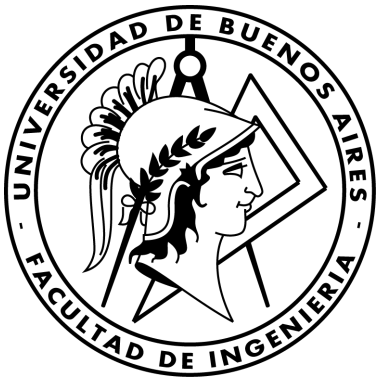

Universidad de Buenos Aires

Facultad de Ingeniería

2do Cuatrimestre de 2020

[86.37 / 66.20] Organización de Computadoras

Curso 2

Trabajo práctico 2

Memoria Caché

Padrón	Alumno	Email
103442	Lovera, Daniel	dlovera@fi.uba.ar
102914	More, Agustín	amore@fi.uba.ar
99846	Torresetti, Lisandro	ltorresetti@fi.uba.ar

Repositorio: https://github.com/DanieLovera/Orga

Índice

1. Objetivos

El trabajo práctico consiste en la simulación del comportamiento de una memoria caché asociativa por conjuntos, bajo la política `LRU`, bajo la política de escritura `WB/WA`.

2. Introducción

La finalidad de una memoria caché consiste en reducir la cantidad de accesos de lecturas y escrituras de la memoria principal, poniéndose como intermediario entre la memoria principal y la `CPU`. Al tratarse de una memoria que es menor a la memoria principal, existirán colisiones que deben manejarse. Particularmente, para el presente trabajo se cuenta con las políticas reemplazo `LRU` (*Least Recently Used*), que al momento de reemplazar algún registro, se reemplazará el que menos se haya solicitado últimamente. Luego, para la política de escritura, se utiliza `WB/WA` (*Write-Back/Write-Allocate*), es decir, cuando se tiene que escribir, primero se escribe en la memoria caché, y cuando sea necesario reemplazar ese registro de la caché, se escribirá finalmente en la memoria principal.

3. Detalles de implementación

3.1. Address Parser

Este módulo fue diseñado con la intención de que sea el encargado de convertir una dirección de memoria en cada campo especifico de la memoria caché **tag, set, offset**.
La interfaz del usuario queda definida como:

```
1    unsigned int address_parser_tag(unsigned int address
2          , unsigned int block_size, unsigned int total_sets);
3    unsigned int address_parser_set(unsigned int address
4          , unsigned int block_size, unsigned int total_sets);
5    unsigned int address_parser_offset(unsigned int address
6          , unsigned int block_size);
```

En donde cada función devuelve el valor del campo correspondiente

3.2. Memory

El módulo de memoria simula ser la memoria principal con un arreglo de caracteres de tamaño **64 Kb**, además implementa las operaciones de lectura y escritura necesarias. Las funciones mas importantes correspondientes a este modulo son:

```
1    void memory_write(memory_t *self, char *data
2         , int address, unsigned int data_size);
3    void memory_read(memory_t *self, char *data_buffer
4         , int address, unsigned int data_size);
```

- La función **memory_write** escribe un bloque entero de datos en memoria, para eso requiere recibir los datos, y la dirección de inicio en donde se van a escribir, el tamaño de los datos determina cuantos datos se van a copiar.
- La función **memory_read** es análoga a la anterior con la salvedad que recibe un buffer en donde se van a guardar los datos que hayan sido leídos de memoria.

3.3. Cache

Es finalmente el módulo mas importante para la simulación de la memoria caché SA/LRU, se utilizan el **Address parser** y **Memory** para completar el programa principal.

A su vez la cache esta compuesta de dos estructuras de datos más pequeñas **struct set** y **struct block**, y se hace uso de memoria dinámica para poder implementar una cache que cambie dinámicamente dependiendo de la elección del usuario. Se define La interfaz del usuario como:

```
1  void cache_init(unsigned int capacity, unsigned int ways_number,
       ↪ unsigned int block_size);
2  void cache_uninit();
3  void init();
4  unsigned int find_set(int address);
5  unsigned int find_lru(int setnum);
6  unsigned int is_dirty(int way, int setnum);
7  void read_block(int blocknum);
8  void write_block(int way, int setnum);
9  char read_byte(int address);
10 void write_byte(int address, char value);
11 int get_miss_rate();
```

3.3.1. Funciones importantes de implementación de cache.

- La implementación de la política LRU se realizo en la función **find_lru**, para ello cada bloque de memoria cache cuenta con un campo llamado **lru_counter** el cual cada vez que es accedido es aumentado en una unidad si el campo **last_used** lo permite. Por lo tanto para determinar cual es el bloque menos recientemente usado se compara linealmente el **lru_counter** de cada bloque del set y se selecciona el menor de ellos, esto quiere decir que al ser el menos recientemente utilizado sera el bloque a reemplazar.

- Las lecturas a memoria cache se realizan en la función **read_byte**, primero se descomponen los campos de tag y set del address y se realiza una búsqueda interna en cache para verficar si esta o no el bloque cargado.

  - En caso de que el dato este en cache simplemente se devuelve el dato como resultado.
  - Si el dato no esta en cache, automáticamente se pasa a cargar el bloque desde memoria utilizando la función **read_block** y luego se repite el llamado a la función **read_byte** para buscar nuevamente el dato que esta vez estará cargado en cache

- Las escrituras en memoria cache se realizan con la función **write_byte** y el algoritmo de escritura es similar al de lectura, primero se realiza una búsqueda en cache del dato descomponiendo la dirección de memoria en los campos de cache necesarios.

  - En caso de que el dato este en cache simplemente se escribe el dato en cache y se deja desactualizada la memoria.
  - Si el dato no esta en cache, automáticamente se pasa a cargar el bloque desde memoria utilizando la función **read_block** y luego se repite el llamado a la función **write_byte** para buscar nuevamente el dato que esta vez estará cargado en cache.

- Las lecturas de bloques de memoria principal a memoria cache se realizan con la función **read_block**, la cual se encarga de leer los datos necesarios accediendo a memoria, pero antes de que sean guardados en cache verifica que el bloque no este sucio, si esto es así primero tiene que escribir ese bloque de cache en memoria antes de que sea sobreescrito llamando a

la función **write_block**. Una vez que el bloque sea llevado a memoria para actualizarse en caso de que estuviera sucio entonces se procede a leer el bloque correspondiente de memoria y cargarlo en cache.

- Las escrituras de bloques de memoria cache a memoria principal se realizan con la función **write_block**, la cual únicamente se encarga de copiar el bloque entero a memoria luego de descomponer la dirección.
- El miss rate se calcula con la función **get_miss_rate** se encarga de calcular el miss rate de la memoria cache, para ello se cuenta con un campo interno en cache que cuenta la cantidad de misses y hits por accesos a cache (lecturas y escrituras), que son aumentados cada vez que se tiene que ir a buscar un dato a memoria o no.

4. Compilación y ejecución

4.1. Compilación

Para compilar el programa implementado se incluyo un archivo `Makefile` en el repositorio, se debe ejecutar el comando `make` [1], esto generará el archivo ejecutable `main`.

En caso que se requiera limpiar automáticamente el `build` realizado se puede ejecutar el comando `make clean`, y removerá todos los archivos generados por el comando `make`.

4.2. Ejecución

Para ejecutar el archivo compilado:

```
./main [Opciones] [archivo-de-entrada]
```

4.2.1. Descripción de parámetros

`-V` --version: Muestra la versión y sale del programa.

`-h` --help: Muestra información de ayuda de cómo ejecutar el programa y sale del programa.

`-o` --output: Path para el archivo de salida (opcionalmente, se puede obviar este parámetro e indicarlo como el último parámetro de la línea de comandos).

`-w` --ways: Cantidad de vías.

`-cs` --cachesize: Tamaño de la caché en kilobytes.

`-bs` --blocksize: Tamaño de bloque en bytes.

4.2.2. Ejemplos de ejecución

Para poder ejecutar el simulador, se deben pasar las características de la memoria caché (*ways*, *cachesize* y *blocksize*).

El siguiente ejemplo simulará la ejecución de las instrucciones que se encuentran en `prueba1.mem` sobre una caché de 4 vías, de tamaño de 8 KB y de tamaño de bloque 16 bytes.

```
./tp2 -w 4 -cs 8 -bs 16 prueba1.mem
```

Otra manera de ejecutar la misma línea, sin importar el orden de los parámetros, es mediante el comando:

```
./tp2 -w 4 -cs 8 -bs 16 -o prueba1.mem
```

5. Pruebas

5.1. Pruebas de la cátedra

La cátedra presenta dos casos de pruebas sobre dos configuraciones de memoria caché distintas:

- Caché 1:
  - 4KB tamaño de caché.
  - 4 vías.
  - 32 bytes tamaño del bloque.
- Caché 2:
  - 16KB tamaño de caché.
  - 1 vía.
  - 128 bytes tamaño del bloque.

Los dos casos de pruebas son sobre las siguentes secuencias:

- Secuencia 1 (`prueba1.mem`):

```
init
W 0, 255
W 16384, 254
W 32768, 248
W 49152, 096
R 0
R 16384
R 32768
R 49152
MR
```

- Secuencia 2 (`prueba2.mem`):

```
init
W 0, 123
W 1024, 234
W 2048, 33
W 3072, 44
W 4096, 55
R 0
R 1024
R 2048
R 3072
R 4096
MR
```

Las ejecuciones de estas simulaciones serán para la secuencia 1 (`prueba1.mem`):

- En la caché 1:
  - Comando: `./main -cs 4 -w 4 -bs 32 prueba1.mem`

- Salida:

```
Se inicia la caché
Escribe  en la dirección 0
Escribe  en la dirección 16384
Escribe  en la dirección 32768
Escribe ‘ en la dirección 49152
Leo de la dirección 0 y obtengo
Leo de la dirección 16384 y obtengo
Leo de la dirección 32768 y obtengo
Leo de la dirección 49152 y obtengo ‘
MR: %50
```

- En la caché 2:
  - Comando: “‘ ./main -cs 16 -w 1 -bs 128 prueba1.mem “‘
  - Salida:

```
Se inicia la caché
Escribe  en la dirección 0
Escribe  en la dirección 16384
Escribe  en la dirección 32768
Escribe ‘ en la dirección 49152
Leo de la dirección 0 y obtengo
Leo de la dirección 16384 y obtengo
Leo de la dirección 32768 y obtengo
Leo de la dirección 49152 y obtengo ‘
MR: %100
```

Las ejecuciones de estas simulaciones serán para la secuencia 2 (`prueba2.mem`):

- En la caché 1:
  - Comando: `./main -cs 4 -w 4 -bs 32 prueba2.mem`
  - Salida:

```
Se inicia la caché
Escribe { en la dirección 0
Escribe  en la dirección 1024
Escribe ! en la dirección 2048
Escribe , en la dirección 3072
Escribe 7 en la dirección 4096
Leo de la dirección 0 y obtengo {
Leo de la dirección 1024 y obtengo
Leo de la dirección 2048 y obtengo !
Leo de la dirección 3072 y obtengo ,
Leo de la dirección 4096 y obtengo 7
MR: %70
```

- En la caché 2:
  - Comando: `./main -cs 16 -w 1 -bs 128 prueba2.mem`
  - Salida:

```
Se inicia la caché
Escribe { en la dirección 0
Escribe  en la dirección 1024
Escribe ! en la dirección 2048
Escribe , en la dirección 3072
Escribe 7 en la dirección 4096
Leo de la dirección 0 y obtengo {
Leo de la dirección 1024 y obtengo
Leo de la dirección 2048 y obtengo !
Leo de la dirección 3072 y obtengo ,
Leo de la dirección 4096 y obtengo 7
MR: %50
```

5.2. Pruebas propias

Las pruebas propias se encuentran en la carpeta 'pruebas' y van desde la 'prueba3.mem' a la 'prueba6.mem'. Para ejecutar las pruebas se utilizó el siguiente comando:

$$\texttt{./main -cs 4 -w 4 -bs 32 pruebaX.mem}, \quad X = 3, 4, 5, 6$$

A continuación se muestra el contenido de cada prueba y su respectiva salida.

- Prueba 3:
  - Contenido:

```
init
W 0, 0
W 1, 65
R 0
R 1
W 32, 71
R 32
W 16, 90
W 0, 67
R 0
MR
```

  - Salida:

```
Se inicia la caché
Escribe  en la dirección 0
Escribe A en la dirección 1
Leo de la dirección 0 y obtengo
Leo de la dirección 1 y obtengo A
Escribe G en la dirección 32
Leo de la dirección 32 y obtengo G
Escribe Z en la dirección 16
Escribe C en la dirección 0
Leo de la dirección 0 y obtengo C
MR: %22
```

- Prueba 4:

- Contenido:

```
init
W 65535, 77
W 65534, 78
W 65533, 79
W 32768, 65
W 32768, 66
R 65535
R 65534
R 65533
R 32768
R 65535
MR
```

- Salida:

```
Se inicia la caché
Escribe M en la dirección 65535
Escribe N en la dirección 65534
Escribe O en la dirección 65533
Escribe A en la dirección 32768
Escribe B en la dirección 32768
Leo de la dirección 65535 y obtengo M
Leo de la dirección 65534 y obtengo N
Leo de la dirección 65533 y obtengo O
Leo de la dirección 32768 y obtengo B
Leo de la dirección 65535 y obtengo M
MR: %20
```

- Prueba 5:

  - Contenido:

```
init
R 0
W 0, 65
W 1, 66
W 2, 67
R 0
R 1
R 2
MR
```

  - Salida:

```
Se inicia la caché
Leo de la dirección 0 y obtengo
Escribe A en la dirección 0
Escribe B en la dirección 1
Escribe C en la dirección 2
Leo de la dirección 0 y obtengo A
Leo de la dirección 1 y obtengo B
Leo de la dirección 2 y obtengo C
MR: %14
```

- Prueba 6:
  - Contenido:

```
init
W 0, 65
W 1, 66
W 2, 67
W 3, 68
W 4, 69
W 5, 70
W 6, 71
W 7, 72
W 8, 73
W 9, 74
W 10, 75
W 11, 76
W 12, 77
W 13, 78
W 14, 79
W 15, 80
W 16, 81
W 17, 82
W 18, 83
W 19, 84
W 20, 85
W 21, 86
W 22, 87
W 23, 88
W 24, 89
W 25, 90
W 26, 91
R 0
R 1
R 2
R 3
R 4
R 5
R 6
R 7
R 8
R 9
R 10
R 11
R 12
R 13
R 14
R 15
R 16
R 17
R 18
R 19
R 20
```

```
R 21
R 22
R 23
R 24
R 25
R 26
MR
```

- Salida:

```
Se inicia la caché
Escribe A en la dirección 0
Escribe B en la dirección 1
Escribe C en la dirección 2
Escribe D en la dirección 3
Escribe E en la dirección 4
Escribe F en la dirección 5
Escribe G en la dirección 6
Escribe H en la dirección 7
Escribe I en la dirección 8
Escribe J en la dirección 9
Escribe K en la dirección 10
Escribe L en la dirección 11
Escribe M en la dirección 12
Escribe N en la dirección 13
Escribe O en la dirección 14
Escribe P en la dirección 15
Escribe Q en la dirección 16
Escribe R en la dirección 17
Escribe S en la dirección 18
Escribe T en la dirección 19
Escribe U en la dirección 20
Escribe V en la dirección 21
Escribe W en la dirección 22
Escribe X en la dirección 23
Escribe Y en la dirección 24
Escribe Z en la dirección 25
Escribe [ en la dirección 26
Leo de la dirección 0 y obtengo A
Leo de la dirección 1 y obtengo B
Leo de la dirección 2 y obtengo C
Leo de la dirección 3 y obtengo D
Leo de la dirección 4 y obtengo E
Leo de la dirección 5 y obtengo F
Leo de la dirección 6 y obtengo G
Leo de la dirección 7 y obtengo H
Leo de la dirección 8 y obtengo I
Leo de la dirección 9 y obtengo J
Leo de la dirección 10 y obtengo K
Leo de la dirección 11 y obtengo L
Leo de la dirección 12 y obtengo M
Leo de la dirección 13 y obtengo N
Leo de la dirección 14 y obtengo O
Leo de la dirección 15 y obtengo P
```

```
Leo de la dirección 16 y obtengo Q
Leo de la dirección 17 y obtengo R
Leo de la dirección 18 y obtengo S
Leo de la dirección 19 y obtengo T
Leo de la dirección 20 y obtengo U
Leo de la dirección 21 y obtengo V
Leo de la dirección 22 y obtengo W
Leo de la dirección 23 y obtengo X
Leo de la dirección 24 y obtengo Y
Leo de la dirección 25 y obtengo Z
Leo de la dirección 26 y obtengo [
MR: %1
```

6. Conclusiones

Se implemento una memoria cache LRU, es decir la política de reemplazo fue modificar el bloque de cache menos recientemente utilizado, la política de reemplazo es necesaria en caches que tengan algún grado de asociatividad pues el mapeo de un bloque de memoria a uno de cache no es uno a uno, esto lleva a la necesidad de una política de reemplazo. Adicionalmente cuando se escribe un dato en memoria se requiere evaluar una política de escritura, en este caso se implemento la política write back/write allocate, por lo cual ante un miss en escritura, el programa levanta un bloque de memoria principal y lo guarda en cache, para posteriormente ser escrito como si el cache siempre hubiese tenido ese dato, y ante un hit en escritura el programa escribe directamente sobre memoria cache y deja desactualizada la memoria principal, hasta que un bloque de memoria cache tenga que ser reemplazado, en cuyo caso el programa primero baja el bloque a memoria principal para actualizarlo y posteriormente escribe el nuevo bloque en cache.

Según las pruebas ejecutadas vemos la tasa de miss de la memoria cache depende de como esta este conformada, ya que en la prueba1.mem provista por la cátedra la tasa de miss de una cache grande con un menor grado de asociatividad y un tamaño de bloque mayor genera una tasa de miss de 100 %, y en la prueba2.mem se observa una tasa de miss de 50 % es decir menor a la prueba de cache para una memoria mas pequeña de menor grado de asociatividad y un tamaño de bloque menor.

A. Código

A.1. Código en C

A.1.1. main.c

```c
#define _POSIX_C_SOURCE 200809L
#include "address_parser.h"
#include "cache.h"
#include "parsers.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void test_01() {
  write_byte(0, (char)255);
  write_byte(16384, (char)254);
  write_byte(32768, (char)248);
  write_byte(49152, (char)96);

  read_byte(0);
  read_byte(16384);
  read_byte(32768);
  read_byte(49152);
}


void test_02() {
  write_byte(0, (char)123);
  write_byte(1024, (char)234);
  write_byte(2048, (char)33);
  write_byte(3072, (char)44);
  write_byte(4096, (char)55);

  read_byte(0);
  read_byte(1024);
  read_byte(2048);
  read_byte(3072);
  read_byte(4096);
}

void test_basic_small() { // Usar memoria de 64 bytes
  cache_init(32, 2, 4);
  printf("_____\n");
  // printf("%u\n", find_set(1));
  // printf("%u\n", find_set(4));
  // printf("%u\n", find_set(63));

  // printf("%d\n", find_lru(0));
  // printf("%u\n", is_dirty(1, 2));

  printf("_____\n");
  // set 0 via 0
  write_byte(0, 'd');
```

```
49  printf("MR: %%%d\n", get_miss_rate());
50
51  printf("char recuperado %c\n", read_byte(0));
52  printf("MR: %%%d\n", get_miss_rate());
53
54  // <>
55  // C digo para verificar si se actualiza la memoria cach  y la
        ↪ memoria queda
56  // desactualizada
57  printf("_____\n");
58  // set 0 via 0
59  write_byte(1, 'a');
60  printf("MR: %%%d\n", get_miss_rate());
61  //char buff[1];
62  //memory_read_byte(&memory, buff, 1);
63
64  printf("char recuperado %c\n", read_byte(1));
65  printf("MR: %%%d\n", get_miss_rate());
66
67  //printf("%d\n", buff[0]);
68  // </>
69
70  printf("_____\n");
71  // set 0 via 1
72  write_byte(32, 'q');
73  printf("MR: %%%d\n", get_miss_rate());
74
75  printf("char recuperado %c\n", read_byte(32));
76  printf("MR: %%%d\n", get_miss_rate());
77
78  printf("_____\n");
79
80  // set 0 via 1
81  write_byte(16, 'l');
82  printf("MR: %%%d\n", get_miss_rate());
83
84  printf("char recuperado %c\n", read_byte(16));
85  printf("MR: %%%d\n", get_miss_rate());
86
87  printf("_____\n");
88
89  // set 0 via 1
90
91  printf("char recuperado %c\n", read_byte(32));
92  printf("MR: %%%d\n", get_miss_rate());
93
94  // printf("_____\n");
95
96  // char buff[1];
97  // memory_read_byte(&memory, buff, 1);
98  // printf("char recuperado a == %c\n", buff[0]);
99
100 // printf("_____\n");
101 // // set 0 via 0
```

```
102   // write_byte(0, 'd');
103   // printf("MR: %%%d\n", get_miss_rate());
104
105   // printf("_____\n");
106   // printf("char recuperado %c\n", read_byte(16));
107   // printf("MR: %%%d\n", get_miss_rate());
108   // test_01();
109   cache_uninit();
110 }
111
112   // test_01();
113   // printf("MR: %%%d\n", get_miss_rate());
114   // printf("%u\n", address_parser_set(0, block_size, 128));
115   // printf("%u\n", address_parser_set(1024, block_size, 128));
116   // printf("%u\n", address_parser_set(2048, block_size, 128));
117   // printf("%u\n", address_parser_set(3072, block_size, 128));
118   // printf("%u\n", address_parser_set(4096, block_size, 128));
119
120 //  test_02();
121 //  printf("MR: %%%d\n", get_miss_rate());
122
123
124 int main(int argc, char *argv[]) {
125   argparser_t argparser;
126   argparser_init(&argparser);
127   argparser_parse(&argparser, argc, argv);
128   if (!argparser_is_command_valid(&argparser)) {
129     fprintf(stderr, "Error en los argumentos\n");
130     exit(EXIT_FAILURE);
131   }
132
133   char *output_file = argparser.output;
134   FILE *file = fopen(output_file, "r");
135   if (!file) {
136     fprintf(stderr, "No se pudo abrir el archivo %s\n", output_file);
137     exit(EXIT_FAILURE);
138   }
139   size_t num_bytes = 0;
140   char* line = NULL;
141
142   // unsigned int capacity = 4 * 1024;
143   // unsigned int n_ways = 4;
144   // unsigned int block_size = 32;
145
146   int capacity = argparser.capacity * 1024;
147   int n_ways = argparser.n_ways;
148   int block_size = argparser.block_size;
149
150   cache_init(capacity, n_ways, block_size);
151   command_t command;
152   command_init(&command);
153   while(getline(&line, &num_bytes, file) != -1) {
154     if (!command_parse_line(&command, line))
155       break;
```

```
156   }
157
158   free(line);
159   fclose(file);
160   cache_uninit();
161
162   return 0;
163 }
```

code/main.c

A.1.2. cache.c

```
1  #include "cache.h"
2  #include "memory.h"
3  #include "address_parser.h"
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  memory_t memory;
8  cache_t cache;
9
10 void _set_init(set_t *self, unsigned int ways_number, unsigned int
       ↪ block_data_size);
11 void _set_uninit(set_t *self);
12 void _block_init(block_t *self, unsigned int data_size);
13 void _block_uninit(block_t *self) ;
14 unsigned int _cache_total_sets();
15 unsigned int _cache_block_memory_address(unsigned int tag
16   , int setnum, unsigned int total_sets, unsigned int block_size);
17 int _cache_look_up(int tag, int set);
18
19 void cache_init(unsigned int capacity, unsigned int ways_number,
       ↪ unsigned int block_size) {
20   cache.capacity = capacity;
21   cache.ways_number = ways_number;
22   cache.block_size = block_size;
23   cache.hits = 0;
24   cache.misses = 0;
25   unsigned int total_sets = _cache_total_sets();
26   cache.sets = (set_t*)malloc(sizeof(set_t) * total_sets);
27   for (int set = 0; set < total_sets; set++) {
28     _set_init(&(cache.sets[set]), ways_number, block_size);
29   }
30 }
31
32 void update_last_used(int set, int way){
33   for (int _way = 0; _way < cache.ways_number; _way++){
34     cache.sets[set].blocks[_way].last_used = false;
35   }
36   cache.sets[set].blocks[way].last_used = true;
37 }
38
39 void init() {
40   memory_init(&memory);
```

```
41 }
42
43 void cache_uninit() {
44   unsigned int total_sets = _cache_total_sets();
45   for (int set = 0; set < total_sets; set++) {
46     _set_uninit(&(cache.sets[set]));
47   }
48   free(cache.sets);
49   memory_uninit(&memory);
50 }
51
52 unsigned int find_set(int address) {
53   return address_parser_set(address, cache.block_size,
       ↪ _cache_total_sets());
54 }
55
56 unsigned int find_lru(int setnum) {
57   set_t set = cache.sets[setnum];
58   int lru_way = 0;
59   int min_lru_counter = set.blocks[lru_way].lru_counter;
60
61   for (int way = 0; way < cache.ways_number; way++) {
62     // printf("%d\n", set.blocks[way].lru_counter);
63     if (set.blocks[way].lru_counter < min_lru_counter) {
64       lru_way = way;
65       min_lru_counter = set.blocks[lru_way].lru_counter;
66     }
67   }
68
69   // printf("via lru %d\n", lru_way);
70   return lru_way;
71 }
72
73 unsigned int is_dirty(int way, int setnum) {
74   set_t set = cache.sets[setnum];
75   block_t block = set.blocks[way];
76   return block.dirty;
77 }
78
79 void read_block(int blocknum) {
80   int address = blocknum * cache.block_size;
81   unsigned int tag = address_parser_tag(address
82     , cache.block_size, _cache_total_sets());
83   unsigned int set = address_parser_set(address
84     , cache.block_size, _cache_total_sets());
85   unsigned int way = find_lru(set);
86
87   if (is_dirty(way, set)) {
88     write_block(way, set);
89   }
90   cache.sets[set].blocks[way].valid = true;
91   cache.sets[set].blocks[way].dirty = false;
92   cache.sets[set].blocks[way].lru_counter = -1;
93   cache.sets[set].blocks[way].tag = tag;
```

```
94    char *data_buffer = cache.sets[set].blocks[way].data;
95    memory_read(&memory, data_buffer, address, cache.block_size);
96  }
97
98  void write_block(int way, int setnum) {
99    char *data = cache.sets[setnum].blocks[way].data;
100   unsigned int tag = cache.sets[setnum].blocks[way].tag;
101   unsigned int address = _cache_block_memory_address(tag, setnum
102     , _cache_total_sets(), cache.block_size);
103   memory_write(&memory, data, address, cache.block_size);
104 }
105
106 char read_byte(int address) {
107   unsigned int tag = address_parser_tag(address
108     , cache.block_size, _cache_total_sets());
109   unsigned int set = address_parser_set(address
110     , cache.block_size, _cache_total_sets());
111   unsigned int offset = address_parser_offset(address
112   , cache.block_size);
113   int way = _cache_look_up(tag, set);
114   char value = 0;
115   if (way >= 0) {
116     value = cache.sets[set].blocks[way].data[offset];
117     if (!cache.sets[set].blocks[way].last_used){
118       cache.sets[set].blocks[way].lru_counter++;
119     }
120     update_last_used(set, way);
121     // printf("Hit en lectura, add: %d \n", address);
122   } else {
123     // printf("Miss en lectura, add: %d \n", address);
124     read_block(address/cache.block_size);
125     value = read_byte(address);
126     cache.hits--;
127   }
128   return value;
129 }
130
131 void write_byte(int address, char value) {
132   unsigned int tag = address_parser_tag(address
133     , cache.block_size, _cache_total_sets());
134
135   unsigned int set = address_parser_set(address
136     , cache.block_size, _cache_total_sets());
137
138   unsigned int offset = address_parser_offset(address
139   , cache.block_size);
140
141   int way = _cache_look_up(tag, set);
142
143   if (way >= 0) { // HIT EN ESCRITURA ESCRIBO EN CACHE UNICAMENTE
144     cache.sets[set].blocks[way].data[offset] = value;
145     cache.sets[set].blocks[way].dirty = true;
146     if (!cache.sets[set].blocks[way].last_used){
147       cache.sets[set].blocks[way].lru_counter++;
```

```
148     }
149     update_last_used(set, way);
150     // printf("Hit en escritura, add: %d, val: %c \n", address,
            ↪ value);
151   } else { //  MISS EN ESCRITURA IMPLICA CARGAR EL DATO DE MEMORIA A
        ↪ CACHE Y ESCRIBIR EL DATO
152     read_block(address/cache.block_size);
153     // printf("Miss en escritura, add: %d, val: %c \n", address,
            ↪ value);
154     write_byte(address, value);
155     cache.hits--;
156   }
157 }
158
159 int get_miss_rate() {
160   float total = (float) (cache.misses + cache.hits);
161   return ((float) cache.misses / total) * 100;
162 }
163
164 int _cache_look_up(int tag, int set) {
165   int _way = -1;
166   for (int way = 0; way < cache.ways_number; way++) {
167     if (tag == cache.sets[set].blocks[way].tag
168         && cache.sets[set].blocks[way].valid) {
169       _way = way;
170       cache.hits++;
171       return _way;
172     }
173   }
174   cache.misses++;
175   return _way;
176 }
177
178 unsigned int _cache_block_memory_address(unsigned int tag
179   , int setnum, unsigned int total_sets, unsigned int block_size) {
180   unsigned int memory_tag_address = tag *
181       (total_sets * block_size);
182   unsigned int memory_set_address = setnum * block_size;
183   return memory_tag_address + memory_set_address;
184 }
185
186 unsigned int _cache_total_sets() {
187   return cache.capacity / (cache.block_size * cache.ways_number);
188 }
189
190 void _set_init(set_t *self, unsigned int ways_number, unsigned int
    ↪ block_data_size) {
191   self->blocks = (block_t *)malloc(sizeof(block_t) * ways_number);
192   for (int block = 0; block < ways_number; block++) {
193     _block_init(&(self->blocks[block]), block_data_size);
194   }
195 }
196
197 void _set_uninit(set_t *self) {
```

```
198   for (int block = 0; block < cache.ways_number; block++){
199     _block_uninit(&(self->blocks[block]));
200   }
201   free(self->blocks);
202 }
203
204 void _block_init(block_t *self, unsigned int data_size) {
205   self->data = (char *)malloc(sizeof(char) * data_size);
206   self->tag = 0;
207   self->dirty = false;
208   self->valid = false;
209   self->lru_counter = -1;
210   self->last_used = false;
211 }
212
213 void _block_uninit(block_t *self) {
214   free(self->data);
215   self->tag = 0;
216   self->dirty = false;
217   self->valid = false;
218 }
```

code/cache.c

A.1.3. cache.h

```
1  #ifndef _CACHE_H_
2  #define _CACHE_H_
3  #include <stdbool.h>
4  #include "memory.h"
5
6  typedef struct set set_t;
7  typedef struct block block_t;
8
9  typedef struct cache {
10   unsigned int capacity;
11   unsigned int ways_number;
12   unsigned int block_size;
13   set_t *sets;
14   unsigned int misses;
15   unsigned int hits;
16 } cache_t;
17
18 struct set {
19   block_t *blocks;
20 };
21
22 struct block {
23   char *data;
24   int lru_counter;
25   unsigned int tag;
26   bool dirty;
27   bool valid;
28   bool last_used;
29 };
```

```
30
31 void cache_init(unsigned int capacity, unsigned int ways_number,
       ↪ unsigned int block_size);
32 void cache_uninit();
33 void init();
34 unsigned int find_set(int address);
35 unsigned int find_lru(int setnum);
36 unsigned int is_dirty(int way, int setnum);
37 void read_block(int blocknum);
38 void write_block(int way, int setnum);
39 char read_byte(int address);
40 void write_byte(int address, char value);
41 int get_miss_rate();
42
43 #endif
```

code/cache.h

A.1.4. memory.c

```
1  #include "memory.h"
2  #include <string.h>
3  #include <stdio.h>
4
5  void memory_init(memory_t *self) {
6    memset(self->data, 0, sizeof(self->data));
7  }
8
9  void memory_uninit(memory_t *self) {
10
11 }
12
13 void memory_write_byte(memory_t *self, char data,
14     int address) {
15   self->data[address] = data;
16 }
17
18 void memory_read_byte(memory_t *self, char *data_buffer,
19     int address) {
20   *data_buffer = self->data[address];
21 }
22
23 void memory_write(memory_t *self, char *data
24     , int address, unsigned int data_size) {
25   for(int index = 0; index < data_size; index++) {
26     memory_write_byte(self, data[index], address++);
27   }
28 }
29
30 void memory_read(memory_t *self, char *data_buffer
31     , int address, unsigned int data_size) {
32   for(int index = 0; index < data_size; index++) {
33     memory_read_byte(self, (data_buffer + index), address++);
34   }
```

```
35 }
```

code/memory.c

A.1.5. memory.h

```
1  #ifndef _MEMORY_H_
2  #define _MEMORY_H_
3
4  #define CAPACITY 64 * 1024 //En bytes
5
6  typedef struct memory {
7    char data[CAPACITY];
8  } memory_t;
9
10 void memory_init(memory_t *self);
11 void memory_uninit(memory_t *self);
12 void memory_write_byte(memory_t *self
13      , char data, int byte_address);
14 void memory_read_byte(memory_t *self
15      , char *data_buffer, int byte_address);
16 void memory_write(memory_t *self, char *data
17      , int address, unsigned int data_size);
18 void memory_read(memory_t *self, char *data_buffer
19      , int address, unsigned int data_size);
20
21 #endif
```

code/memory.h

A.1.6. address$_p$arser.c

```
1  #include "address_parser.h"
2
3  unsigned int address_parser_tag(unsigned int address
4    , unsigned int block_size, unsigned int total_sets) {
5    return address / (block_size * total_sets);
6  }
7
8  unsigned int address_parser_set(unsigned int address
9    , unsigned int block_size, unsigned int total_sets) {
10   return (address / block_size) % total_sets;
11 }
12
13 unsigned int address_parser_offset(unsigned int address
14   , unsigned int block_size) {
15   return address % block_size;
16 }
```

code/address_parser.c

A.1.7. address$_p$arser.h

```
1  #ifndef _ADDRESS_PARSER_H_
2  #define _ADDRESS_PARSER_H_
3
4  unsigned int address_parser_tag(unsigned int address
```

```
5   , unsigned int block_size, unsigned int total_sets);
6 unsigned int address_parser_set(unsigned int address
7   , unsigned int block_size, unsigned int total_sets);
8 unsigned int address_parser_offset(unsigned int address
9   , unsigned int block_size);
10
11 #endif
```

code/address_parser.h

A.1.8. parsers.c

```
1 #include "parsers.h"
2 #include "strutil.h"
3 #include "cache.h"
4 #include <ctype.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7 #include <stdio.h>
8 #include <string.h>
9
10
11 #define COMMAND_DELIMITER ' '
12
13 static bool init_wrapper(char *args[]);
14 static bool write_wrapper(char *args[]);
15 static bool read_wrapper(char *args[]);
16 static bool get_miss_rate_wrapper(char *args[]);
17 static bool command_not_found(char *args[]);
18 static int vector_look_up(char *vector[], char *search_value, int
    ↪ vector_len);
19 static void show_usage();
20 static void show_version();
21 static bool is_arg_equal_short_long(char *arg, char *short_form,
22                   char *long_form);
23 static char* safe_get(char *vector[], int index, int vector_len,
24             char *long_form);
25 static bool is_a_number(char* num);
26 static int safe_get_int(char *vector[], int index, int vector_len,
27             int long_form);
28
29 // ---------- CommandParser ----------
30
31 void command_init(command_t *self) {
32   self->command_names[0] = "init";
33   self->command_names[1] = "R";
34   self->command_names[2] = "W";
35   self->command_names[3] = "MR";
36
37   self->commands[0] = init_wrapper;
38   self->commands[1] = read_wrapper;
39   self->commands[2] = write_wrapper;
40   self->commands[3] = get_miss_rate_wrapper;
41 }
42
```

```
43 bool command_parse_line(command_t *self, char *line) {
44   char **splitted_line = split(line, COMMAND_DELIMITER);
45   char *str_command = splitted_line[0];
46   int i_command = vector_look_up(self->command_names, str_command,
47                       AVAILABLE_OPTIONS);
48   if (i_command < 0)
49     self->selected_command = command_not_found;
50   else
51     self->selected_command = self->commands[i_command];
52   bool result = self->selected_command(splitted_line);
53   free_strv(splitted_line);
54   return result;
55 }
56
57 static bool init_wrapper(char *args[]) {
58   init();
59   fprintf(stdout, "Se inicia la cach \n");
60   return true;
61 }
62
63 static bool write_wrapper(char *args[]) {
64   if (!args[1] || !args[2]) return false;
65   int mem_direction = atoi(args[1]);
66   char data = (char)atoi(args[2]);
67   write_byte(mem_direction, data);
68   fprintf(stdout, "Escribe %c en la direcci n %d\n", data,
       ↪ mem_direction);
69   return true;
70 }
71
72 static bool read_wrapper(char *args[]) {
73   if (!args[1]) return false;
74   int mem_direction = atoi(args[1]);
75   char resultado = read_byte(mem_direction);
76   fprintf(stdout, "Leo de la direcci n %d y obtengo %c\n",
       ↪ mem_direction,
77        resultado);
78   return true;
79 }
80
81 static bool get_miss_rate_wrapper(char *args[]) {
82   fprintf(stdout, "MR: %%%d%%n" , get_miss_rate());
83   return true;
84 }
85
86 static bool command_not_found(char *args[]) {
87   fprintf(stderr, "Archivo con formato inv lido.\n");
88   return false;
89 }
90
91
92 static int vector_look_up(char *vector[], char *search_value, int
     ↪ vector_len) {
93   for (int i = 0; i < vector_len; i++) {
```

```
94      if (strncmp(search_value, vector[i],
95              strlen(vector[i])) == 0) {
96        return i;
97      }
98    }
99    return -1;
100 }
101
102 // ------------------ ArgParser ------------------
103
104 void argparser_init(argparser_t *self) {
105   self->capacity = -1;
106   self->n_ways = -1;
107   self->block_size = -1;
108   self->output = NULL;
109 }
110
111 void argparser_parse(argparser_t *self, int argc, char *argv[]) {
112   int last_index = 0;
113   for (int i = 1; i < argc; i++) {
114     char *arg = argv[i];
115     if (is_arg_equal_short_long(arg, "-h", "--help")) {
116       show_usage();
117       exit(EXIT_SUCCESS);
118     } else if (is_arg_equal_short_long(arg, "-V", "--version")) {
119       show_version();
120       exit(EXIT_SUCCESS);
121     } else if (is_arg_equal_short_long(arg, "-o", "--output")) {
122       self->output = safe_get(argv, ++i, argc, "");
123       last_index = i;
124     } else if (is_arg_equal_short_long(arg, "-w", "--ways")) {
125       self->n_ways = safe_get_int(argv, ++i, argc, -1);
126       last_index = i;
127     } else if (is_arg_equal_short_long(arg, "-cs", "--cachesize")) {
128       self->capacity = safe_get_int(argv, ++i, argc, -1);
129       last_index = i;
130     } else if (is_arg_equal_short_long(arg, "-bs", "--blocksize")) {
131       self->block_size = safe_get_int(argv, ++i, argc, -1);
132       last_index = i;
133     }
134   }
135   if (!self->output && last_index < argc - 1) {
136     self->output = argv[++last_index];
137   }
138 }
139
140 bool argparser_is_command_valid(argparser_t *self) {
141   return (self->output != NULL) && (self->n_ways > 0) &&
142        (self->capacity > 0) && (self->block_size > 0);
143 }
144
145 static void show_usage() {
146   fprintf(stdout,
147       "Usage:\n"
```

```
148         " tp2 -h\n"
149         " tp2 -V\n"
150         " tp2 options archivo\n"
151         "Options:\n"
152         " -h, --help      Imprime ayuda.\n"
153         " -V, --version     Versi n del programa.\n"
154         " -o, --output    Archivo de salida.\n"
155         " -w, --ways      Cantidad de v as.\n"
156         " -cs --cachesize   Tama o del cach  en kilobytes.\n"
157         " -bs, --blocksize  Tama o del bloque en bytes.\n"
158         "Examples:\n"
159         " tp2 -w 4 -cs 8 -bs 16 prueba1.mem\n");
160 }
161
162 static void show_version() {
163   fprintf(stdout, "Version 1.0\n");
164 }
165
166
167 static bool is_arg_equal_short_long(char *arg, char *short_form,
168                   char *long_form) {
169   return strcmp(arg, short_form) == 0 || strcmp(arg, long_form) == 0;
170 }
171
172 static char* safe_get(char *vector[], int index, int vector_len,
173             char *default_value) {
174   if (index < 0 || index >= vector_len)
175     return default_value;
176   return vector[index];
177 }
178
179 static bool is_a_number(char* num) {
180   if (num[0] != '-' && !isdigit(num[0]))
181     return false;
182
183   size_t len_number = strlen(num);
184   for (size_t i = 1; i < len_number; i++){
185     if (!isdigit(num[i]))
186       return false;
187   }
188   return true;
189 }
190
191 static int safe_get_int(char *vector[], int index, int vector_len,
192             int default_value) {
193   if (index < 0 || index >= vector_len || !is_a_number(vector[index]))
194     return default_value;
195   return atoi(vector[index]);
196 }
```

code/parsers.c

A.1.9. parsers.h

```
1 #ifndef __PARSERS_H__
```

```h
#define __PARSERS_H__

#define AVAILABLE_OPTIONS 4
#include <stdbool.h>

typedef bool (*command_f_t)(char*[]);

typedef struct {
  command_f_t selected_command;
  char *command_names[AVAILABLE_OPTIONS];
  command_f_t commands[AVAILABLE_OPTIONS];
} command_t;


void command_init(command_t *self);

bool command_parse_line(command_t *self, char *line);

typedef struct {
  int capacity;
  int n_ways;
  int block_size;
  char *output;
} argparser_t;

void argparser_init(argparser_t *self);

void argparser_parse(argparser_t *self, int argc, char *argv[]);

bool argparser_is_command_valid(argparser_t *self);



#endif
```

code/parsers.h

A.1.10. strutil.c

```c
/*
Alumno: Torresetti Lisandro
Padron: 99846
*/
#define _POSIX_C_SOURCE 200809L  //Para que ande strdup
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "strutil.h"

size_t obtener_longitud(char** str_vector);
size_t contar_separadores(const char* cadena, char sep, size_t*
    ↪ larger_str);

char *substr(const char *str, size_t n){
  size_t long_ideal = strlen(str);
```

```
16    if (long_ideal >= n) long_ideal = n;
17
18    char *cadena_resultante = malloc(sizeof(char) * (long_ideal + 1));
          ↪ //para poner barra cero sumas 1
19    if(!cadena_resultante) return NULL;
20
21    strncpy(cadena_resultante, str, long_ideal); //copia n - 1
          ↪ caracteres
22    cadena_resultante[long_ideal] = '\0';
23    return cadena_resultante;
24  }
25
26  char **split(const char *str, char sep){
27    size_t long_str = strlen(str) + 1;
28    size_t cad_mas_larga = 0;
29    size_t long_ideal = contar_separadores(str, sep, &cad_mas_larga);
30
31    //Creo el vector
32    char** str_vector = malloc(sizeof(char*) * long_ideal);
33    if (!str_vector) return NULL;
34
35    char cadena_aux[cad_mas_larga + 1]; //memoria estatica
36
37    size_t pos_vector = 0, pos_cad_aux = 0;
38    for (size_t i = 0; i < long_str; ++i){
39      if(str[i] == sep || i + 1 == long_str){
40        cadena_aux[pos_cad_aux] = '\0';
41        str_vector[pos_vector++] = strdup(cadena_aux);
42        pos_cad_aux = 0;
43        continue;
44      }
45      cadena_aux[pos_cad_aux++] = str[i];
46    }
47    str_vector[pos_vector] = NULL;
48    return str_vector;
49  }
50
51  char *join(char **strv, char sep){
52    size_t long_requerida = obtener_longitud(strv);
53    char* cadena_aux = malloc(sizeof(char) * long_requerida);
54    if (!cadena_aux) return NULL;
55
56    char *cadena = NULL;
57    size_t pos_cad_aux = 0, pos_vector = 0, long_cadena = 0;
58
59    while((cadena = strv[pos_vector]) != NULL){
60      long_cadena  = strlen(cadena);
61      for (size_t i = 0; i < long_cadena; i++)
          ↪ cadena_aux[pos_cad_aux++] = cadena[i];
62      if (strv[++pos_vector] != NULL && sep != '\0')
          ↪ cadena_aux[pos_cad_aux++] = sep;
63      //La cadena que concatene no era la ultima
64    }
65    cadena_aux[pos_cad_aux] = '\0';
```

```
66    return cadena_aux;
67  }
68
69  void free_strv(char *strv[]){
70    size_t contador = 0;
71    while(strv[contador] != NULL) free(strv[contador++]); //libero las
        ↪ cadenas
72    free(strv); //libero el vector
73  }
74
75  //Funciones auxiliares
76  size_t contar_separadores(const char* cadena, char sep, size_t*
      ↪ mayor_long){
77    /*Cuenta la cantidad de separadores que tiene la cadena, a su vez
        ↪ calcula
78    la longitud de la maxima cadena posible formada*/
79    size_t cant_sep = 2; //porque si pasan cadena vacia necesito uno
        ↪ para el \0 y otro para el NULL
80    size_t long_max_aux = 0, long_cadena = strlen(cadena);
81
82    for (size_t i = 0; i < long_cadena; i++){
83      long_max_aux++;
84      if(cadena[i] == sep){
85        *mayor_long = (*mayor_long < long_max_aux) ? long_max_aux :
            ↪ *mayor_long;
86        cant_sep++;
87        long_max_aux = 0;
88      }
89      else if (i + 1 == long_cadena) *mayor_long = (*mayor_long <
          ↪ long_max_aux) ? long_max_aux : *mayor_long;
90    }
91    return cant_sep;
92  }
93
94  size_t obtener_longitud(char** vec_str){
95    size_t long_total= 1; //Aunque sea tiene que ser uno para guardar
        ↪ el \0
96    size_t pos_vector = 0;
97    while(vec_str[pos_vector] != NULL) long_total +=
        ↪ strlen(vec_str[pos_vector++]) + 1;
98    return long_total;
99  }
```

code/strutil.c

A.1.11. strutil.h

```
1  #ifndef STRUTIL_H
2  #define STRUTIL_H
3
4  #include <stddef.h>
5
6  /*
7   * Devuelve una nueva cadena con los primeros    n    caracteres de
      ↪ la cadena
```

```
8  *     str    . La liberaci n de la memoria din mica devuelta queda a
       ↪ cargo de
9  * quien llame a esta funci n.
10 *
11 * Devuelve NULL si no se pudo reservar suficiente memoria.
12 */
13 char *substr(const char *str, size_t n);
14
15 /*
16  * Devuelve en un arreglo din mico terminado en NULL todos los
       ↪ subsegmentos de
17 *     str     separados por el car cter     sep    . Tanto el arreglo
       ↪ devuelto como las
18 * cadenas que contiene se ubicar n en nuevos espacios de memoria
       ↪ din mica.
19 *
20 * La liberaci n de la memoria din mica devuelta queda a cargo de
       ↪ quien llame a
21 * esta funci n. La funci n devuelve NULL en caso de error.
22 */
23 char **split(const char *str, char sep);
24
25 /*
26  * Devuelve la cadena resultante de unir todas las cadenas del arreglo
27 * terminado en NULL     str     con     sep     entre cadenas. La cadena
       ↪ devuelta se
28 * ubicar  en un nuevo espacio de memoria din mica.
29 *
30 * La liberaci n de la memoria din mica devuelta queda a cargo de
       ↪ quien llame a
31 * esta funci n. La funci n devuelve NULL en caso de error.
32 */
33 char *join(char **strv, char sep);
34
35 /*
36  * Libera un arreglo din mico de cadenas, y todas las cadenas que
       ↪ contiene.
37 */
38 void free_strv(char *strv[]);
39
40 #endif  // STRUTIL_H
```

code/strutil.h

B. Enunciado del trabajo práctico

66:20 Organización de Computadoras
Trabajo práctico 2: Memorias caché

1. Objetivos

Familiarizarse con el funcionamiento de la memoria caché implementando una simulación de una caché dada.

2. Alcance

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

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 8), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido. Por este motivo, el día de la entrega deben concurrir todos los integrantes del grupo.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo, y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Este trabajo práctico debe ser implementado en C, y correr al menos en Linux.

5. Introducción

La memoria a simular es una caché [1] asociativa por conjuntos, en que se puedan pasar por parámetro el número de vías, la capacidad y el tamaño de bloque. La política de reemplazo será LRU y la política de escritura

será WB/WA. Se asume que el espacio de direcciones es de 16 bits, y hay entonces una memoria principal a simular con un tamaño de 64KB. Estas memorias pueden ser implementadas como variables globales. Cada bloque de la memoria caché deberá contar con su metadata, incluyendo el bit V, el bit D, el *tag*, y un campo que permita implementar la política de LRU.

6. Programa

6.1. Comportamiento deseado

Primero, usamos la opción `-h` para ver el mensaje de ayuda:

```
$ tp2 -h
Usage:
  tp2 -h
  tp2 -V
  tp2 options archivo
Options:
  -h,  --help       Imprime ayuda.
  -V,  --version    Versión del programa.
  -o,  --output     Archivo de salida.
  -w,  --ways       Cantidad de vías.
  -cs, --cachesize  Tamaño del caché en kilobytes.
  -bs, --blocksize  Tamaño de bloque en bytes.
Examples:
  tp2 -w 4 -cs 8 -bs 16 prueba1.mem
```

El tamaño del caché es en total, sin contar metadatos: si tiene 4 KB de tamaño , 4 vías y bloques de 256 bytes, tendrá 4 bloques por vía. Si falta alguno de los parámetros del cache o el archivo de entrada no es suministrado, el programa debe reportarlo como un error.

6.2. Primitivas de caché

Se deben implementar las siguientes primitivas:

```
void init()
unsigned int find_set(int address)
unsigned int find_lru(int setnum)
unsigned int is_dirty(int way, int setnum)
void read_block(int blocknum)
void write_block(int way, int setnum)
char read_byte(int address)
void write_byte(int address, char value)
int get_miss_rate()
```

- La función `init()` debe inicializar los bloques de la caché como inválidos, la memoria simulada en 0 y la tasa de misses a 0.

- La función `find_set(int address)` debe devolver el conjunto de caché al que mapea la dirección `address`.
- La función `find_lru(int setnum)` debe devolver el bloque menos recientemente usado dentro de un conjunto (o alguno de ellos si hay más de uno), utilizando el campo correspondiente de los metadatos de los bloques del conjunto.
- La función `is_dirty(int way, int blocknum)` debe devolver el estado del bit D del bloque correspondiente.
- La función `read_block(int blocknum)` debe leer el bloque `blocknum` de memoria y guardarlo en el lugar que le corresponda en la memoria caché.
- La función `write_block(int way, int setnum)` debe escribir en memoria los datos contenidos en el bloque `setnum` de la vía `way`.
- La función `read_byte(address)` debe retornar el valor correspondiente a la posición de memoria `address`, buscándolo primero en el caché.
- La función `write_byte(int address, char value)` debe escribir el valor `value` en la posición correcta del bloque que corresponde a `address`.
- La función `get_miss_rate()` debe devolver el porcentaje de misses desde que se inicializó el cache.
- `read_byte()` y `write_byte()` sólo deben interactuar con la memoria a través de las otras primitivas.

Con estas primitivas (más las funciones que hagan falta para manejar LRU y WB/WA), hacer un programa que lea de un archivo una serie de comandos, que tendrán la siguiente forma:

```
init
R ddddd
W ddddd, vvv
MR
```

- Los comandos de la forma "init" se ejecutan llamando a la función `init` para inicializar la caché y la memoria simulada.
- Los comandos de la forma "`R ddddd`" se ejecutan llamando a la función `read_byte(ddddd)` e imprimiendo el resultado y si es hit o miss.
- Los comandos de la forma "`W ddddd, vvv`" se ejecutan llamando a la función `write_byte(int ddddd, char vvv)` e imprimiendo si es hit o miss.

- Los comandos de la forma "`MR`" se ejecutan llamando a la función `get_miss_rate()` e imprimiendo el resultado.

El programa deberá chequear que los valores de los argumentos a los comandos estén dentro del rango de direcciones y valores antes de llamar a las funciones, e imprimir un mensaje de error informativo cuando corresponda.

7. Pruebas

Se deberá incluir la salida que produzca el programa con los siguientes archivos de prueba, para las siguientes cachés: [4 KB, 4WSA, 32bytes] y [16KB, una vía, 128 bytes].

- prueba1.mem
- prueba2.mem

8. Informe

El informe deberá incluir:

- Este enunciado;
- Una descripción detallada de las decisiones de diseño y el comportamiento deseado para el caché.
- El código fuente completo del programa.

9. Fecha de entrega

La fecha de entrega y presentación es el jueves 17 de Diciembre de 2020.

Referencias

[1] Hennessy, John L. and Patterson, David A., Computer Architecture: A Quantitative Approach, Third Edition, 2002.

C. Código auxiliar

Referencias

[1] *Herramienta* `make`. https://www.gnu.org/software/make/manual/make.html