msg  
27 1 0 0 0 0 0 0 0 0 1i a2, 12  
28 1 0 0 0 0 0 0 0 0 1i v0, SYS_write  
29 1 1 1 0 0 0 0 0 0 syscall  
30 1 0 0 0 0 0 0 0 0 nop  
31 . . . . . . .  
32 1 0 0 0 0 0 0 0 0 move sp, fp  
33 1 0 0 1 0 0 0 0 0 lw fp, 20(sp)  
34 1 0 0 0 0 0 0 0 0 addiu sp, sp, 24  
35 . . . . . . .  
36 1 0 0 0 0 0 0 0 0 move v0, zero  
37 1 0 0 0 0 0 0 0 0 jr ra  
38 . . . . . . . end main  
39 . . . . . . .  
40 . . . . . . . rdata  
41 . . . . . . . msg:  
42 . . . . . . . . asciiiz "Hola mundo.\n"  
43  
44 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw  
45  
46 19 3 3 2 0 0 2 0 0 events annotated
```

## 1.5. Desarrollo

Para cada combinación de los benchmarks y configuraciones del sistema de memoria presentados en la sección 4 (12 combinaciones o casos de análisis en total), deberá realizarse un análisis detallado del profiling realizado con Valgrind y cg\_annotate, analizando la salida línea por línea, como vimos en la sección 4.2.

En el informe del trabajo práctico, para caso de análisis deberá incluirse:

- Detalle de todos los comandos usados para recolectar los datos de cada una de las corridas: invocación de valgrind, cg annotate, etc.
- Incorporar las anotaciones línea por línea de cg annotate correspondiente al archivo .S del benchmark analizado.
- Para el cada línea del benchmark, explicar en detalle porqué se produce la cantidad de desaciertos indicada para L1D y L1I.
- Calcular la cantidad total de accesos a memoria, aciertos y desaciertos realizada por los caches L1.
- Calcular los miss-rates para L1D y L1I asociado a esa combinación de benchmark y configuración del sistema de memoria.

En todos los casos, La ejecución deberá realizarse en el entorno MIPS32 simulado por QEMU. Asimismo, como en el TP anterior deberá usarse el modo 1 del sistema operativo para manejo de acceso no alineado a memoria [7].

## 1.6. Informe

El informe deberá incluir:

- Análisis detallado de cada uno de los benchmarks, siguiendo los lineamientos descriptos en las sección 5, respetando la estructura de sub-secciones y títulos.
- El código fuente de todos los programas analizados en el TP.
- Instrucciones de compilación y ejecución de cada caso.
- Este enunciado.

## 1.7. Entrega de TPs

La entrega de este trabajo deberá realizarse usando el campus virtual de la materia. Asimismo, en todos los casos, estas presentaciones deberán ser realizadas durante los días martes. El feedback estará disponible de un martes hacia el otro, como ocurre durante la modalidad presencial de cursada. Por otro lado, la ultima fecha de entrega y presentación para esta trabajo sera el martes 16/6.

## 2. Introducción

El objetivo de este segundo trabajo práctico es familiarizarse con el comportamiento de diversos sistemas de memoria cache utilizando una serie de benchmarks.

A medida que se exploren cada una de estas memorias caches con los distintos benchmarks, queremos ver los distintos comportamientos de ellas que puedan resultar interesante. Se propone inicialmente:

- Para cada línea del benchmark, explicar en detalle porque se produce la cantidad de desaciertos indicada para L1D y L1I.
- Calcular la cantidad total de accesos a memoria, aciertos y desaciertos realizada por los caches L1.
- Calcular los miss-rates para L1D y L1I asociado a esa combinación de benchmark y configuración del sistema de memoria.

### 3. Código fuente utilizado

En esta sección publicamos el código fuente de los benchmarks utilizados para poder realizar el análisis. Estos fueron brindados por la cátedra.

Un **benchmark** es una técnica utilizada para medir el rendimiento de un sistema o uno de sus componentes. Una prueba de rendimiento es el resultado de la ejecución de un programa con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con programas similares. En este trabajo vamos a aplicar benchmarks a memorias cache con distintas estructuras para ver como es el rendimiento para cada una de ellas.

#### 3.1. Benchmark 0

```
1 #include <regdef.h>
2 #include <sys/asm.h>
3 #include <sys/syscall.h>
4
5
6 .text
7 .align 2
8
9 .globl main
10 .ent main
11 main:
12 .set noreorder
13 .cpload t9
14 .set nomacro
15 addiu sp, sp, -24
16 sw fp, 20(sp)
17 move fp, sp
18 .cprestore 0
19
20 la t0, aligned
21 .align 20
22 li t1, 100
23 loop: lw t2, 1024(t0)
24 subu t1, t1, 1
25 bnez t1, loop
26
27 # Destruimos el stack frame antes de retornar de main().
28 #
29 move sp, fp
30 lw fp, 20(sp)
31 addiu sp, sp, 24
32
33 # Para volver al sistema operativo cargamos un código de retorno nulo.
34 #
35 move v0, zero
36 jr ra
37 .end main
38
39 .rdata
```

```
40 . align 20
41 aligned:
42 . skip 8192
```

### 3.2. Benchmark 1

```
1 #include <regdef.h>
2 #include <sys/asm.h>
3 #include <sys/syscall.h>
4
5 .text
6 .align 2
7
8 .globl main
9 .ent main
10
11 main:
12     .set noreorder
13     .cupload t9
14     .set nomacro
15     addiu sp, sp, -24
16     sw fp, 20(sp)
17     move fp, sp
18     .cprestore 0
19
20     la t0, aligned
21     li t1, 100
22     .align 20
23 loop: lw t2, 0(t0)
24     addu t0, t0, 4
25     subu t1, t1, 1
26     bnez t1, loop
27
28 # Destruimos el stack frame antes de retornar de main().
29 #
30     move sp, fp
31     lw fp, 20(sp)
32     addiu sp, sp, 24
33
34 # Para volver al sistema operativo cargamos un código de retorno nulo.
35 #
36     move v0, zero
37     jr ra
38     .end main
39
40 .rdata
41 .align 20
42 aligned:
43     .skip 8192
```

### 3.3. Benchmark 2

```
1 #include <regdef.h>
2 #include <sys/asm.h>
3 #include <sys/syscall.h>
4
5 .text
6 .align 2
7
8 .globl main
9 .ent main
10
11 main:
12     .set noreorder
13     .cupload t9
14     .set nomacro
15     addiu sp, sp, -24
16     sw fp, 20(sp)
17     move fp, sp
18     .cprestore 0
19
20     la t0, aligned
21     li t1, 100
22     .align 20
23 loop: lw t2, 0(t0)
24     addu t0, t0, 16
25     subu t1, t1, 1
26     bnez t1, loop
27
28 # Destruimos el stack frame antes de retornar de main().
29 #
30     move sp, fp
31     lw fp, 20(sp)
32     addiu sp, sp, 24
33
34 # Para volver al sistema operativo cargamos un código de retorno nulo.
35 #
36     move v0, zero
37     jr ra
38     .end main
39
40 .rdata
41 .align 20
42 aligned:
43     .skip 8192
```

### 3.4. Benchmark 3

```
1
2 #include <regdef.h>
3 #include <sys/asm.h>
4 #include <sys/syscall.h>
5
6 .text
7 .align 2
8
9 .globl main
10 .ent main
11 main:
12 .set noreorder
13 .cupload t9
14 .set nomacro
15 addiu sp, sp, -24
16 sw fp, 20(sp)
17 move fp, sp
18 .cprestore 0
19
20 la t0, aligned0
21 la t1, aligned1
22 la t2, aligned2
23 li t3, 256
24 loop: lw t4, 0(t0)
25 lw t5, 0(t1)
26 addu t6, t5, t4
27 sw t6, 0(t2)
28 addu t0, t0, 4
29 addu t1, t1, 4
30 addu t2, t2, 4
31 subu t3, t3, 1
32 bnez t3, loop
33
34 # Destruimos el stack frame antes de retornar de main().
35 #
36 move sp, fp
37 lw fp, 20(sp)
38 addiu sp, sp, 24
39
40 # Para volver al sistema operativo cargamos un código de retorno nulo.
41 #
42 move v0, zero
43 jr ra
44 .end main
45
46 .data
47 .align 20
48 aligned0:
49 .skip 8192
50 aligned1:
51 .skip 8192
52 aligned2:
```

53 . skip 8192

### 3.5. MakeFile

```
1  CFLAGS=-Wall -g
2
3  a11: benchmark-b0 benchmark-b1 benchmark-b2 benchmark-b3
4      :
5
6
7  benchmark-b0: benchmark-b0.S
8      $(CC) $(CFLAGS) -o $@ $<
9
10 benchmark-b1: benchmark-b1.S
11     $(CC) $(CFLAGS) -o $@ $<
12
13 benchmark-b2: benchmark-b2.S
14     $(CC) $(CFLAGS) -o $@ $<
15
16 benchmark-b3: benchmark-b3.S
17     $(CC) $(CFLAGS) -o $@ $<
18
19 clean:
20     rm -f cachegrind.out.*
21     rm -f benchmark-b3
22     rm -f benchmark-b2
23     rm -f benchmark-b1
24     rm -f benchmark-b0
```

## 4. Análisis de Benchmarks

### 4.1. ¿Qué es una Memoria Cache?

Una memoria cache es una memoria chica y rápida que captura la localidad temporal y espacial de los programas y se ubica entre la CPU y la memoria principal.

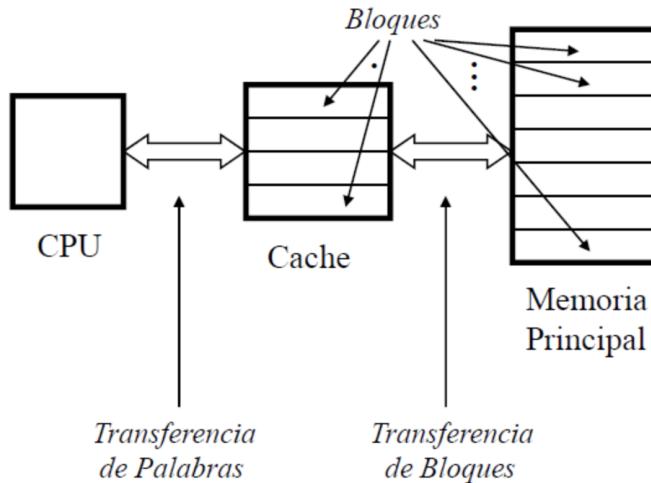


Figura 4.1: Funcionamiento básico de la memoria cache

En la Figura 4.1 podemos observar su funcionamiento de manera simple: La interfase entre la CPU y la memoria caché transfiere lo que pide la ejecución del programa (instrucciones y datos). La memoria caché también tiene interfaz con la memoria principal. Esta transferencia se hace por bloques de memoria: esto conviene porque hay un componente de localidad espacial. Al organizar por bloques, se trae el bloque completo que contiene una palabra, y entonces se anticipa a los pedidos que pueda hacer después el programa (una vez que se accede a una palabra, cosa que lleva mucho tiempo, las palabras vecinas se acceden rápidamente porque las traigo con la palabra que busqué originalmente).

Se denomina **HIT** cuando el CPU encuentra el dato requerido en el caché. En caso de no encontrarlo, se denomina **MISS** (esto significa que habrá que ir a buscarlo a la memoria principal lo cual lleva una penalidad).

Con esto se pueden calcular la tasa de hit y la tasa de miss de una caché como:

$$\text{tasa de hit} = \frac{\text{cantidad de veces que se encuentra lo que se busca en la caché}}{\text{cantidad de veces que se busca en la caché}}$$

Figura 4.2

$$\text{tasa de miss} = \frac{\text{cantidad de veces que no se encuentra lo que se busca en la caché}}{\text{cantidad de veces que se busca en la caché}}$$

Figura 4.3

Las memorias tienen distintas estructuras, las que vamos a analizar en este trabajo son:

- Direct Map
- 2 Way Set Associative
- Full Associative

Para facilitar la lectura y análisis del código vamos a ir separándolo en bloques. De esta forma podremos ir enfocándonos en un grupo de instrucciones a la vez sin que se dificulte seguir el funcionamiento del programa en general. Esto nos sirve para poder explicar de una forma más clara lo que sucede a nivel cache durante la sección importante de los benchmarks que son los loops. Para esto vamos a visualizar distintos diagramas que muestren lo que se va guardando. Además a modo de no hacer muy repetitivo el análisis, si hubiera bloques que se comportan de la misma manera vamos a hacer referencia a bloques anteriores.

## 4.2. Cache 1 - Direct Map

Se dice que una memoria cache tiene estructura de Direct Map si cada bloque tiene un único lugar donde puede aparecer en la caché. El mapeo usualmente se realiza como:

- (*Block address*) MOD (*Number of blocks in cache*)

En particular, la memoria cache que vamos a analizar cuenta con cuatro *bloques* de 16 bytes cada uno y se organiza de la siguiente manera:

- Byte Offset: Addr[1:0]. 2 bytes para direccionar los 4 bytes que posee cada *word*.
- Word Offset: Addr[3:2]. 2 bytes para direccionar los 4 *words* de cada *bloque*.
- Index: Addr[5:4]. 2 bytes de índice para direccionar los cada uno de los 4 *bloques*.
- Tag: Addr[31:6].

IDX	TAG	DATA			
		W0	W1	W2	W3
0					
1		W0	W1	W2	W3
2		W0	W1	W2	W3
3		W0	W1	W2	W3

Figura 4.4: Diseño ilustrativo simplificado de la estructura de la cache

Al ser una cache del tipo *split*, existen dos caches iguales a la de la figura, una para datos y otra para instrucciones. Cada word (*Wi*) indicado, ocupa un tamaño de 4 bytes. Finalmente, a modo de simplificación, asumiremos que la cache se encuentra vacía al comienzo de la ejecución del programa.

#### 4.2.1. Benchmark 0

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	.	.	.	.	.	.	.	.	.	.text
2	.	.	.	.	.	.	.	.	.	.align 2
3	.	.	.	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	.globl main
6	.	.	.	.	.	.	.	.	.	.ent main
7	.	.	.	.	.	.	.	.	.	main:
8	.	.	.	.	.	.	.	.	.	.set noreorder
9	3	1	1	0	0	0	0	0	0	.cupload t9
10	.	.	.	.	.	.	.	.	.	.set nomacro
11	1	0	0	0	0	0	0	0	0	addiu sp, sp, -24

Figura 4.5: Bloque 1

Comenzando por la primer linea vemos que en la columna IR hay un 3, es decir, se ejecutaron 3 operaciones. Esto se debe a que la directiva *.cupload* se expande a tres instrucciones:

- *lui gp,0x42*
- *addiu gp, gp, -24544*
- *addu gp, gp, t9*

Estas instrucciones las pudimos extraer realizando un *objdump -S* al archivo compilado.

Ahora que ya vimos la expansión de *.cupload*, podemos observar que hay un **MISS compulsivo**, que se dará justo en la primer instrucción expandida, recordamos que era *lui gp, 0x42*. Este **MISS** provoca que se traiga a la memoria cache un bloque de memoria completo, que en este caso es de 16 bytes, por lo que las siguientes tres instrucciones van a ser un **HIT**.

En ninguna de las 4 instrucciones hay accesos a datos, sean estos accesos tanto para escritura como para lectura, entonces, no tendremos **MISS** de datos.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	1	1	1	0	0	0	1	1	1	sw fp, 20(sp)
2	1	0	0	0	0	0	0	0	0	move fp, sp
3	1	0	0	0	0	0	1	1	1	.cprestore 0
4	.	.	.	.	.	.	.	.	.	.
5	1	0	0	1	1	1	0	0	0	la t0, aligned

Figura 4.6: Bloque 2

En el bloque 2 podemos observar un comportamiento muy similar al mencionado previamente. La primer instrucción es un **MISS** ya que no estaba incluida en el bloque traído a la cache previamente (este bloque se encuentra 16 bytes por debajo de *lui gp,0x42*).

Este **MISS** genera que se traiga un nuevo bloque completo de memoria (Empezando por  $sw\ fp\ 20(sp)$ ) hasta los siguientes 12 bytes). Entonces, por lo que las siguientes 3 instrucciones serán **HIT**.

Se puede observar que en la primer instrucción ( $sw\ fp\ 20(sp)$ ) hay un **MISS** de escritura, esto sucede por que la instrucción esta escribiendo en una dirección de memoria nunca antes escrita durante la ejecución del programa. Este **MISS**, al igual que pasa con una instrucción también hará que se traigan las siguientes 3 direcciones de memoria además de la dirección  $20 + sp$ .

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	.	.	.	.	.	.	.	.	.	. align 20
2	.	1	1	0	0	0	0	0	0	li t1 , 100
3	100	0	0	100	1	1	0	0	0	loop: lw t2 , 1024(t0)
4	100	0	0	0	0	0	0	0	0	subu t1 , t1 , 1
5	100	0	0	0	0	0	0	0	0	bnez t1 , loop
6	100	0	0	0	0	0	0	0	0	

Figura 4.7: Bloque 3

El bloque 3 es el mas importante del benchmark, ya que es el único bloque de instrucciones que se ejecuta mas de una vez y por lo tanto se puede apreciar la estructura de la cache. Para poder apreciar la estructura nos vamos a auxiliar de la Figura 4.24 para ir mostrando el estado de la memoria cache.

Comenzamos viendo que el código del *loop* esta precedido por la directiva *.align 20*, lo que nos permite saber que los 20 primeros bits del la instrucción *li t1 100* son 0. La instrucción *li t1 100* produce un **MISS compulsivo**, haciendo que se traiga el bloque completo de 16 bytes que incluye el *loop* en su totalidad. Por el motivo mencionado previamente, todas las instrucciones del *loop* son **HIT**.

IDX	TAG	INSTRUCCIONES			
		W0	W1	W2	W3
0		<i>li t1, 100</i>	<i>lw t2, 1024(t0)</i>	<i>subu t1, t1, 1</i>	<i>bnez t1, loop</i>
1		W0	W1	W2	W3
2		W0	W1	W2	W3
3		W0	W1	W2	W3

Figura 4.8: Instrucciones en cache de instrucciones durante la ejecución del loop.

IDX	TAG	DATA			
		W0	W1	W2	W3
0		gp-31692	gp-31688	gp-31684	gp-31680
1		W0	W1	W2	W3
2		W0	W1	W2	W3
3		W0	W1	W2	W3

Figura 4.9: Datos en cache de datos durante la ejecución del loop.

Si bien en la imagen solo se muestran en la cache las instrucciones y los datos traídos durante la ejecución del loop, no necesariamente significa que estén cargados apenas esa información. Decidimos mostrar apenas esa información para simplificar y resaltar la información relevante.

Respecto a los **MISS** de datos, solo existe uno en la primera iteración. Este **MISS** genera que se traiga las direcciones  $1024 + \text{addr}(\text{aligned})$ ,  $1028 + \text{addr}(\text{aligned})$ ,  $1032 + \text{addr}(\text{aligned})$  y  $1036 + \text{addr}(\text{aligned})$ . En las siguientes iteraciones del *loop*, la dirección ya estará cargada en cache haciendo que sean **HIT**.

1	100	1	1	0	0	0	0	0	move	sp , fp
2	1	0	0	1	0	0	0	0	lw	fp , 20(sp)
3	1	0	0	0	0	0	0	0	addiu	sp , sp , 24
4	.	.	.	.	.	.	.	.		
5										
6	1	0	0	0	0	0	0	0	move	v0 , zero

Figura 4.10: Bloque 4

En el bloque 4, la instrucción *move, sp, fp* se ejecuta 100 veces, ya que el procesador MIPS ejecuta siempre la instrucción debajo de un branch.

Nuevamente se puede apreciar el patrón de **MISS compulsivo**, en que la primera instrucción genera un **MISS**, trae los 16 bytes a cache y ejecuta las siguientes 3 instrucciones que van a ser **HIT**.

Todas las instrucciones son operaciones de ALU, por lo que no existen accesos a memoria para datos, consecuentemente, no existen **MISS** de datos.

1	1	1	1	0	0	0	0	0	jr	ra
2	.	.	.	.	.	.	.	.	.	.end main
3	.	.	.	.	.	.	.	.	.	
4	.	.	.	.	.	.	.	.	.	.rdata
5	.	.	.	.	.	.	.	.	.	.align 20
6	.	.	.	.	.	.	.	.	.	aligned:
7	.	.	.	.	.	.	.	.	.	.skip 8192
8										

Figura 4.11: Bloque 5

En la ultima instrucción del benchmark tenemos un **MISS compulsivo**, ya que la instrucción *jr ra* no esta dentro del bloque traído a memoria por el **MISS** de la instrucción *move sp, fp*.

#### 4.2.2. Benchmark 1

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2
3 —— line 10 ——————
4 .     .     .     .     .     .     .     .     .
5 .     .     .     .     .     .     .     .     .     .text
6 .     .     .     .     .     .     .     .     .     .align 2
7 .     .     .     .     .     .     .     .     .
8 .     .     .     .     .     .     .     .     .     .globl main
9 .     .     .     .     .     .     .     .     .     .ent main
10    .     .     .     .     .     .     .     .     .     .main:
11    .     .     .     .     .     .     .     .     .     .set noreorder
12    3     1     1     0     0     0     0     0     0     .cupload t9
13    .     .     .     .     .     .     .     .     .     .set nomacro
14    1     0     0     0     0     0     0     0     0     addiu sp, sp, -24

```

Figura 4.12: Bloque 1

Nuevamente comenzando por la primer linea vemos que en la columna IR hay un 3, es decir que al igual que en el benchmark 0, se ejecutaron 3 operaciones. Esto se debe a que la directiva *.cupload* se expande de la misma manera que vimos antes.

Una vez vista la expansión de *.cupload*, observamos que hay un **MISS compulsivo**, que se dará justo en la primer instrucción de las tres que se expande (lui gp, 0x42). Este **MISS** provoca que se traiga a memoria cache un bloque de memoria completo, que en este caso es de 16 bytes, por lo que las siguientes tres instrucciones serán **HIT**.

Luego ni las instrucciones en las que se expande *.cupload t9*, ni tampoco la instrucción siguiente *addiu sp, sp, -24* realizan accesos a memoria de datos, por lo que no habrá **MISS**.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2   1     1     1     0     0     0     1     1     1     sw  fp , 20(sp)
3   1     0     0     0     0     0     0     0     0     move fp , sp
4   1     0     0     0     0     0     1     1     1     .cprestore 0
5   .

```

Figura 4.13: Bloque 2

En este bloque, la instrucción *sw fp, 20(sp)* será un **MISS compulsivo**, el cual va a provocar que se traiga a la cache tanto esta instrucción como las siguientes tres. También hay que notar que la última instrucción del bloque que se guarda en la cache es la primera de las 2 en las que se divide la pseudo instrucción *la t0, aligned* (si vemos el objdump del archivo compilado vemos que se expande en las siguientes instrucciones:

- *lw t0,-32716(gp)*
- *addiu t0,t0,0*

La primera de estas dos va a ser guardada en el bloque de instrucciones. También tenemos que la instrucción *sw fp, 20(sp)* realiza un acceso a memoria de datos para realizar una escritura y debido a que no tiene el dato *sp+20* habrá un **MISS compulsivo**. Luego la instrucción siguiente, *move fp,sp* no realiza accesos a memoria y la directiva *.cprestore 0* (la cual se expande a *sw gp, 0(sp)*), va a realizar un acceso a memoria de datos para escribir sobre la dirección guardada en *sp+0*. Debido a que se encuentra a mas de 16 bytes de distancia del dato anteriormente guardado (*sp+20*) habrá un nuevo **MISS compulsivo**. Finalmente la ultima instrucción de este bloque (la cual es la primera de las dos en las que se expande la pseudo instrucción *la t0, aligned* que se encuentra en la siguiente parte del código) *lw t0,-32716(gp)* realiza un acceso a memoria de datos para leer la dirección ubicada en *gp-32716* lo cual será un nuevo **MISS**.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw		
1	2	1	1	1	1	1	0	0	0	1a	<i>t0 , aligned</i>
2	1	0	0	0	0	0	0	0	0	1i	<i>t1 , 100</i>
3	.	.	.	.	.	.	.	.	.	.	<i>align 20</i>
4											

Figura 4.14: Bloque 3

La primera instrucción de este tercer bloque, la cual es *addiu t0,t0,0*, nos da un **MISS compulsivo**, esta es la segunda instrucción de las dos en las que se divide *la t0, aligned*. No realiza accesos a memoria de datos pero de todas formas por ser un **MISS** de la cache de instrucciones se traerá el bloque que la contiene y también a la siguiente instrucción *li t1, 100*. Debido a la directiva *.align 20* que pone los 20 bits menos significativos de la dirección de la siguiente instrucción en 0, el bloque no incluirá a las primeras dos instrucciones del loop.

Por ultimo, la instrucción restante de este bloque *li t1, 100* tampoco realiza accesos a la memoria de datos ya que se realiza utilizando la ALU.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw		
1	100	1	1	100	25	25	0	0	0	loop:	<i>lw t2 , 0(t0 )</i>
2	100	0	0	0	0	0	0	0	0	addu	<i>t0 , t0 , 4</i>
3	100	0	0	0	0	0	0	0	0	subu	<i>t1 , t1 , 1</i>
4	100	0	0	0	0	0	0	0	0	bnez	<i>t1 , loop</i>
5	.	.	.	.	.	.	.	.	.		
6	.	.	.	.	.	.	.	.	.	# Destruimos el stack frame	
7	.	.	.	.	.	.	.	.	.		
8											

Figura 4.15: Bloque 4

En el código incluido en este bloque vemos que el loop entra en su totalidad. Por lo tanto, solo habrá un **MISS compulsivo** al principio de las 100 iteraciones en la primera de las 4 instrucciones que componen al loop. Lo que va a suceder en las instrucciones de este bloque es que se va a realizar 100 veces una asignación del contenido de la dirección ubicada en *t0+0* y luego a esto se le suma 4. Por lo tanto, debido a que los bloques son de 16 bytes y que la dirección guardada en *t0+0* se le van sumando de a 4 bytes, vamos a tener que en la memoria de datos al intentar leer el contenido de esa dirección se va a traer el bloque que lo contiene

junto a los próximos 12 bytes (3 words). Entonces debido a la estructura de **Direct Mapped** vamos a tener una secuencia de la forma **HIT-HIT-HIT-MISS** la cual se va a repetir 25 veces ( $4*25 = 100$  accesos a memoria) y en cada **MISS** vamos a tener que se escribe el bloque siguiente en la memoria cache de datos. La única instrucción que accede a memoria de datos en este loop es  $lw t2, 0(t0)$ .

Para poder visualizar lo que sucede dentro del loop, mostramos un esquema de la memoria cache de instrucciones y de la memoria cache de datos. Esta va a estar llena hasta las primeras 4 iteraciones a partir de la dirección inicial de *aligned* representada como  $gp - 32716$ ):

IDX	TAG	INSTRUCCION			
0		W0	W1	W2	W3
		$lw t2, 0(t0)$	$addu t0, t0, 4$	$subu t1, t1, 1$	$bnez t1, loop$
1		W0	W1	W2	W3
		$move sp, fp$	$lw fp, 20(sp)$	$addiu sp, sp, 24$	$move v0, zero$
2		W0	W1	W2	W3
3		W0	W1	W2	W3

Figura 4.16: Cache de instrucciones durante la ejecución del loop

IDX	TAG	DATA			
0		W0	W1	W2	W3
		$gp - 32716$	$gp - 32712$	$gp - 32708$	$gp - 32704$
1		W0	W1	W2	W3
		$gp - 32700$	$gp - 32696$	$gp - 32692$	$gp - 32688$
2		W0	W1	W2	W3
		$gp - 32684$	$gp - 32680$	$gp - 32676$	$gp - 32672$
3		W0	W1	W2	W3
		$gp - 32668$	$gp - 32664$	$gp - 32660$	$gp - 32656$

Figura 4.17: Cache de datos durante la ejecución del loop

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 100 1 1 0 0 0 0 0 0 move sp , fp
3 1 0 0 1 1 1 0 0 0 lw fp , 20(sp)
4 1 0 0 0 0 0 0 0 0 addiu sp , sp , 24
5 .
6 .
7 .
8 . . . . . . . . # Para volver al sistema
9 . . . . . . . . operativo cargamos un código de retorno nulo.
10 .
11 .
12 .
13 .
14 .
15 .
16 .
17 .
18 .
19 .
20 .
21 .
22 .
23 .
24 .
25 .
26 .
27 .
28 .
29 .
30 .
31 .
32 .
33 .
34 .
35 .
36 .
37 .
38 .
39 .
40 .
41 .
42 .
43 .
44 .
45 .
46 .
47 .
48 .
49 .
50 .
51 .
52 .
53 .
54 .
55 .
56 .
57 .
58 .
59 .
60 .
61 .
62 .
63 .
64 .
65 .
66 .
67 .
68 .
69 .
70 .
71 .
72 .
73 .
74 .
75 .
76 .
77 .
78 .
79 .
80 .
81 .
82 .
83 .
84 .
85 .
86 .
87 .
88 .
89 .
90 .
91 .
92 .
93 .
94 .
95 .
96 .
97 .
98 .
99 .
100 .
101 .
102 .
103 .
104 .
105 .
106 .
107 .
108 .
109 .
110 .
111 .
112 .
113 .
114 .
115 .
116 .
117 .
118 .
119 .
120 .
121 .
122 .
123 .
124 .
125 .
126 .
127 .
128 .
129 .
130 .
131 .
132 .
133 .
134 .
135 .
136 .
137 .
138 .
139 .
140 .
141 .
142 .
143 .
144 .
145 .
146 .
147 .
148 .
149 .
150 .
151 .
152 .
153 .
154 .
155 .
156 .
157 .
158 .
159 .
160 .
161 .
162 .
163 .
164 .
165 .
166 .
167 .
168 .
169 .
170 .
171 .
172 .
173 .
174 .
175 .
176 .
177 .
178 .
179 .
180 .
181 .
182 .
183 .
184 .
185 .
186 .
187 .
188 .
189 .
190 .
191 .
192 .
193 .
194 .
195 .
196 .
197 .
198 .
199 .
200 .
201 .
202 .
203 .
204 .
205 .
206 .
207 .
208 .
209 .
210 .
211 .
212 .
213 .
214 .
215 .
216 .
217 .
218 .
219 .
220 .
221 .
222 .
223 .
224 .
225 .
226 .
227 .
228 .
229 .
230 .
231 .
232 .
233 .
234 .
235 .
236 .
237 .
238 .
239 .
240 .
241 .
242 .
243 .
244 .
245 .
246 .
247 .
248 .
249 .

```

Figura 4.18: Bloque 5

En este bloque vamos a tener que la primera instrucción será accedida 100 veces y nos genera el primer **MISS compulsivo**). Debido a que ya está en cache solo tendrá un único **MISS**. Los 100 accesos se deben a que se encuentra a continuación de un branch que antes de saltar va a ejecutar la instrucción siguiente y luego saltará a la dirección de la etiqueta indicada, como el loop se ejecuta 100 veces (se ejecutan 100 saltos por ser t1 distinto de 0) entonces la instrucción *move sp, fp* se ejecutará 100 veces.

Luego solo la instrucción *lw fp, 20(sp)* será la única que acceda a memoria de datos para leer lo guardado en *sp+20* (El cual guardamos en un principio en la cache de datos pero fue reemplazado por los datos usados en el loop).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 1 1 0 0 0 0 0 0 jr ra
3 .
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 .
13 .
14 .
15 .
16 .
17 .
18 .
19 .
20 .
21 .
22 .
23 .
24 .
25 .
26 .
27 .
28 .
29 .
30 .
31 .
32 .
33 .
34 .
35 .
36 .
37 .
38 .
39 .
40 .
41 .
42 .
43 .
44 .
45 .
46 .
47 .
48 .
49 .
50 .
51 .
52 .
53 .
54 .
55 .
56 .
57 .
58 .
59 .
60 .
61 .
62 .
63 .
64 .
65 .
66 .
67 .
68 .
69 .
69 .
70 .
71 .
72 .
73 .
74 .
75 .
76 .
77 .
78 .
79 .
79 .
80 .
81 .
82 .
83 .
84 .
85 .
86 .
87 .
88 .
89 .
89 .
90 .
91 .
92 .
93 .
94 .
95 .
96 .
97 .
98 .
99 .
100 .
101 .
102 .
103 .
104 .
105 .
106 .
107 .
108 .
109 .
109 .
110 .
111 .
112 .
113 .
114 .
115 .
116 .
117 .
118 .
119 .
119 .
120 .
121 .
122 .
123 .
124 .
125 .
126 .
127 .
128 .
129 .
129 .
130 .
131 .
132 .
133 .
134 .
135 .
136 .
137 .
138 .
139 .
139 .
140 .
141 .
142 .
143 .
144 .
145 .
146 .
147 .
148 .
149 .
149 .
150 .
151 .
152 .
153 .
154 .
155 .
156 .
157 .
158 .
159 .
159 .
160 .
161 .
162 .
163 .
164 .
165 .
166 .
167 .
168 .
169 .
169 .
170 .
171 .
172 .
173 .
174 .
175 .
176 .
177 .
178 .
178 .
179 .
180 .
181 .
182 .
183 .
184 .
185 .
186 .
187 .
188 .
188 .
189 .
189 .
190 .
191 .
192 .
193 .
194 .
195 .
196 .
197 .
197 .
198 .
199 .
199 .
200 .
201 .
202 .
203 .
204 .
205 .
206 .
207 .
208 .
208 .
209 .
209 .
210 .
211 .
212 .
213 .
214 .
215 .
216 .
217 .
218 .
218 .
219 .
219 .
220 .
221 .
222 .
223 .
224 .
225 .
226 .
227 .
228 .
229 .
229 .
230 .
231 .
232 .
233 .
234 .
235 .
236 .
237 .
238 .
239 .
239 .
240 .
241 .
242 .
243 .
244 .
245 .
246 .
247 .
248 .
249 .

```

Figura 4.19: Bloque 6

Finalmente, al igual que en el benchmark 0, en la ultima instrucción tenemos un **MISS compulsivo**, ya que la instrucción *jr ra* no esta dentro del bloque traído a memoria por el **MISS** de la instrucción *move sp, fp*.

#### 4.2.3. Benchmark 2

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 . . . . . . . . . . . . . . . .
3 . . . . . . . . . . . . . . . . text
4 . . . . . . . . . . . . . . . . align 2
5 . . . . . . . . . . . . . . . .
6 . . . . . . . . . . . . . . . . globl main
7 . . . . . . . . . . . . . . . . ent main
8 . . . . . . . . . . . . . . . . main:
9 . . . . . . . . . . . . . . . . set noreorder
10 3 1 1 0 0 0 0 0 0 . cupload t9
11 . . . . . . . . . . . . . . . . set nomacro
12 1 0 0 0 0 0 0 0 0 addiu sp, sp, -24
13

```

Figura 4.20: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 0.

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 1 1 0 0 0 1 1 1 sw fp, 20(sp)
3 1 0 0 0 0 0 0 0 0 move fp, sp
4 1 0 0 0 0 0 1 1 1 .cprestore 0

```

Figura 4.21: Bloque 2

Este Bloque se comporta de la misma manera que el Bloque 2 del Benchmark 1.

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 2 1 1 1 1 1 0 0 0 la t0, aligned
3 1 0 0 0 0 0 0 0 0 li t1, 100

```

Figura 4.22: Bloque 3

Este Bloque se comporta de la misma manera que el Bloque 3 del Benchmark 1.

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 . . . . . . . . . . . . . . . . align 20
3 100 1 1 100 100 100 0 0 0 loop: lw t2, 0(t0)
4 100 0 0 0 0 0 0 0 0 addu t0, t0, 16
5 100 0 0 0 0 0 0 0 0 subu t1, t1, 1
6 100 0 0 0 0 0 0 0 0 bnez t1, loop

```

Figura 4.23: Bloque 4

El Bloque 4, como ya mencionamos previamente, es el mas importante del benchmark, ya que es el único bloque de instrucciones que se ejecuta mas de una vez y por lo tanto se puede apreciar la estructura de la cache.

Nos vamos a volver a auxiliar de la Figura 4.24 para poder ir mostrando el estado de la memoria cache. Comenzamos viendo que el código del loop esta precedido por la directiva .align 20, lo que permite saber que los 20 primeros bits del la instrucción *lw, t2, 0(t0)* son 0. La instrucción *lw, t2, 0(t0)* produce un **MISS compulsivo**, haciendo que se traiga el bloque completo de 16 bytes que incluye el loop en su totalidad. Por este motivo, todas las instrucciones del loop son **HIT**.

IDX	TAG	INSTRUCCION			
0		W0 <i>lw t2, 1024(t0)</i>	W1 <i>addiu t0, t0 16</i>	W2 <i>subu t1, t1, 1</i>	W3 <i>bnez t1, loop</i>
1		W0	W1	W2	W3
2		W0	W1	W2	W3
3		W0	W1	W2	W3

Figura 4.24: Cache de instrucciones durante la ejecución del loop.

Respecto a los **MISS** de datos, vemos que en todas las iteraciones hay un fallo de cache. Esto de debe a que siempre que hay un **MISS**, se trae el dato y los siguientes 12 bytes, pero en cada iteración se accede a un dato que se encuentra 16 bytes mas adelante, por lo que nunca se encuentra en el bloque traído a cache previamente. Esto lo podemos observar en 4.25

IDX	TAG	DATA			
0		W0 0x00000000	W1 0x00000004	W2 0x00000008	W3 0x0000000C
1		W0 0x00000010	W1 0x00000014	W2 0x00000018	W3 0x0000001C
2		W0 0x00000020	W1 0x00000024	W2 0x00000028	W3 0x0000002C
3		W0 0x00000030	W1 0x00000034	W2 0x00000038	W3 0x0000003C

Figura 4.25: Cache de datos durante la ejecución del loop.

Además, en la Figura 4.25 podemos observar como el dato requerido (Word 0) nunca se encuentra en el bloque traído en la iteración previa.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	100	1	1	0	0	0	0	0	0	move sp , fp
2	1	0	0	1	1	1	0	0	0	lw fp , 20(sp)
3	1	0	0	0	0	0	0	0	0	addiu sp , sp , 24
4	1	0	0	0	0	0	0	0	0	move v0 , zero

Figura 4.26: Bloque 5

Este Bloque se comporta de la misma manera que el Bloque 4 del Benchmark 0.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	1	1	1	0	0	0	0	0	0	jr ra
2	.	.	.	.	.	.	.	.	.	.end main
3	.	.	.	.	.	.	.	.	.	
4	.	.	.	.	.	.	.	.	.	
5	.	.	.	.	.	.	.	.	.	.rdata
6	.	.	.	.	.	.	.	.	.	.align 20
7	.	.	.	.	.	.	.	.	.	aligned:
8	.	.	.	.	.	.	.	.	.	.skip 8192

Figura 4.27: Bloque 6

Finalmente, este Bloque se comporta de la misma manera que el Bloque 5 del Benchmark 0.

#### 4.2.4. Benchmark 3

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2
3 -- line 10 --
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 3   1   1   0   0   0   0   0   0   .text
13   .   .   .   .   .   .   .   .   .   .align 2
14 1   0   0   0   0   0   0   0   0   .globl main
     .ent main
     main:
     .set noreorder
     .cupload t9
     .set nomacro
     addiu sp, sp, -24

```

Figura 4.28: Bloque 1

En el comienzo de este Bloque, vamos a tener que nuevamente la primera instrucción que se ejecuta es debido a la expansión de la directiva *.cupload t9* la cual se expande en 3 instrucciones:

- lui gp,0x21
- addiu gp,sp,-9672
- addu gp,sp,t9

Estas las pudimos obtener realizando el comando *objdump -S benchmark-03* (siendo benchmark-03 el ejecutable obtenido al realizar el makefile).

Luego estas 3 instrucciones y la siguiente *addiu sp, sp, -24* serán guardadas en la cache de instrucciones. Serán guardadas al realizarse el primer **MISS compulsivo** en la primera instrucción de la directiva expandida *cupload* por lo tanto las demás serán **HITS**. Luego ninguna de las cuatro instrucciones ejecutadas hasta el momento realizan acceso a la memoria de datos, ni para escribir ni para realizar lecturas por lo que no habrá **MISSES**.

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1   1   1   0   0   0   1   1   1   sw fp , 20(sp )
3 1   0   0   0   0   0   0   0   0   move fp , sp
4 1   0   0   0   0   0   1   1   1   .cprestore 0
5 .

```

Figura 4.29: Bloque 2

En este bloque la primer instrucción en ejecutarse es *sw fp, 20(sp)* la cual será un **MISS compulsivo**. Debido a esto se guarda en la memoria cache de instrucciones el bloque que la

incluye y a las siguientes 3 instrucciones, de forma que serán **HITS**. Notar que nuevamente la última instrucción de este bloque de cache será la primera de las dos instrucciones en las que se dividirá la pseudoinstrucción *la t0, aligned 0*. Esta pseudo-instrucción se expande en las instrucciones:

- *lw t0,-32728(gp)*
- *addiu t0,t0,0*

(Nuevamente obtenidas mediante *objdump*).

La primera de estas dos será **HIT** de instrucción y **MISS** de lectura del dato *gp-32728* y la segunda será un **MISS compulsivo** de instrucción. En la primera de las instrucciones se realiza un acceso al dato que está en *sp + 20* generando un **MISS compulsivo** de datos. Además en la directiva *.cprestore 0*, que como sabemos se expande en la instrucción *sw gp, 0(sp)* tiene un acceso al dato que está en *sp + 0*. Esto genera un (**MISS compulsivo**) ya que como los bloques de la cache son de 16 bytes es imposible que el dato en *sp + 0* esté en el mismo bloque que el dato de *sp + 20*.

Finalmente tenemos un último acceso (ya mencionado) de lectura al dato que está en *gp-32728*, esto genera un (**MISS compulsivo**) que lo realiza la primera de las dos instrucciones en las que se expande la pseudoinstrucción *la t0, aligned0*.

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	Dlmw	DLmw	
1	2	1	1	1	1	1	0	0	0	la t0, aligned0
2	2	0	0	1	0	0	0	0	0	la t1, aligned1

Figura 4.30: Bloque 3

Al analizar el Bloque que observamos en la Figura 4.30 es un poco más complicado ya que en la siguiente seguidilla de pseudoinstrucciones:

- *la t0, aligned0*
- *la t0, aligned1*
- *la t0, aligned2* (esta pertenece al bloque siguiente)

Cada una de ellas se expanden en 2 instrucciones. Resulta que el **MISS compulsivo** de este bloque ocurre con *addiu t0,t0,0*, que es la segunda instrucción en la que se expande *la t0, aligned0*. Este **MISS** hace que se traiga a la memoria esta instrucción y las tres siguientes.

Por lo tanto, para la pseudoinstrucción *la t0, aligned2*, el bloque guardado en cache solo llegará a tomar la primera de estas dos, es decir, *lw t0,-32728(gp)*). La siguiente instrucción (*addiu t2,t2,16384*) será un **MISS compulsivo** (lo analizaremos en el siguiente bloque de código).

Debido a que las pseudoinstrucciones *la t0, aligned0*, *la t0, aligned1* y *la t0, aligned2* se expanden en las instrucciones:

- *lw t0,-32728(gp)*
- *addiu t0,t0,0*
- *lw t1,-32728(gp)*
- *addiu t1,t1,8192*
- *lw t2,-32728(gp)*
- *addiu t2,t2,16384*

Tenemos que se realiza un acceso a memoria de datos para realizar una lectura en las instrucciones *lw* que acceden al dato *gp - 32728* y como siempre el acceso es al mismo dato tenemos que hay solo un **MISS compulsivo** en la primera lectura y luego los siguientes dos accesos serán **HIT**.

Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
2	1	1	1	0	0	0	0	0	<i>la t2 , aligned2</i>
1	0	0	0	0	0	0	0	0	<i>li t3 , 256</i>
256	0	0	256	256	256	0	0	0	<i>loop: lw t4 , 0(t0)</i>
256	0	0	256	256	256	0	0	0	<i>lw t5 , 0(t1)</i>
256	1	1	0	0	0	0	0	0	<i>addu t6 , t5 , t4</i>
256	0	0	0	0	0	256	256	256	<i>sw t6 , 0(t2)</i>
256	0	0	0	0	0	0	0	0	<i>addu t0 , t0 , 4</i>
256	0	0	0	0	0	0	0	0	<i>addu t1 , t1 , 4</i>

Figura 4.31: Bloque 4

La primer instrucción del bloque 4 de código es la segunda instrucción en la que se expandió la pseudoinstrucción *la, t2, aligned2*. Esta misma genera un **MISS compulsivo** que cargara en cache de instrucciones esta misma y las siguientes tres instrucciones, por ese motivo veremos tres **HITS** consecutivos.

Similarmente, en el segundo conjunto de 4 instrucciones tenemos un **MISS compulsivo** en la instrucción *addu t6, t5, t4* y las siguientes 3 instrucciones serán **HITS**.

La cache de instrucciones al comienzo del loop sera:

IDX	TAG	INSTRUCCIONES			
0		W0	W1	W2	W3
0		<i>addiu t2, t2, 16384</i>	<i>li t3, 256</i>	<i>lw t4, 0(t0)</i>	<i>lw t5, 0(t1)</i>
1		W0	W1	W2	W3
1		<i>addu t6, t5, t4</i>	<i>sw t6, 0(t2)</i>	<i>addu t0, t0, 4</i>	<i>addu t1, t1, 4</i>
2		W0	W1	W2	W3
2		<i>addu t2, t2, 4</i>	<i>subu t3, t3, 1</i>	<i>bnez t3, loop</i>	<i>move sp, fp</i>
3		W0	W1	W2	W3
3					

Figura 4.32: Cache de instrucciones durante la ejecución del loop.

Como antes del inicio del loop no existe ninguna directiva que alinee las instrucciones, utilizamos el programa *objdump* y analizamos en base al offset de cada instrucción respecto al inicio del programa. Esto hace que una vez cargado el benchmark en memoria, el índice de bloque al cual se debe cargar en cache sea distinto al mostrado en la Figura 4.32, pero no cambiara la forma en que están agrupadas las instrucciones.

Previamente a comenzar el análisis sobre el acceso a datos, nos basaremos en los valores obtenidos del registro *gp* al comienzo del loop mediante el uso del debugger *gdb*.

- **gp:** 0x5575e000 (1433788416 en decimal)

Respecto a los accesos a datos, se presenta una situación particular no observada en los benchmarks anteriores. La cache entra en un proceso conocido como **trashing**. Esto sucede cuando la cache sobrescribe los datos previamente almacenados, haciendo que nunca los pueda reutilizar.

En este caso existen tres accesos a memoria de datos, en los cuales dos son para lectura y uno para escritura. Las direcciones accedidas en cada iteración *mapearán* al mismo bloque de cache, dado que poseen los mismos bits de índice. En particular, sucede que hay un **MISS** al cargar una dirección dentro del espacio de *aligned0*, se trae el bloque a cache y luego es sobrescrito en la siguiente instrucción de acceso a datos, al haber un fallo de cache cuando al acceder una dirección del bloque de *aligned1*. Similarmente sucede con la *aligned2*.

Si analizamos lo que ocurre en la primer iteración del loop, vemos que:

1. Hay un **MISS** al tratar de obtener un dato de la dirección guardada en *t0* la cual es *gp* - 32728 (0x55756028) en la instrucción *lw t4, 0(t0)*. Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en el bloque de índice 2 (bits 4 y 5).
2. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en *t1* la cual es *gp* - 24536 (0x55758028) en la instrucción *lw t5, 0(t1)*. Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en el bloque de índice 2 (bits 4 y 5). Este bloque sobrescribe al traído en el paso anterior.

3. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en  $t2$  la cual es  $gp - 16384$  (0x5575A028) en la instrucción  $sw t6, 0(t2)$ . Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en el bloque de índice 2 (bits 4 y 5). Este bloque sobrescribe al traído en el paso anterior.

Como vemos en la secuencia, el siguiente bloque leído sobrescribe al anterior, imposibilitando reutilizarlo cuando se lo vuelva a requerir.

En la Figura 4.33 mostramos como en la cache de datos se van guardando las direcciones mencionadas sobre el mismo índice (Los datos no están simultáneamente en cache. La representación es a modo de esclarecer la superposición sobre el mismo índice). Las direcciones del mismo color corresponden a datos traídos en el mismo bloque.

IDX	TAG	DATA			
0		W0	W1	W2	W3
1		W0	W1	W2	W3
2		W0 <i>gp-32728</i> <i>gp-24536</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-24524</i> <i>gp-16332</i>
3		W0	W1	W2	W3

Figura 4.33: Cache de datos durante la ejecución del loop.

Además tenemos que en cada iteración a las direcciones guardadas en  $t0$ ,  $t1$  y  $t2$  aumentan en 4 por lo que si no hubiese thrashing, de todas formas tendríamos un **MISS compulsivo** cada 4 iteraciones ya que intentaríamos acceder al dato de una dirección 16 bytes mas alta que la originalmente guardada en la cache de datos.

Luego de estas 4 iteraciones se usará el 'bloque siguiente de la cache', es decir que si en las 4 primeras iteraciones los datos mapeaban al bloque 2, entonces las siguientes 4 van a mapear al bloque 3, las siguientes 4 iteraciones al bloque 0, las siguientes 4 iteraciones al bloque 1 y así hasta completar las 256 iteraciones, de forma tal que la cache es completamente ocupada por datos del loop, removiendo datos cargados anteriores a la ejecución de este.

En la Figura 4.34 mostramos como queda la cache llena de datos necesarios para el loop luego de 16 iteraciones:

IDX	TAG	DATA			
		W0	W1	W2	W3
0		<i>gp-32696</i> <i>gp-24504</i> <i>gp-16312</i>	<i>gp-32692</i> <i>gp-24500</i> <i>gp-16308</i>	<i>gp-32688</i> <i>gp-24496</i> <i>gp-16304</i>	<i>gp-32684</i> <i>gp-24492</i> <i>gp-16300</i>
1		<b>W0</b> <i>gp-32680</i> <i>gp-24488</i> <i>gp-16296</i>	<b>W1</b> <i>gp-32676</i> <i>gp-24484</i> <i>gp-16292</i>	<b>W2</b> <i>gp-32672</i> <i>gp-24480</i> <i>gp-16288</i>	<b>W3</b> <i>gp-32668</i> <i>gp-24476</i> <i>gp-16284</i>
2		<b>W0</b> <i>gp-32728</i> <i>gp-24536</i> <i>gp-16344</i>	<b>W1</b> <i>gp-32724</i> <i>gp-24532</i> <i>gp-16340</i>	<b>W2</b> <i>gp-32720</i> <i>gp-24528</i> <i>gp-16336</i>	<b>W3</b> <i>gp-32716</i> <i>gp-24524</i> <i>gp-16332</i>
3		<b>W0</b> <i>gp-32712</i> <i>gp-24520</i> <i>gp-16328</i>	<b>W1</b> <i>gp-32708</i> <i>gp-24516</i> <i>gp-16324</i>	<b>W2</b> <i>gp-32704</i> <i>gp-24512</i> <i>gp-16320</i>	<b>W3</b> <i>gp-32700</i> <i>gp-24508</i> <i>gp-16316</i>

Figura 4.34: Cache de datos completa luego de 16 iteraciones.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr Dw   D1mw  DLmw
2 256    1     1     0     0     0     0     0     0      addu   t2 , t2 , 4
3 256    0     0     0     0     0     0     0     0      subu   t3 , t3 , 1
4 256    0     0     0     0     0     0     0     0      bnez   t3 , loop
5 .
6 .
7 .
8 .
# Destruimos el stack frame
antes de retornar de main().
#
move sp , fp

```

Figura 4.35: Bloque 5

En el último bloque de instrucciones perteneciente al loop tenemos un nuevo **MISS impulsivo** en la instrucción *addu t2, t2, 4* lo que va a resultar en que se guarde en cache el bloque que la contiene a esta y a las siguientes 3 instrucciones las cuales serán **HIT**. También vemos que sucede lo mismo que en los benchmarks anteriores en los que la instrucción *move sp, fp* es ejecutada 256 veces (una vez por iteración del loop) debido a que aunque haya un branch siempre se ejecuta la instrucción siguiente a este antes de realizar el salto a la dirección indicada.

En este bloque de instrucciones no tenemos accesos a memoria de datos para realizar lecturas o escrituras. Luego como la totalidad de las instrucciones del loop puede ser guardado en cache sin ser reemplazado vamos a tener que no hubo **MISSES en régimen** en lo que refiere a instrucciones.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 1     1      1     1     1      1     0     0     0    lw   fp , 20(sp)
3 1     0      0     0     0      0     0     0     0    addiu sp , sp , 24
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 .
13 .
14 .
15 .
16 .
17 — line 56 ——————

```

# Para volver al sistema operativo cargamos un código de retorno nulo.  
#  
move v0, zero  
jr ra  
.end main  
.data  
.align 20  
aligned0:  
.skip 8192  
aligned1:  
.skip 8192

Figura 4.36: Bloque 6

Finalmente el último **MISS compulsivo** en acceso a instrucciones es en la línea de *lw fp, 20(sp)* la cual una vez que se guarde en la cache el bloque que la contiene ya vamos a tener guardadas también a las tres instrucciones siguientes que serían las últimas que conforman el final del programa por lo que serán **HIT**.

Finalmente, el último acceso a memoria de datos es en el *lw fp, 20(sp)* en el que se intenta leer el dato que está en *sp + 20*, el cual será un **MISS en régimen**(ya que el dato había sido guardado previamente en la memoria cache de datos y fue removido durante la ejecución del loop).

### 4.3. Cache 2 - 2 Way Set Associative

Se dice que una memoria cache tiene estructura de Set Associative (2 Ways) si un bloque puede ser ubicado en un conjunto de lugares restringido dentro de la caché, la misma se dice que es asociativa por conjuntos (Siendo un conjunto un grupo de bloques en la caché). Aquí, un bloque primero es mapeado a un conjunto y luego el bloque puede ser ubicado en cualquier parte del conjunto.

En particular, la memoria cache que vamos a analizar cuenta con cuatro *bloques* de 16 bytes cada uno, un total de 64 bytes. Al ser una cache de 2 vías y 4 bloques, tendremos 2 líneas (llamamos linea a n-bloques en paralelo). Esta memoria se organiza de la siguiente manera:

- Byte Offset: Addr[1:0]. 2 bytes para direccionar cada uno de los 4 bytes de la palabra.
- Word Offset: Addr[3:2]. 2 bytes para direccionar cada una de las 4 palabras del bloque.
- Index: Addr[4]. 1 solo byte para direccionar cada una de las 2 líneas.
- Tag: Addr[31:5].

En esta estructura de cache, ante una búsqueda, ambos bloques de la línea de analizan en paralelo.

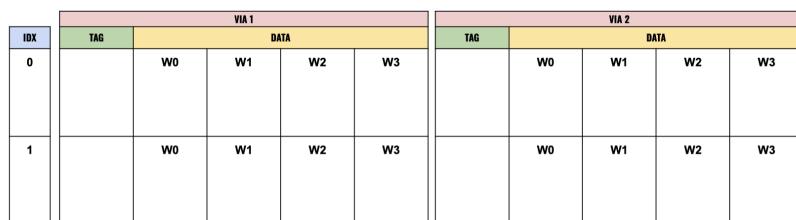


Figura 4.37: Diseño ilustrativo simplificado de la estructura de la cache

Al ser una cache del tipo *split*, existen dos secciones iguales a la de la imagen, una para datos y otra para instrucciones.

Previo a comenzar con los análisis de cada uno de los benchmarks, explicaremos el motivo de porque esta cache se comporta igual a la cache analizada en la sección anterior (Direct Map), en algunos bloques de código. En las secciones del benchmark donde las instrucciones se ejecutan una única vez, se visualiza un comportamiento constante en las caches analizadas que consiste en un **MISS** seguido de tres **HIT**. Esto se debe a que el bloque mínimo de transferencia de la memoria de nivel superior a la cache es un bloque, es decir, 16 bytes. Como en la arquitectura trabajada, **MIPS32**, todas las instrucciones ocupan 4 bytes, entonces, ante un **MISS** se trae un conjunto de 4 instrucciones completas a cache. Finalmente, como cada una de esas instrucciones solo se ejecuta una única vez, no importa en cual bloque de cache guarda o que bloque sobrescriben, ya que no afectara el funcionamiento de la memoria cache a lo largo del benchmark. Por ese motivo, haremos referencia a las secciones previas del informe donde se explica con detalles el motivo de los fallos de cache en esa parte y nos enfocaremos en explicar detalladamente aquellas secciones del benchmark donde si es posible visualizar la estructura interna de la cache y que difieren de las demás.

#### 4.3.1. Benchmark 0

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 . . . . . . . . . .
3 . . . . . . . . . . . . text
4 . . . . . . . . . . . . align 2
5 . . . . . . . . . . . .
6 . . . . . . . . . . . . globl main
7 . . . . . . . . . . . . ent main
8 . . . . . . . . . . . . main:
9 . . . . . . . . . . . . set noreorder
10 3 1 1 0 0 0 0 0 0 . cupload t9
11 . . . . . . . . . . . . set nomacro
12 1 0 0 0 0 0 0 0 0 addiu sp, sp, -24

```

Figura 4.38: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 1 1 0 0 0 1 1 1 sw fp, 20(sp)
3 1 0 0 0 0 0 0 0 0 move fp, sp
4 1 0 0 0 0 0 1 1 1 . cprestore 0
5 . . . . . . . . .
-- -

```

Figura 4.39: Bloque 2

Nuevamente, este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 0 0 1 1 1 0 0 0 la t0, aligned
3 . . . . . . . . . . align 20

```

Figura 4.40: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 1    1      1      0     0      0      0      0      0      1i   t1 , 100
3 100  0      0     100   1      1      0      0      0      loop: lw   t2 , 1024(t0)
4 100  0      0      0     0      0      0      0      0      subu t1 , t1 , 1
5 100  0      0      0     0      0      0      0      0      bnez t1 , loop
6 .
7 .
8 antes de retornar de main() .
9 .

```

Figura 4.41: Bloque 4

La instrucción *li, t1, 100* sera un **MISS compulsivo** y el traer el bloque de instrucciones que la contiene a cache provocara que se guarde todo el loop en su totalidad en un único bloque, por lo que durante el desarrollo del mismo (sus 100 iteraciones) no habrá **MISSES** de lectura de instrucciones. Como todo el loop entra en un único bloque de instrucciones de cache no habrá diferencia en cuanto accesos si utilizamos esta estructura de cache (2WA) o si usamos alguno de los otros 2 (DM o 4WA).

En este loop tenemos también un único **MISS** de lectura de dato en la instrucción *lw, t2, 1024(t0)* ya que no teníamos guardado en la cache el dato que se encuentra en *t0+1024*. El **MISS** será único debido a que siempre se accede al mismo dato para leerlo y nunca se lo modifica, por lo que en cada iteración siempre se lo va a volver a encontrar en la cache, entonces tenemos (**HIT**).

Una representación de como quedarían las caches de instrucciones y datos en esta sección con el loop sería la siguiente:

IDX	VIA 1					VIA 2				
	TAG	INSTRUCCIONES				TAG	INSTRUCCIONES			
0		W0	W1	W2	W3		W0	W1	W2	W3
		<i>li t1, 100</i>	<i>lw t2, 1024(t0)</i>	<i>subu t1, t1, 1</i>	<i>bnez t1, loop</i>					
1		W0	W1	W2	W3		W0	W1	W2	W3

Figura 4.42: Cache de instrucciones durante la ejecución del loop

IDX	VIA 1					VIA 2				
	TAG	DATA				TAG	DATA			
0		W0	W1	W2	W3		W0	W1	W2	W3
		<i>gp-31692</i>	<i>gp-31688</i>	<i>gp-31684</i>	<i>gp-31680</i>					
1		W0	W1	W2	W3		W0	W1	W2	W3

Figura 4.43: Cache de datos durante la ejecución del loop

Mostramos en las Figuras 4.42 y 4.43 lo que sería el contenido inicial de *t0* que era *gp-32716* más el offset de 1024, dando como resultado el bloque de 16 bytes que arranca en *gp-31696*.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 100    1      1     0      0      0     0      0      0      move  sp , fp
3 1      0      0     1      0      0     0      0      0      lw   fp , 20(sp)
4 1      0      0     0      0      0     0      0      0      addiu sp , sp , 24
5 .
6 .
7 .
8 1      0      0     0      0      0     0      0      0      # Para volver al sistema
   operativo cargamos un código de retorno nulo.
   #
   move v0 , zero

```

Figura 4.44: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 1      1      1     0      0      0     0      0      0      jr   ra
3 .
4 .
5 .
6 .
7 .
8 .
9 .

```

Figura 4.45: Bloque 6

En este ultimo bloque tenemos un **MISS compulsivo**, ya que la instrucción *jr ra* no esta dentro del bloque traído a memoria por el **MISS** de la instrucción *move sp, fp*.

### 4.3.2. Benchmark 1

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 2   1   1   1   1   1   0   0   0   la   t0 , aligned
3 1   0   0   0   0   0   0   0   0   li   t1 , 100
4 .   .   .   .   .   .   .   .   .   align 20

```

Figura 4.46: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1   1   1   0   0   0   1   1   1   sw   fp , 20(sp)
3 1   0   0   0   0   0   0   0   0   move  fp , sp
4 1   0   0   0   0   0   1   1   1   .cprestore 0
5 .   .   .   .   .   .   .   .   .
6 .

```

Figura 4.47: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2
3 — line 10 ——————
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 3   1   1   0   0   0   0   0   0   .text
13 .
14 1   0   0   0   0   0   0   0   0   .align 2
15 .

```

Figura 4.48: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 100    1     1 100    25    25  0     0     0     loop: lw t2 , 0(t0)
3 100    0     0 0     0     0  0     0     0     addu t0 , t0 , 4
4 100    0     0 0     0     0  0     0     0     subu t1 , t1 , 1
5 100    0     0 0     0     0  0     0     0     bnez t1 , loop
6 .
7 .
8 antes de retornar de main().      #

```

Figura 4.49: Bloque 4

En el loop principal de este benchmark vamos a tener que al igual que en la cache anterior tenemos primero un **MISS compulsivo** que va a hacer que nos guardemos en cache un bloque de instrucciones que incluye la totalidad del loop (ya que esta compuesto por solo 4 instrucciones). Luego tenemos que solo la instrucción *lw t2, 0(t0)* accede a memoria de datos para realizar una operación de lectura y de forma similar a lo que sucedía en este benchmark en la cache anterior debido a que la operación que se realiza en el loop es el de sumarle 4 al valor guardado en *t0* (una dirección de memoria) por lo que lo único que va a suceder es que cada 4 ciclos (luego de una secuencia de accesos a datos de la forma **HIT-HIT-HIT-MISS**) vamos a tener que el siguiente bloque de datos de 16 bytes que no encontramos se va a guardar en un bloque con índice distinto al bloque guardado en la iteración anterior, de forma que se van intercalando primero los índices y luego las vías.

Un ejemplo de como quedaría la cache de datos y la de instrucciones luego de las primeras 4 iteraciones sería la siguiente:

IDX	VIA 1					VIA 2				
	TAG	INSTRUCCIONES				TAG	INSTRUCCIONES			
0		W0	W1	W2	W3					
		<i>lw t2, 0(t0)</i>	<i>addu t0, t0, 4</i>	<i>subu t1, t1, 1</i>	<i>bnez t1, loop</i>					
1		W0	W1	W2	W3					

Figura 4.50: Cache de instrucciones durante la ejecución del loop

IDX	VIA 1					VIA 2				
	TAG	DATA				TAG	DATA			
0		W0	W1	W2	W3					
		<i>gp-32716</i>	<i>gp-32712</i>	<i>gp-32708</i>	<i>gp-32704</i>					
1		W0	W1	W2	W3					
		<i>gp-32700</i>	<i>gp-32696</i>	<i>gp-32692</i>	<i>gp-32688</i>					

Figura 4.51: Cache de datos durante la ejecución del loop

Luego de esas 4 iteraciones el siguiente bloque se guardaría en la vía 1 del conjunto con índice 0, el siguiente bloque en la vía 1 del conjunto con índice 1, el siguiente en la vía 2 del conjunto con índice 0 y el siguiente en la vía 2 del conjunto con índice 1. Esta secuencia se repite 23 veces mas (Para un total de 25).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2 100    1     1     0     0     0     0     0     0      move sp , fp
3   1     0     0     1     1     1     0     0     0      lw   fp , 20(sp)
4   1     0     0     0     0     0     0     0     0      addiu sp , sp , 24
5   .
6   .
7   .
8   .
9   .

```

operativo cargamos un código de retorno nulo.

#

move v0 , zero

Figura 4.52: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2   1     1     1     0     0     0     0     0     0      jr   ra
3   .
4   .
5   .
6   .
7   .
8   .

```

.end main

.rdata

.align 20

aligned:

skip 8192

Figura 4.53: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 1 en la Cache 1 (Direct Map).

#### 4.3.3. Benchmark 2

```

1 Ir   I1mr  ILmr   Dr  D1mr  DLmr  Dw  D1mw  DLmw
2   .     .      .     .     .     .     .     .     .
3   .     .      .     .     .     .     .     .     .
4   .     .      .     .     .     .     .     .     .
5   .     .      .     .     .     .     .     .     .
6   .     .      .     .     .     .     .     .     .
7   .     .      .     .     .     .     .     .     .
8   .     .      .     .     .     .     .     .     .
9   3     1      1     0     0     0     0     0     .
10  .     .      .     .     .     .     .     .     .
11  1     0      0     0     0     0     0     0     addiu sp , sp , -24

```

Figura 4.54: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 2 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr   Dr  D1mr  DLmr  Dw  D1mw  DLmw
2   1     1      1     0     0     0     1     1     1     sw   fp , 20(sp)
3   1     0      0     0     0     0     0     0     0     move  fp , sp
4   1     0      0     0     0     0     1     1     1     .cprestore 0

```

Figura 4.55: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 2 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr   Dr  D1mr  DLmr  Dw  D1mw  DLmw
2   2     1      1     1     1     1     0     0     0     la   t0 , aligned
3   1     0      0     0     0     0     0     0     0     li   t1 , 100

```

Figura 4.56: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 2 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr   Dr  D1mr  DLmr  Dw  D1mw  DLmw
2   .     .      .     .     .     .     .     .     .
3 100   1      1 100   100   100   0     0     0     .align 20
4 100   0      0 0    0     0     0     0     0     loop: lw   t2 , 0(t0)
5 100   0      0 0    0     0     0     0     0     addu  t0 , t0 , 16
6 100   0      0 0    0     0     0     0     0     subu  t1 , t1 , 1

```

Figura 4.57: Bloque 4

Como ya mencionamos, el .align 20 nos permite saber que los 20 primeros bits de la instrucción (lw t2, 0(t0)) son 0. Esta produce un **MISS compulsivo** haciendo que se traiga el bloque completo de 16 bytes que incluye el loop en su totalidad. Por el motivo mencionado previamente, todas las instrucciones del loop son **HIT**

IDX	VIA 1					VIA 2				
	TAG	INSTRUCCIONES				TAG	INSTRUCCIONES			
0		W0	W1	W2	W3		W0	W1	W2	W3
		lw t2, 0(t0)	addu t0,t0,16	subu t1, t1, 1	bnez t1, loop					
1		W0	W1	W2	W3		W0	W1	W2	W3

Figura 4.58: Cache de instrucciones durante la ejecución del loop

En la Figura 4.58 arriba se como queda la cache cargada con las instrucciones del loop y como entra completamente en 1 bloque, por lo que no hay fallos de cache en régimen.

IDX	VIA 1					VIA 2				
	TAG	DATA				TAG	DATA			
0		W0	W1	W2	W3		W0	W1	W2	W3
		0x00000000	0x00000004	0x00000008	0x0000000C		0x00000020	0x00000024	0x00000028	0x0000002C
1		W0	W1	W2	W3		W0	W1	W2	W3
		0x00000010	0x00000014	0x00000018	0x0000001C		0x00000030	0x00000034	0x00000038	0x0000003C

Figura 4.59: Cache de datos durante la ejecución del loop

En la Figura 4.59 podemos observar como se acomodan los datos en las primeras cuatro iteraciones. Si bien la estructura de la cache es diferente a la Cache 1 (Direct Map), presenta el mismo problema, que el dato requerido nunca esta incluido en el bloque traído previamente. Por ese motivo, siempre hay un fallo de cache de datos.

1	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
2	100	1	1	0	0	0	0	0	0	move sp , fp
3	1	0	0	1	1	1	0	0	0	lw fp , 20(sp)
4	1	0	0	0	0	0	0	0	0	addiu sp , sp , 24

Figura 4.60: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 2 en la Cache 1 (Direct Map).

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	1	0	0	0	0	0	0	0	0	move v0, zero
2	1	1	1	0	0	0	0	0	0	jr ra
3	.	.	.	.	.	.	.	.	.	.end main
4	.	.	.	.	.	.	.	.	.	.rdata
5	.	.	.	.	.	.	.	.	.	.align 20
6	.	.	.	.	.	.	.	.	.	aligned:
7	.	.	.	.	.	.	.	.	.	skip 8192
8	.	.	.	.	.	.	.	.	.	
9	.	.	.	.	.	.	.	.	.	

Figura 4.61: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 2 en la Cache 1 (Direct Map).

#### 4.3.4. Benchmark 3

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2
3 -- Line 10 --
4 .
5 .
6 . . . . . . . . . . . . . . . text
7 . . . . . . . . . . . . . . align 2
8 . . . . . . . . . . . . . . globl main
9 . . . . . . . . . . . . . . ent main
10 . . . . . . . . . . . . . . main:
11 . . . . . . . . . . . . . . set noreorder
12 3 1 1 0 0 0 0 0 0 cpload t9
13 . . . . . . . . . . . . . . set nomacro
14 1 0 0 0 0 0 0 0 0 addiu sp, sp, -24

```

Figura 4.62: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 3 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 1 1 0 0 0 1 1 1 sw fp, 20(sp)
3 1 0 0 0 0 0 0 0 0 move fp, sp
4 1 0 0 0 0 0 1 1 1 cprestore 0
5 .

```

Figura 4.63: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 3 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 2 1 1 1 1 1 0 0 0 1a t0, aligned0
3 2 0 0 1 0 0 0 0 0 1a t1, aligned1

```

Figura 4.64: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 3 en la Cache 1 (Direct Map).

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
1	2	1	1	1	0	0	0	0	0	la t2 , aligned2
2	1	0	0	0	0	0	0	0	0	li t3 , 256
3	256	0	0	256	256	256	0	0	0	loop : lw t4 , 0(t0)
4	256	0	0	256	256	256	0	0	0	lw t5 , 0(t1)
5	256	1	1	0	0	0	0	0	0	addu t6 , t5 , t4
6	256	0	0	0	0	256	256	256	256	sw t6 , 0(t2)
7	256	0	0	0	0	0	0	0	0	addu t0 , t0 , 4
8	256	0	0	0	0	0	0	0	0	addu t1 , t1 , 4
9	256	0	0	0	0	0	0	0	0	

Figura 4.65: Bloque 4

Al igual que como sucedía en la cache anteriormente analizada, tenemos que la primer instrucción del bloque 4 de código, es la segunda instrucción en la que se expandió la pseudoinstrucción *la, t2, aligned2*. Esta misma genera un **MISS compulsivo** que cargara en cache de instrucciones esta misma y las siguientes tres instrucciones, por ese motivo veremos tres **HITS** consecutivos.

Similarmente, en el segundo conjunto de 4 instrucciones tenemos un **MISS compulsivo** en la instrucción *addu t6, t5, t4* y las siguientes 3 instrucciones serán **HITS**.

La cache de instrucciones al comienzo del loop sera:

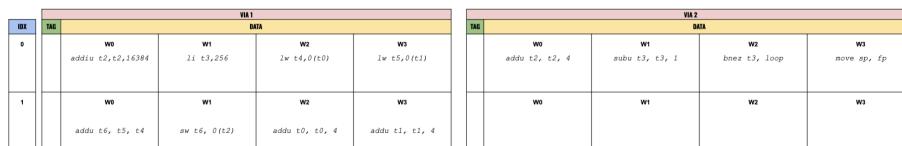


Figura 4.66: Cache de instrucciones durante la ejecución del loop.

Como antes del inicio del loop no existe ninguna directiva que alinee las instrucciones, utilizamos el programa *objdump* y analizamos en base al offset de cada instrucción respecto al inicio del programa. Esto hace que una vez cargado el benchmark en memoria, el índice de bloque al cual se debe cargar en cache sea distinto al mostrado en la Figura 4.66, pero no cambiara la forma en que están agrupadas las instrucciones.

Previamente a comenzar el análisis sobre el acceso a datos, nos basaremos en los valores obtenidos del registro *gp* al comienzo del loop mediante el uso del debugger *gdb*.

- **gp:** 0x5575e000 (1433788416 en decimal)

Respecto a los accesos a datos, vamos a tener que sucede prácticamente lo mismo que en el loop de la cache anterior, y esto se debe a que los conjuntos son de solo 2 vías y estamos tratando de acceder a 3 direcciones distintas que mapean al mismo índice durante una misma iteración, nuevamente vamos a tener un caso de trashing. Esto puede explicarse viendo que en la primera iteración sucederá lo siguiente:

1. Hay un **MISS** al tratar de obtener un dato de la dirección guardada en *t0* la cual es *gp* - 32728 (0x55756028) en la instrucción *lw t4, 0(t0)*. Esto hace que se traiga todo un

bloque de 16 bytes a cache y se lo guarde en el bloque de índice 0 (bit 4) en, digamos por ejemplo, la vía 0.

2. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en  $t1$  la cual es  $gp - 24536$  (0x5575A028) en la instrucción  $lw t5, 0(t1)$ . Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en el bloque de índice 0 (bit 4) en la vía que no uso el bloque anterior, es decir la vía 1.
3. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en  $t2$  la cual es  $gp - 16384$  (0x5575A028) en la instrucción  $sw t6, 0(t2)$ . Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en el bloque de índice 0 (bits 4) pero como ya utilicé ambas vías debo reemplazar uno de los bloques traídos, por ejemplo el de la vía 0 (en caso de tener una política de reemplazo LRU, esta sería la vía que utilicé hace más tiempo).

Esto lo podemos mostrar en la Figura 4.67. En el se indica en azul el bloque de direcciones de  $t0$ , en naranja el bloque de direcciones de  $t1$  y en verde el bloque de direcciones de  $t2$  que va a reemplazar al bloque de direcciones de  $t0$  por lo tanto lo ponemos en la misma vía y una fila mas abajo:

VIA 1				VIA 2				
RR	TAG	DATA			TAG	DATA		
0		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0	W1	W2	W3			
1						W0 <i>gp-24536</i>	W1 <i>gp-24532</i>	
						W2 <i>gp-24528</i>	W3 <i>gp-24524</i>	

Figura 4.67: Cache de instrucciones durante la ejecución del loop.

También vamos a tener que por tener la mitad de vías la cache va a llenarse luego de la mitad de iteraciones que la cache anterior, es decir que luego de apenas 5 iteraciones (en la 5ta se va a usar el índice siguiente ya que en cada iteración se le suman 4 a los valores guardados en  $t0$ ,  $t1$  y  $t2$ ) la cache ya va a estar llena. Mostramos en la Figura 4.68 (utilizando política de reemplazo LRU) como iría quedando la cache luego de 8 iteraciones:

VIA 1				VIA 2				
RR	TAG	DATA			TAG	DATA		
0		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
1		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
2		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
3		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
4		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
5		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
6		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
7		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			
8		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-32724</i> <i>gp-16340</i>	W2 <i>gp-32720</i> <i>gp-16336</i>	W3 <i>gp-32716</i> <i>gp-16332</i>			
		W0 <i>gp-32728</i> <i>gp-16344</i>	W1 <i>gp-24532</i> <i>gp-16340</i>	W2 <i>gp-24528</i> <i>gp-16336</i>	W3 <i>gp-24524</i> <i>gp-16332</i>			

Figura 4.68: Cache de instrucciones durante la ejecución del loop.

Esto se lee como el anterior en el que la dirección guardada en  $t0$  se indica con el color azul, la dirección guardada en  $t1$  se indica con naranja y la dirección guardada en  $t2$  se indica

con verde y se fue rellenando poniendo un bloque de direcciones debajo del otro para indicar el reemplazo del bloque que estaba anteriormente guardado en esa vía. Luego en la iteración siguiente (la novena) los bloques mapearan al índice 0 y luego de otras 4 iteraciones mapearán al índice 1 y así hasta completar las 256 iteraciones.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2 256    1      1     0     0     0     0     0     0      addu   t2 , t2 , 4
3 256    0      0     0     0     0     0     0     0      subu   t3 , t3 , 1
4 256    0      0     0     0     0     0     0     0      bnez   t3 , loop
5 .
6 .
7 .
8 256    0      0     0     0     0     0     0     0      # Destruimos el stack frame
          antes de retornar de main().
9 .
10 .
11 .
12 .
13 .
14 .
15 .
16 .
17 .

```

Figura 4.69: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 3 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2 1     1      1     1     1     1     0     0     0      lw    fp , 20(sp)
3 1     0      0     0     0     0     0     0     0      addiu sp , sp , 24
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 .
13 .
14 .
15 .
16 .
17 .

```

Figura 4.70: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 3 en la Cache 1 (Direct Map).

#### 4.4. Cache 3 - 4 Way Set Associative

Se dice que una memoria cache tiene estructura de Set Associative (4 Ways) si un bloque puede ser ubicado en cualquier parte de la caché. En particular, la memoria cache que vamos a analizar también posee bloques de 16 bytes cada uno y una memoria total de 64 bytes, es decir, con 4 bloques, por lo tanto, también podríamos llamar a la estructura de la cache como **Fully-Associative**. Esta memoria se organiza de la siguiente manera:

- Byte Offset: Addr[1:0]. 2 bytes para direccionar cada uno de los 4 bytes de la palabra.
- Word Offset: Addr[3:2]. 2 bytes para direccionar cada una de las 4 palabras del bloque.
- Index: No es necesario índice, ya que al ser *fully-associative* todos los bloques se analizan en paralelo.
- Tag: Addr[31:4]

TAG	DATA			
	W0	W1	W2	W3
	W0	W1	W2	W3
	W0	W1	W2	W3
	W0	W1	W2	W3
	W0	W1	W2	W3

Figura 4.71: Diseño ilustrativo simplificado de la estructura de la cache

Al ser una cache del tipo *split*, existen dos secciones iguales a la de la imagen, una para datos y otra para instrucciones.

Nuevamente en esta sección haremos referencias a los análisis previos de la Cache 1 (Direct Map) y Cache 2 (2-Way Set Associative), ya que existen ciertas secciones del código en que la estructura de la cache no es relevante para explicar el motivo de su comportamiento respecto a los **HIT** y **MISS**.

#### 4.4.1. Benchmark 0

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2
3 -- line 10 --
4 .
5 .
6 .
7 .
8 .
9 .
10 .
11 .
12 3   1   1   0   0   0   0   0   0   .cupload t9
13 .
14 1   0   0   0   0   0   0   0   0   addiu sp, sp, -24

```

Figura 4.72: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1   1   1   0   0   0   1   1   1   sw fp, 20(sp)
3 1   0   0   0   0   0   0   0   0   move fp, sp
4 1   0   0   0   0   0   1   1   1   .cprestore 0
5 .

```

Figura 4.73: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1   0   0   1   1   1   0   0   0   la t0, aligned
3 .

```

Figura 4.74: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2   1     1      1    0     0      0    0     0      0      1i   t1 , 100
3 100   0     0 100   1     1    0     0     0      0      loop: lw   t2 , 1024(t0)
4 100   0     0    0   0     0    0     0     0      0      subu t1 , t1 , 1
5 100   0     0    0   0     0    0     0     0      0      bnez t1 , loop
6 .
7 .
8 .      .      .      .      .      .      .      .      .      # Destruimos el stack frame
      .      .      .      .      .      .      .      .      .
      antes de retornar de main().                         #
      .      .      .      .      .      .      .      .      #

```

Figura 4.75: Bloque 4

Nuevamente vamos a tener que el loop principal de este benchmark entra en su totalidad dentro del bloque de instrucciones que se guarda en la cache de instrucciones luego del **MISS compulsivo** en la instrucción *li t1, 100*. Además vamos a tener que en cada iteración del loop, se accede únicamente al dato 1024+t0 sin ser modificado, entonces solo vamos a tener un primer **MISS compulsivo** al querer acceder a ese dato y luego en cada una de las siguientes 99 iteraciones vamos a tener **HIT**.

En las Figuras 4.76 y 4.77 podemos observar como queda la cache de instrucciones y la cache de datos en la primera iteración del loop (El dato en 1024+t0 va a mantenerse igual en todas las iteraciones):

TAG	INSTRUCCIONES			
	W0	W1	W2	W3
	<i>li t1, 100</i>	<i>lw t2, 1024(t0)</i>	<i>subu t1, t1 ,1</i>	<i>bnez t1, loop</i>
	W0	W1	W2	W3
	W0	W1	W2	W3
	W0	W1	W2	W3

Figura 4.76: Cache de instrucciones durante la ejecución del loop.

TAG	DATA			
	W0	W1	W2	W3
	<i>gp-31692</i>	<i>gp-31688</i>	<i>gp-31684</i>	<i>gp-31680</i>
	W0	W1	W2	W3
	W0	W1	W2	W3
	W0	W1	W2	W3

Figura 4.77: Cache de datos durante la ejecución del loop.

Nuevamente debemos aclarar que el contenido inicial de t0 que era *gp-32716* más el offset de 1024, da como resultado al ser guardado en la cache el bloque de 16 bytes que arranca en *gp-31696*.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr Dw  D1mw  DLmw
2 100    1     1     0     0     0   0   0     0
3 1     0     0     1     0     0   0   0     0
4 1     0     0     0     0     0   0   0     0
5 .
6 .
7 .
8 1     0     0     0     0     0   0   0     0
                                move sp , fp
                                lw fp , 20(sp)
                                addiu sp , sp , 24
# Para volver al sistema
operativo cargamos un código de retorno nulo.
#
move v0 , zero

```

Figura 4.78: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 0 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr Dw  D1mw  DLmw
2 1     1     1     0     0     0   0   0     0      jr  ra
3 .
4 .
5 .
6 .
7 .
8 .
.jend main
.rdata
.align 20
aligned:
.skip 8192

```

Figura 4.79: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 0 en la Cache 1 (Direct Map).

#### 4.4.2. Benchmark 1

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2
3 --- line 10 ---
4 .      .      .      .      .      .      .      .      .
5 .      .      .      .      .      .      .      .      .      .text
6 .      .      .      .      .      .      .      .      .      .align 2
7 .      .      .      .      .      .      .      .      .
8 .      .      .      .      .      .      .      .      .globl main
9 .      .      .      .      .      .      .      .      .ent main
10 .     .      .      .      .      .      .      .      .main:
11 .     .      .      .      .      .      .      .      .set noreorder
12 3     1     1     0     0     0     0     0     0     .cupload t9
13 .     .      .      .      .      .      .      .      .set nomacro
14 1     0     0     0     0     0     0     0     0     addiu sp, sp, -24

```

Figura 4.80: Bloque 1

Esta sección del benchmark es igual que la sección inicial el benchmark 1 de la cache 1 (Direct Mapped) y la cache 2 (2 Way Asociative). Para una explicacion en detalles de sobre el comportamiento de la memoria cache en esta seccion del codigo, dirigirse al bloque 1 del benchmark 1 de la cache 1.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2 1     1     1     0     0     0     1     1     1     sw  fp, 20(sp)
3 1     0     0     0     0     0     0     0     0     move fp, sp
4 1     0     0     0     0     0     1     1     1     .cprestore 0
5 .

```

Figura 4.81: Bloque 2

Esta sección del benchmark es igual que la sección inicial el benchmark 1 de la cache 1 (Direct Mapped) y la cache 2 (2 Way Asociative). Para una explicacion en detalles de sobre el comportamiento de la memoria cache en esta seccion del codigo, dirigirse al bloque 2 del benchmark 1 de la cache 1.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2 2     1     1     1     1     1     0     0     0     la  t0, aligned
3 1     0     0     0     0     0     0     0     0     li  t1, 100
4 .

```

Figura 4.82: Bloque 3

Esta sección del benchmark es igual que la sección inicial el benchmark 1 de la cache 1 (Direct Mapped) y la cache 2 (2 Way Asociative). Para una explicacion en detalles de sobre

el comportamiento de la memoria cache en esta sección del código, dirigirse al bloque 3 del benchmark 1 de la cache 1.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2 100    1     1 100    25    25  0     0     0   loop: lw t2 , 0(t0)
3 100    0     0 0     0     0  0     0     0   addu t0 , t0 , 4
4 100    0     0 0     0     0  0     0     0   subu t1 , t1 , 1
5 100    0     0 0     0     0  0     0     0   bnez t1 , loop
6 .
7 .
8 .      .     .  .     .     .  .     .     .   # Destruimos el stack frame
      .     .     .  .     .     .  .     .     .
      antes de retornar de main(). .
      .     .     .  .     .     .  .     .     .   #

```

Figura 4.83: Bloque 4

Vemos que en la primer instrucción de este loop vamos a tener un *MISS compulsivo* el cual al traer el bloque de instrucciones a memoria cache de 16 bytes vamos a tener que el loop entra en su totalidad en un único bloque, por lo tanto no habrá **MISSES** de acceso a instrucciones durante la ejecución de las 100 iteraciones del loop. Y al igual que en las 2 caches analizadas previamente vamos a tener que solo la instrucción *lw t2, 0(t0)* es la única que accede a memoria para realizar una lectura del dato que se encuentra en *t0+0* (que en las primeras 4 iteraciones será un **HIT** ya que fue guardado en el bloque de instrucciones anterior para luego tener un **MISS** luego de que se le hayan sumado 16 bytes a la posición original guardada en *t0* que era la de *.aligned*) y que debido a que en cada iteración a la dirección guardada en *t0* se le suman 4 vamos a tener que cada 3 iteraciones habrá un **MISS** ya que se intenta acceder a una dirección de memoria que no se encuentra guardada en el bloque de memoria de 16 bytes que se trae a la cache luego de un **MISS**. Luego como no se vuelve a reutilizar ningún bloque de datos una vez reemplazado ya que siempre se está sumando de a 4 a *t0* da igual la política de reemplazo utilizada que el trashing será inevitable debido a falta de capacidad en este benchmark. El resto de las instrucciones del loop no acceden a memoria de datos.

A continuación mostramos un diagrama de como quedaría la cache de instrucciones y datos luego de las primeras 16 iteraciones, a continuación se buscaría la dirección de memoria *gp-32652* y al no encontrarla se reemplazaría alguno de los bloques de la cache por el bloque de 16 bytes que contiene la dirección mencionada:

TAG	INSTRUCCIONES			
	W0	W1	W2	W3
	<i>lw t2, 0(t0)</i>	<i>addu t0, t0, 4</i>	<i>subu t1, t1, 1</i>	<i>bnez t1, loop</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>move sp, fp</i>	<i>lw fp, 20(sp)</i>	<i>addiu sp, sp, 24</i>	<i>move v0, zero</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>

Figura 4.84: Instrucciones en cache de instrucciones durante la ejecución del loop

TAG	DATA			
	W0	W1	W2	W3
	<i>gp - 32716</i>	<i>gp - 32712</i>	<i>gp - 32708</i>	<i>gp - 32704</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>gp - 32700</i>	<i>gp - 32696</i>	<i>gp - 32692</i>	<i>gp - 32688</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>gp - 32684</i>	<i>gp - 32680</i>	<i>gp - 32676</i>	<i>gp - 32672</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>gp - 32668</i>	<i>gp - 32664</i>	<i>gp - 32660</i>	<i>gp - 32656</i>

Figura 4.85: Datos en cache de datos durante la ejecución del loop

Mencionar que en t0 antes de iniciar el loop se había cargado la dirección de memoria de aligned la cual se representa como *gp-32716* por lo que el bloque de 16 bytes que lo incluye también tendrá las words que estén a 4, 8 y 12 bytes de distancia (representadas como *gp - 32712*, *gp - 32708* y *gp - 32704* respectivamente).

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 100 1 1 0 0 0 0 0 0 move sp , fp
3 1 0 0 1 1 0 0 0 lw fp , 20(sp )
4 1 0 0 0 0 0 0 0 addiu sp , sp , 24
5 .
6 .
7 .
8 1 0 0 0 0 0 0 0 # Para volver al sistema
    operativo cargamos un código de retorno nulo .
# move v0 , zero

```

Figura 4.86: Bloque 5

Esta sección del benchmark es igual que la sección inicial el benchmark 1 de la cache 1 (Direct Mapped) y la cache 2 (2 Way Asociativa). Para una explicacion en detalles de sobre el comportamiento de la memoria cache en esta sección del código, dirigirse al bloque 5 del benchmark 1 de la cache 1.

```

1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1 1 1 0 0 0 0 0 jr ra
3 . .
4 . .
5 . .
6 . .
7 . .
8 . .

    .end main
    .rdata
    .align 20
    aligned:
    .skip 8192

```

Figura 4.87: Bloque 6

Esta sección del benchmark es igual que la sección inicial el benchmark 1 de la cache 1 (Direct Mapped) y la cache 2 (2 Way Asociativa). Para una explicacion en detalles de sobre el comportamiento de la memoria cache en esta sección del código, dirigirse al bloque 6 del benchmark 1 de la cache 1.

#### 4.4.3. Benchmark 2

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2
3 --- line 10 ---
4   .   .   .   .   .   .   .   .   .
5   .   .   .   .   .   .   .   .   .   .text
6   .   .   .   .   .   .   .   .   .   .align 2
7   .   .   .   .   .   .   .   .   .
8   .   .   .   .   .   .   .   .   .   .globl main
9   .   .   .   .   .   .   .   .   .   .ent main
10  .   .   .   .   .   .   .   .   .   .main:
11  .   .   .   .   .   .   .   .   .   .set noreorder
12  3   1   1   0   0   0   0   0   0   .cupload t9
13  .   .   .   .   .   .   .   .   .   .set nomacro
14  1   0   0   0   0   0   0   0   0   addiu sp, sp, -24

```

Figura 4.88: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2   1   1   1   0   0   0   1   1   1   sw fp, 20(sp)
3   1   0   0   0   0   0   0   0   0   move fp, sp
4   1   0   0   0   0   0   1   1   1   .cprestore 0
5   .

```

Figura 4.89: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw  D1mw  DLmw
2   2   1   1   1   1   1   0   0   0   la t0, aligned
3   1   0   0   0   0   0   0   0   0   li t1, 100
4   .

```

Figura 4.90: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   Dlmw  DLmw
2 100    1     1  100   100   100   0     0     0   loop: lw t2 , 0(t0)
3 100    0     0   0     0     0     0     0     0   addu t0 , t0 , 16
4 100    0     0   0     0     0     0     0     0   subu t1 , t1 , 1
5 100    0     0   0     0     0     0     0     0   bnez t1 , loop
6 .
7 .
8 .      .      .      .      .      .      .      .      .      # Destruimos el stack frame
      .      .      .      .      .      .      .      .      .
      antes de retornar de main() .
      .      .      .      .      .      .      .      .      #

```

Figura 4.91: Bloque 4

Al igual que en el Benchmark anterior vamos a tener un primer **MISS compulsivo** en la primer instrucción del loop y como la totalidad del mismo esta implementado con 4 instrucciones vamos a tener que este bloque que se guarda en cache va a incluirlo en su totalidad por lo que no tendremos **MISSES** al buscar las instrucciones del loop durante las 100 iteraciones del mismo. También, de la misma forma que sucedía tanto en la Cache 1 como en la Cache 2, vamos a tener que la única instrucción que accede a memoria de datos para realizar una lectura es *lw t2, 0(t0)* en busca del dato guardado en la dirección de memoria que esta en *t0*. En el loop se le van sumando de a 16 en cada iteración a la dirección guardada en *t0*, vamos a tener que en cada iteración hay un **MISS** ya que el dato buscado en cada iteración estará en el bloque siguiente de datos al traído previamente a la cache, ya que estos tienen un tamaño de 4 words. Por lo tanto, los **MISSES** del loop serán siempre todos compulsivos porque si se tuviera una capacidad de 100 bloques de todas formas no se podrían evitar los **MISS** ya que siempre se accede a un bloque de memoria distinto sin repetir en cada iteración del loop.

En las Figuras 4.92 y 4.93 podemos observar una representación de los datos guardados en las memorias cache de datos y de instrucciones en las primeras 4 iteraciones.

TAG	INSTRUCCIONES			
	W0	W1	W2	W3
	<i>lw t2, 0(t0)</i>	<i>addu t0, t0, 4</i>	<i>subu t1, t1, 1</i>	<i>bnez t1, loop</i>
	<i>move sp, fp</i>	<i>lw fp, 20(sp)</i>	<i>addiu sp, sp, 24</i>	<i>move v0, zero</i>
	W0	W1	W2	W3
	W0	W1	W2	W3

Figura 4.92: Cache de instrucciones durante la ejecución del loop

TAG	DATA			
	W0	W1	W2	W3
	gp - 32716	gp - 32712	gp - 32708	gp - 32704
	W0	W1	W2	W3
	gp - 32700	gp - 32696	gp - 32692	gp - 32688
	W0	W1	W2	W3
	gp - 32684	gp - 32680	gp - 32676	gp - 32672
	W0	W1	W2	W3
	gp - 32668	gp - 32664	gp - 32660	gp - 32656

Figura 4.93: Cache de datos durante la ejecución del loop

A pesar de que esto mismo suceda en el Benchmark estudiado previamente, No podemos aprovechar para nada la cache ya que en cada iteración vamos a tener un **MISS**.

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 100    1     1     0     0     0     0     0     0      move  sp , fp
3   0     0     1     1     1     0     0     0      lw   fp , 20(sp)
4   0     0     0     0     0     0     0     0      addiu sp , sp , 24
5   .
6   .
7   .
8   .
# Para volver al sistema
operativo cargamos un código de retorno nulo.
#
move v0 , zero

```

Figura 4.94: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 1 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 1     1     1     0     0     0     0     0     0      jr   ra
3   .
4   .
5   .
6   .
7   .
8   .
.end main
.rdata
.align 20
aligned:
.skip 8192

```

Figura 4.95: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 1 en la Cache 1 (Direct Map).

#### 4.4.4. Benchmark 3

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2
3 — line 10 ——————
4 .     .     .     .     .     .     .     .     .
5 .     .     .     .     .     .     .     .     .     .text
6 .     .     .     .     .     .     .     .     .     .align 2
7 .     .     .     .     .     .     .     .     .
8 .     .     .     .     .     .     .     .     .     .globl main
9 .     .     .     .     .     .     .     .     .     .ent main
10    .     .     .     .     .     .     .     .     .     main:
11    .     .     .     .     .     .     .     .     .     .set noreorder
12    3     1     1     0     0     0     0     0     0     .cupload t9
13    .     .     .     .     .     .     .     .     .     .set nomacro
14    1     0     0     0     0     0     0     0     0     addiu sp, sp, -24

```

Figura 4.96: Bloque 1

Este bloque se comporta de la misma manera que el Bloque 1 del Benchmark 3 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 1     1     1     0     0     0     1     1     1     sw fp, 20(sp)
3 1     0     0     0     0     0     0     0     0     move fp, sp
4 1     0     0     0     0     0     1     1     1     .cprestore 0
5 .

```

Figura 4.97: Bloque 2

Este bloque se comporta de la misma manera que el Bloque 2 del Benchmark 3 en la Cache 1 (Direct Map).

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLmw
2 2     1     1     1     1     1     0     0     0     la t0, aligned0
3 2     0     0     1     0     0     0     0     0     la t1, aligned1

```

Figura 4.98: Bloque 3

Este bloque se comporta de la misma manera que el Bloque 3 del Benchmark 3 en la Cache 1 (Direct Map).

	Ir	I1mr	ILmr	Dr	D1mr	DLmr	Dw	D1mw	DLmw	
2	2	1	1	1	0	0	0	0	0	la t2 , aligned2
3	1	0	0	0	0	0	0	0	0	li t3 , 256
4	256	0	0	256	64	64	0	0	0	loop: lw t4 , 0(t0)
5	256	0	0	256	64	64	0	0	0	lw t5 , 0(t1)
6	256	1	1	0	0	0	0	0	0	addu t6 , t5 , t4
7	256	0	0	0	0	0	256	64	64	sw t6 , 0(t2)
8	256	0	0	0	0	0	0	0	0	addu t0 , t0 , 4
9	256	0	0	0	0	0	0	0	0	addu t1 , t1 , 4

Figura 4.99: Bloque 4

En esta caché, al analizar los accesos a instrucciones al igual que como sucedía en la cache anteriormente analizadas, tenemos que la primer instrucción del bloque 4 de código es la segunda instrucción en la que se expandió la pseudoinstrucción *la, t2, aligned2*. Esta misma genera un **MISS compulsivo** que cargara en cache de instrucciones esta misma y las siguientes tres instrucciones, por ese motivo veremos tres **HITS** consecutivos.

Similarmente, en el segundo conjunto de 4 instrucciones tenemos un **MISS compulsivo** en la instrucción *addu t6, t5, t4* y las siguientes 3 instrucciones serán **HIT's**.

La cache de instrucciones al comienzo del loop sera:

TAG	INSTRUCCIONES			
	W0	W1	W2	W3
	<i>addiu t2,t2,16384</i>	<i>li t3,256</i>	<i>lw t4,0(t0)</i>	<i>lw t5,0(t1)</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>addu t6, t5, t4</i>	<i>sw t6, 0(t2)</i>	<i>addu t0, t0, 4</i>	<i>addu t1, t1, 4</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>addu t2, t2, 4</i>	<i>subu t3, t3, 1</i>	<i>bnez t3, loop</i>	<i>move sp, fp</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>

Figura 4.100: Cache de instrucciones durante la ejecución del loop.

Como antes del inicio del loop no existe ninguna directiva que alinee las instrucciones, utilizamos el programa *objdump* y analizamos en base al offset de cada instrucción respecto al inicio del programa. Esto hace que una vez cargado el benchmark en memoria, el índice de bloque al cual se debe cargar en cache sea distinto al mostrado en la Figura 4.100, pero no cambiara la forma en que están agrupadas las instrucciones.

Previamente a comenzar el análisis sobre el acceso a datos, nos basaremos en los valores obtenidos del registro *gp* al comienzo del loop mediante el uso del debugger *gdb*.

- **gp:** 0x5575e000 (1433788416 en decimal)

Si vemos lo que sucede con los accesos a datos en esta cache asociativa por conjuntos de 4 vías (Equivalente a una Fully associative en este caso) vamos a tener que por fin no va a ocurrir el trashing que pasaba en las caches anteriores, debido a que en este caso no habrá bits de índice y los bloques de la memoria principal pueden mapearse a cualquier bloque de la cache. De este modo vamos a tener que durante la primera iteración sucederá lo siguiente:

1. Hay un **MISS** al tratar de obtener un dato de la dirección guardada en *t0* la cual es *gp* - 32728 (0x55756028) en la instrucción *lw t4, 0(t0)*. Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en alguno de los 4 bloques de la cache, por ejemplo el bloque 0 (por ser fully associative podría mapear a cualquiera de los bloques de la cache).
2. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en *t1* la cual es *gp* - 24536 (0x55758028) en la instrucción *lw t5, 0(t1)*. Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en alguno de los 4 bloques de la cache, por ejemplo el bloque 1 (por ser fully associative podría mapear a cualquiera de los bloques de la cache).
3. Hay un **MISS** al tratar de obtener un dato en la dirección guardada en *t2* la cual es *gp* - 16384 (0x5575A028) en la instrucción *sw t6, 0(t2)*. Esto hace que se traiga todo un bloque de 16 bytes a cache y se lo guarde en alguno de los 4 bloques de la cache, por ejemplo el bloque 3 (por ser fully associative podría mapear a cualquiera de los bloques de la cache).

Esto lo podemos mostrar en la Figura 4.101 en el que se indica en azul el bloque de direcciones de *t0*, en naranja el bloque de direcciones de *t1* y en verde el bloque de direcciones de *t2*:

TAG	DATA			
	W0	W1	W2	W3
	<i>gp-32728</i>	<i>gp-32724</i>	<i>gp-32720</i>	<i>gp-32716</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>gp-24536</i>	<i>gp-24532</i>	<i>gp-24528</i>	<i>gp-24524</i>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<b>W0</b>	<b>W1</b>	<b>W2</b>	<b>W3</b>
	<i>gp-16344</i>	<i>gp-16340</i>	<i>gp-16336</i>	<i>gp-16332</i>

Figura 4.101: Cache de datos inicialmente en la ejecucion del loop

Debido a que con esta cache no hubo trashing vamos a tener que en las primeras 4 iteraciones solo vamos a tener un **MISS compulsivo** al cargar los datos la primera vez y luego habrá **HITs** hasta que en la 5ta iteración en la que se sumó ya 16 a las direcciones guardadas originalmente en t0, t1 y t2 vamos a tener que cargar otros 3 bloques de 16 bytes teniendo nuevamente otros 3 **MISSES compulsivos** en esa iteración, pero teniendo **HITs** en las siguientes 3 iteraciones. Este comportamiento de 1 **MISS** y 3 **HITs** se va a seguir repitiendo a lo largo de las 256 iteraciones, llenando la cache en el proceso, pero logrando reducir a 1/4 los **MISSES** con respecto a lo sucedido en las caches anteriormente evaluadas.

En el diagrama de abajo mostramos como podría quedar la cache en la 5ta iteración, con todos los bloques ocupados con datos usados en loop. Se ocupa el bloque no utilizado previamente para el bloque de datos (nuevo) de t0 (azul) y los bloques de datos nuevos de t1 (naranja) y t2 (verde) reemplazan bloques usados previamente (en particular podrían reemplazar los bloques que tenían los datos viejos del bloque de datos de t0 y t1 y esto lo representamos escribiendo las direcciones nuevas un renglón mas abajo que las direcciones viejas, pero dentro del mismo bloque):

TAG	DATA			
	W0	W1	W2	W3
	<i>gp-32728</i> <i>gp-24520</i>	<i>gp-32724</i> <i>gp-24516</i>	<i>gp-32720</i> <i>gp-24512</i>	<i>gp-32716</i> <i>gp-24508</i>
	<b>W0</b>  <i>gp-24536</i> <i>gp-16328</i>	<b>W1</b>  <i>gp-24532</i> <i>gp-16324</i>	<b>W2</b>  <i>gp-24528</i> <i>gp-16320</i>	<b>W3</b>  <i>gp-24524</i> <i>gp-16316</i>
	<b>W0</b>  <i>gp-32712</i>	<b>W1</b>  <i>gp-32708</i>	<b>W2</b>  <i>gp-32704</i>	<b>W3</b>  <i>gp-32700</i>
	<b>W0</b>  <i>gp-16344</i>	<b>W1</b>  <i>gp-16340</i>	<b>W2</b>  <i>gp-16336</i>	<b>W3</b>  <i>gp-16332</i>

Figura 4.102: Cache de datos luego de 5 iteraciones del loop

```

1 Ir   I1mr  ILmr  Dr   D1mr  DLmr  Dw   D1mw  DLnw
2 256    1      1     0     0      0     0     0      0      addu   t2 , t2 , 4
3 256    0      0     0     0      0     0     0      0      subu   t3 , t3 , 1
4 256    0      0     0     0      0     0     0      0      bnez   t3 , loop
5 .
6 .
7 .
8 256    0      0     0     0      0     0     0      0      move   sp , fp

```

Figura 4.103: Bloque 5

Este bloque se comporta de la misma manera que el Bloque 5 del Benchmark 3 en la Cache 1 (Direct Map).

```
1 Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
2 1   1     1   1   1     1   0   0   0
3 1   0     0   0   0     0   0   0   0      lw fp , 20(sp)
4 .   .     .   .   .     .   .   .   .
5 .   .     .   .   .     .   .   .   .      # Para volver al sistema
6   operativo cargamos un codigo de retorno nulo.
7   .
8   .
9   .
10  .
11  .
12  .
13  .
14  .
15  .
16  .
17 — line 56 —
```

Figura 4.104: Bloque 6

Este bloque se comporta de la misma manera que el Bloque 6 del Benchmark 3 en la Cache 1 (Direct Map).

## 5. Compilación y Ejecución

En esta sección vamos a mostrar los pasos de compilación y ejecución realizados para poder obtener las anotaciones de Cachegrind sobre los archivos fuente de cada Benchmark para cada Cache.

El comando \$ make compila todos los archivos .S y genera los ejecutables para poder utilizar en los pasos siguientes. El comando:

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=A –D1=B –LL=C /root/benchmarks-tp2/benchmark-bX

Corre Cachegrind con los flags de configuración:

- I1=A,B,C
- D1=A,B,C
- LL=A,B,C

Siendo A el tamaño de la cache, B la asociatividad de la cache (cantidad de vías), C la cantidad de bytes por bloque. Esto lo corre sobre el ejecutable benchmark-bX generando un archivo cachegrind.out.F, Siendo X el benchmark deseado, Y F el numero identificador de proceso (PID) del proceso que se ejecuto el Cachegrind.

Por ultimo, corremos el comando:

- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.F/root/benchmarks-tp2/benchmark-bX.S >c1\_bX.txt

para poder utilizar cg\_annotate, este nos imprime por salida estándar(nosotros lo redirigimos a un .txt) el archivo.S con anotaciones de accesos. Para esto, cg\_annotate utiliza el archivo cachegrind.out.F sobre el archivo.S (el compilado en el ejecutable previamente).

A continuación vamos a ver los comandos específicos para cada cache con cada uno de los benchmarks.

## 5.1. Cache 1

### 5.1.1. Benchmark 0

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,1,16 –D1=64,1,16 –LL=64,1,16 /root/benchmarks-tp2/benchmark-b0
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.727 /root/benchmarks-tp2/benchmark-b0.S >c1\_b0.txt

### 5.1.2. Benchmark 1

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,1,16 –D1=64,1,16 –LL=64,1,16 /root/benchmarks-tp2/benchmark-b1
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.747 /root/benchmarks-tp2/benchmark-b1.S >c1\_b1.txt

### 5.1.3. Benchmark 2

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,1,16 –D1=64,1,16 –LL=64,1,16 /root/benchmarks-tp2/benchmark-b2
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.755 /root/benchmarks-tp2/benchmark-b2.S >c1\_b2.txt

### 5.1.4. Benchmark 3

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,1,16 –D1=64,1,16 –LL=64,1,16 /root/benchmarks-tp2/benchmark-b3
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.792 /root/benchmarks-tp2/benchmark-b3.S >c1\_b3.txt

## 5.2. Cache 2

### 5.2.1. Benchmark 0

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,2,16 –D1=64,2,16 –LL=64,2,16 /root/benchmarks-tp2/benchmark-b0
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.796 /root/benchmarks-tp2/benchmark-b0.S >c2\_b0.txt

### 5.2.2. Benchmark 1

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,2,16 –D1=64,2,16 –LL=64,2,16 /root/benchmarks-tp2/benchmark-b1
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.802 /root/benchmarks-tp2/benchmark-b1.S >c2\_b1.txt

### 5.2.3. Benchmark 2

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,2,16 –D1=64,2,16 –LL=64,2,16 /root/benchmarks-tp2/benchmark-b2
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.808 /root/benchmarks-tp2/benchmark-b2.S >c2\_b2.txt

### 5.2.4. Benchmark 3

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,2,16 –D1=64,2,16 –LL=64,2,16 /root/benchmarks-tp2/benchmark-b3
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.812 /root/benchmarks-tp2/benchmark-b3.S >c2\_b3.txt

### 5.3. Cache 3

#### 5.3.1. Benchmark 1

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,4,16 –D1=64,4,16 –LL=64,4,16 /root/benchmarks-tp2/benchmark-b0
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.818 /root/benchmarks-tp2/benchmark-b0.S >c3\_b0.txt

#### 5.3.2. Benchmark 1

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,4,16 –D1=64,4,16 –LL=64,4,16 /root/benchmarks-tp2/benchmark-b1
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.827 /root/benchmarks-tp2/benchmark-b1.S >c3\_b1.txt

#### 5.3.3. Benchmark 2

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,4,16 –D1=64,4,16 –LL=64,4,16 /root/benchmarks-tp2/benchmark-b2
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.831 /root/benchmarks-tp2/benchmark-b2.S >c3\_b2.txt

#### 5.3.4. Benchmark 3

- \$ /opt/valgrind/bin/valgrind –tool=cachegrind –I1=64,4,16 –D1=64,4,16 –LL=64,4,16 /root/benchmarks-tp2/benchmark-b3
- \$ /opt/valgrind/bin/cg\_annotate cachegrind.out.837 /root/benchmarks-tp2/benchmark-b3.S >c3\_b3.txt

## 6. Cálculos de Misses y Miss rate

### 6.1. Cache 1

#### 6.1.1. Benchmark 0

En este benchmark tenemos por un lado que se realizaron un total de 413 accesos a la memoria de instrucciones, de los cuales solo 5 son **MISSES** y todos son compulsivos, por lo tanto:

- **Miss Rate:**  $5/413 \cong 0,0121$
- **Hit Rate:**  $408/413 \cong 0,987$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Por otro lado si vemos la cache de datos tenemos que se realizaron un total de 104 accesos a la memoria de datos (102 fueron para leer datos y 2 accesos fueron para escribir) de los cuales hubieron 2 **MISSES** de lectura compulsivos y 2 **MISSES** de escritura compulsivos

- **Miss Rate:**  $4/104 \cong 0,0385$
- **Hit Rate:**  $100/104 \cong 0,9615$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

#### 6.1.2. Benchmark 1

El benchmark 1 en cambio vimos que si analizamos la cache de instrucciones obtenemos un total de 514 accesos a la memoria de instrucciones y hubo un total de 6 **MISSES** los cuales fueron todos compulsivos.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Si vemos la cache de datos obtenemos que hubo en total 104 accesos (nuevamente 102 fueron para lectura y 2 fueron para escritura) y un total de 27 **MISSES** de lectura (de los cuales 26 fueron compulsivos, ya que en el loop siempre se accede a bloques que nunca estuvieron en memoria y el único **MISS** no compulsivo es al final del programa cuando se llama a la instrucción `lw fp, 20(sp)` ya que el dato `20(sp)` había sido guardado anteriormente

en la cache pero fue reemplazado cuando se ejecutó el loop) y 2 **MISSES** de escritura, ambos compulsivos.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** cada cuatro iteraciones, es decir, cada cuatro accesos a memoria.

- **Miss Rate:**  $29/104 \cong 0,2788$
- **Hit Rate:**  $75/104 \cong 0,7212$
- **Miss Rate en régimen:** 0.25
- **Hit Rate en régimen:** 0.75

### 6.1.3. Benchmark 2

Al igual que en el benchmark anterior tenemos que la cantidad total de accesos a la memoria de instrucciones fue 514 y hubo un total de 6 **MISSES**, los cuales al igual que en el benchmark 1 fueron todos compulsivos.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Este benchmark cuando se ejecuta sobre esta cache con estructura Direct Mapped Realiza un total de 104 accesos a la memoria de datos (102 lecturas y 2 escrituras) los cuales fueron lamentablemente todos **MISS**. Solamente hubo un **MISS** no compulsivo el cual se dió por el mismo motivo que el único **MISS** no compulsivo fue al hacer el llamado a la instrucción `lw fp, 20(sp)` que no puede acceder al dato `20(sp)` el cual fue cargado antes de acceder al loop principal a la cache de datos y luego fue reemplazada al ser ejecutado el loop.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** en cada iteración, es decir, en cada acceso a memoria.

- **Miss Rate:**  $104/104 = 1$
- **Hit Rate:**  $0/104 = 0$
- **Miss Rate en régimen:** 1
- **Hit Rate en régimen:** 0

### 6.1.4. Benchmark 3

En este benchmark se realizaron 2578 accesos a memoria de instrucciones y hubo un total de 7 **MISSES** de los cuales fueron todos compulsivos.

- **Miss Rate:**  $7/2578 = 0.0027$
- **Hit Rate:**  $2571/2578 = 0.997$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Al analizar los accesos a memoria de datos, vemos que hubo un total de 516 accesos para escritura y 258 accesos de lectura, contabilizando un total de 774 accesos a memoria. Como la cache entra en un *trashing* (explicado en la sección 4.2.4), todos los accesos en régimen son **MISSES**. Los únicos dos accesos que son **HIT** son los accesos que cargan las etiquetas previo al loop.

Respecto al régimen, es decir, mirando solamente al loop, hay 3 accesos a datos los cuales son todos **MISS** en cada iteración , es decir, que todos los accesos en el loop son **MISSES**.

- **Miss Rate:**  $772/774 = 0.997$
- **Hit Rate:**  $2/774 = 0.003$
- **Miss Rate en régimen:** 1
- **Hit Rate en régimen:** 0

## 6.2. Cache 2

### 6.2.1. Benchmark 0

Al ver la cantidad de accesos a la memoria de instrucciones que realiza este benchmark sobre esta Cache 2WA vemos que hubo un total de 413 accesos, de los cuales solo hubieron 5 **MISSES** y fueron todos compulsivos.

- **Miss Rate:**  $5/413 \cong 0,0121$
- **Hit Rate:**  $408/413 \cong 0,987$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Ahora si vemos los accesos a la memoria de datos vemos que hubo un total de 104 accesos a la memoria de datos (102 lecturas y 2 escrituras) y al igual que como sucedió con la cache anterior hay un total de 2 **MISSES** de lectura, ambos compulsivos y 2 **MISSES** de escritura, ambos también compulsivos.

- **Miss Rate:**  $4/104 \cong 0,0385$
- **Hit Rate:**  $100/104 \cong 0,9615$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

### 6.2.2. Benchmark 1

Nuevamente vamos a tener que el comportamiento en cuanto a accesos a memoria de instrucciones en este benchmark con esta cache es idéntico al de la cache anterior, es decir que tendremos un total de 514 accesos y solo 6 **MISSES**, todos compulsivos.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Esto también sucede con los accesos a la memoria de datos, ya que al igual que en la cache anterior tenemos un total de 104 accesos a memoria (102 lecturas y 2 escrituras) con 27 **MISSES** de lectura (y solo 1 de esos no compulsivo al querer obtener el dato 20(sp) en la instrucción *lw fp, 20(sp)* ya que este dato fue guardado inicialmente en la cache y fue reemplazado al ser ejecutado el loop) y tenemos 2 **MISSES** de escritura, ambos compulsivos.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** cada cuatro iteraciones, es decir, cada cuatro accesos a memoria.

- **Miss Rate:**  $29/104 \cong 0,2788$
- **Hit Rate:**  $75/104 \cong 0,7212$
- **Miss Rate en régimen:** 0.25
- **Hit Rate en régimen:** 0.75

### 6.2.3. Benchmark 2

En este benchmark vamos a tener que de nuevo el comportamiento es el mismo que con la cache anterior, es decir que hay un total de 514 accesos a la memoria de instrucciones y 6 **MISSES**, todos compulsivos.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

También si vemos los accesos a la memoria de datos vemos que sucede lo mismo que en la cache anterior, es decir que habrán 104 accesos a memoria de datos, de los cuales 102 son lecturas y 2 son escrituras y de estos ninguno se salvará de ser **MISS**. Solo habrá un **MISS** no compulsivo al igual que en la cache anterior, al querer acceder al dato de *20(sp)* guardado antes de correr el loop con la instrucción *sw fp, 20(sp)* (guardada en el segundo bloque de instrucciones cargado en cache) y que luego de haber corrido el loop y que se hayan reemplazado todos los bloques de la memoria de datos ya no estaría disponible para cuando se ejecute la instrucción *lw fp, 20(sp)*.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** en cada iteración, es decir, en cada acceso a memoria.

- **Miss Rate:**  $104/104 = 1$
- **Hit Rate:**  $0/104 = 0$
- **Miss Rate en régimen:** 1
- **Hit Rate en régimen:** 0

### 6.2.4. Benchmark 3

En este benchmark se realizaron 2578 accesos a memoria de instrucciones y hubo un total de 7 **MISSES** de los cuales fueron todos compulsivos.

- **Miss Rate:**  $7/2578 = 0.0027$
- **Hit Rate:**  $2571/2578 = 0.997$

- **Miss Rate en régimen:** 0

- **Hit Rate en régimen:** 1

Al analizar los accesos a memoria de datos, vemos que hubo un total de 516 accesos para escritura y 258 accesos de lectura, contabilizando un total de 774 accesos a memoria. Como la cache entra en un *trashing* (explicado en la sección 4.3.4), todos los accesos en régimen son **MISSES**. Los únicos dos accesos que son **HIT** son los accesos que cargan las etiquetas previo al loop.

Respecto al régimen, es decir, mirando solamente al loop, hay 3 accesos a datos los cuales son todos **MISS** en cada iteración , es decir, que todos los accesos en el loop son **MISSES**.

- **Miss Rate:**  $772/774 = 0.997$

- **Hit Rate:**  $2/774 = 0.003$

- **Miss Rate en régimen:** 1

- **Hit Rate en régimen:** 0

### 6.3. Cache 3

#### 6.3.1. Benchmark 0

Este benchmark en la cache Fully Asociative no tuvo un comportamiento distinto del visto en las caches analizadas previamente. Se vio nuevamente que el total de accesos a memoria de instrucciones fue de 413 y hubieron 5 **MISSES**, todos compulsivos.

- **Miss Rate:**  $5/413 \cong 0,0121$
- **Hit Rate:**  $408/413 \cong 0,987$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

También en los accesos a datos para lectura y escritura el comportamiento fue el mismo que el observado previamente, es decir que hubieron un total de 104 accesos a memoria de datos, con 102 lecturas de las cuales 2 fueron **MISSES** (compulsivos) y 2 escrituras, ambas **MISSES** compulsivos.

- **Miss Rate:**  $4/104 \cong 0,0385$
- **Hit Rate:**  $100/104 \cong 0,9615$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

#### 6.3.2. Benchmark 1

Vemos en este benchmark que el total de accesos a la memoria de instrucciones fue 514 y hubieron 6 **MISSES**, todos compulsivos al igual que como sucedió en las caches anteriores.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Y si analizamos los accesos a memoria de datos vemos que hubieron 104 accesos, con 102 lecturas de las cuales 27 fueron **MISSES** (y solo uno no fue compulsivo, al igual que con las caches anteriores sucedió lo mismo con el dato de 20(sp) que fue almacenado antes de correr el loop y luego al querer volver a acceder a este después de correr el loop ya había sido reemplazado, por lo que fue un **MISS** en régimen) y 2 escrituras, ambos **MISSES** compulsivos.

De igual manera que en la cache 1 y la cache 2, todos los misses que suceden dentro del loop se deben a cargar una nueva parte del array nunca accedida previamente a cache, por lo que el miss-rate en régimen es nulo.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** cada cuatro iteraciones, es decir, cada cuatro accesos a memoria.

- **Miss Rate:**  $29/104 \cong 0,2788$
- **Hit Rate:**  $75/104 \cong 0,7212$
- **Miss Rate en régimen:** 0.25
- **Hit Rate en régimen:** 0.75

### 6.3.3. Benchmark 2

Sorprendentemente en este benchmark tampoco hubo diferencias en cuanto accesos a datos e instrucciones se refiere cuando comparamos lo obtenido en esta cache con lo obtenido en las 2 caches analizadas previamente. Hubo un total de 514 accesos a la memoria de instrucciones y 6 **MISSES**, todos compulsivos.

- **Miss Rate:**  $6/514 \cong 0,0117$
- **Hit Rate:**  $508/514 \cong 0,9883$
- **Miss Rate en regimen:** 0
- **Hit Rate en regimen:** 1

Los accesos a la memoria de datos también fueron iguales en este benchmark que los obtenidos con las caches anteriores, es decir que hubo un total de 104 accesos, 102 lecturas las cuales fueron todas **MISS** y nuevamente solo 1 de esos **MISSES** fue en régimen (al querer obtener el dato *20(sp)* como en las caches anteriores el dato aunque estuvo guardado en algún momento en la memoria cache, luego de ser ejecutado el loop fue reemplazado y ya no estaba disponible para cuando se lo quiso acceder con la instrucción *lw fp, 20(sp)*) y hubieron también 2 escrituras, ambos **MISSES** compulsivos.

Respecto al régimen, es decir, mirando solamente al loop, hay un **MISS** en cada iteración, es decir, en cada acceso a memoria.

- **Miss Rate:**  $104/104 = 1$
- **Hit Rate:**  $0/104 = 0$
- **Miss Rate en regimen:** 1
- **Hit Rate en regimen:** 0

#### 6.3.4. Benchmark 3

En este benchmark se realizaron 2578 accesos a memoria de instrucciones y hubo un total de 7 **MISSES** de los cuales fueron todos compulsivos.

- **Miss Rate:**  $7/2578 = 0.0027$
- **Hit Rate:**  $2571/2578 = 0.997$
- **Miss Rate en régimen:** 0
- **Hit Rate en régimen:** 1

Al analizar los accesos a memoria de datos, vemos que hubo un total de 516 accesos para escritura y 258 accesos de lectura, contabilizando un total de 774 accesos a memoria. Habrá un total de 130 **MISSES** de lectura de datos y 2 **MISSES** de escritura, pero a diferencia de las otras dos caches previas, la cache 3 no entra en *trashing* porque todos los bloques requeridos para una iteración entran en cache simultáneamente, es decir, sin ser reemplazados.

Respecto al régimen, es decir, mirando solamente al loop, en cada iteración se acceden a tres datos, de los cuales siempre son miss (los tres simultáneamente) cada 4 iteraciones, es decir, cada 12 accesos a memoria, 3 son miss.

- **Miss Rate:**  $196/774 = 0.997$
- **Hit Rate:**  $578/774 = 0.003$
- **Miss Rate en régimen:** 0.25
- **Hit Rate en régimen:** 0.75

## 7. Conclusión

El análisis en detalle de los *benchmarks* nos permitió sacar algunas conclusiones interesantes.

La primer conclusión es que los *benchmarks* no son siempre una buena referencia para medir el desempeño de una memoria cache, ya que dependiendo de la forma en que estén escritos pueden alterar notablemente el resultado para cierto tipo de estructura.

La segunda conclusión es que en algunos tipos de accesos pueden inutilizar completamente la cache, como en caso del *benchmark 2*, en que los datos se acceden de manera que no sirvan los previamente cacheados, haciendo que ninguna de las tres estructuras de cache tenga buen desempeño.

Por ultimo, la cache fully-associative fue la cache mas estable, dando muy buenos resultados principalmente en el *benchmark 3*, donde el resto falla.