# Trabajo Práctico 2:
# Memorias caché

Ronnie Del Pino Cárdenas, *Padrón 93575*
delpinor@gmail.com
@delpinor

Emiliano Vega, *Padrón 76676*
emiliano.vega@mail.com
@emiliano.vega

Romina Casal, *Padrón 86429*
casal.romina@gmail.com
@romina

2do. Cuatrimestre de 2019
86.37 / 66.20 Organización de Computadoras − Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

**Resumen**

Por medio del presente trabajo práctico se implementó una simulación de una memoria caché de características específicas para analizar su comportamiento con ciertos programas de prueba para medir su performance.

## 1. Introducción

Usando el programa implementado, se realizará la interpretación de varios programas que ejecutarán comandos definidos para este trabajo simulando accesos a la memoria para analizar el uso de la memoria caché simulada y observar su desempeño.

## 2. Proceso de Compilación

Para la compilación del programa se definió un makefile que ejecuta gcc con opciones convencionales sobre el fuente **main.c**

Creamos un archivo script pruebas.sh que escribirá los resultados en un archivo de salida **test_result.txt**

## 3. Desarrollo

Para la realización del trabajo práctico se solicitó simular un espacio de memoria principal de 64KB (array **mem**) y un caché de 16KB en 8 vias y bloques de 64 bytes (array **cache**). Al ser

un espacio pequeño de memoria se ha definido ambas estructuras como un array estático. **cache** además de su espacio de memoria guarda metadatos de control. Por cada conjunto guarda cuál es el próximo conjunto a reemplazar por política FIFO, y por cada vía guarda también un flag de valid, dirty, el tag de la dirección del bloque, y la propia dirección del bloque en la ram, además de un bloque de 64 bytes para los datos. Si bien el array **cache** se ha definido como una estructura de words (int) para convivir en la misma los metadatos y el espacio de memoria, se ha controlado que se esté accediendo a nivel byte en las funciones de acceso a memoria implementadas. Para el tipo de simulación que se desea evaluar no impacta en los resultados ya que no estamos midiendo tiempo sino tasa de éxito del caché.

Para la realización de las pruebas se ha solicitado implementar las siguientes funciones en un programa en lenguaje C:

**void init()**: Usamos memset() para inicializar a 0 los espacios de memoria que simulan la principal y la caché.

**unsigned int get_offset (unsigned int address)**: devuelve el resto entre la dirección indicada y el blocksize como offset del bloque.

**unsigned int find_set(unsigned int address)**: busca el conjunto de acuerdo a la dirección de memoria solicitada.

**unsigned int select_oldest(unsigned int setnum)**: obtiene cuál será el próximo bloque a reemplazar de ese conjunto.

**void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)** : transfiere un bloque de memoria RAM al caché

**void write_tomem(unsigned int address, unsigned char)** : transfiere un bloque de memoria caché a la memoria RAM

**unsigned char read_byte(unsigned int address)** : realiza el proceso de acceso a un dato en memoria buscando previamente si existe el dato en el caché y si no existe guarda el bloque en el caché.

**void write_byte(unsigned int address, unsigned char value)** : si encuentra el bloque en el caché, guarda el valor en el caché. Si no lo encuentra, guarda el valor en memoria y caché.

**float get_miss_rate()** : calcula la cantidad de misses realizados durante la ejecución del programa.

# 4. Resultados obtenidos

Ejecutamos el script **pruebas.sh** y arrojó los siguientes resultados que exporta al archivo **test_result.txt**:

```
Pruebas TP2
prueba1.mem
Resultado: 255
Resultado: 48
Resultado: 96
Resultado: 192
Resultado: 84
Resultado: 68
Resultado: 88
Resultado: 80
Resultado: 255
MR: 55.56%

prueba2.mem
Resultado: 0
Resultado: 0
Resultado: 10
Resultado: 20
MR: 33.33%

prueba3.mem
Resultado: 255
```

```
Resultado: 2
Resultado: 3
Resultado: 4
Resultado: 5
Resultado: 0
Resultado: 0
Resultado: 255
Resultado: 2
Resultado: 3
Resultado: 4
Resultado: 5
MR: 17.65%

prueba3b.mem
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 1
Resultado: 2
Resultado: 3
Resultado: 4
MR: 62.50%

prueba4.mem
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 1
Resultado: 2
Resultado: 3
Resultado: 4
MR: 62.50%

prueba5.mem
Dirección no válida: 131072
Resultado: 0
Dirección o valores no válidos: W 16384,  256

Resultado: 0
Resultado: 0
Resultado: 0
Resultado: 0
MR: 60.00%
```

En el programa prueba1.mem observamos que casi todos los misses ocurridos son compulsivos, ya que como partimos de una caché inicializada, los bloques accedidos en la primera parte del programa aún no se encontraban en la caché y están accediendo a distintos conjuntos de la misma, separados por bloques mayores a 64 bytes.
En el programa prueba2.mem la tasa de misses es también dada por misses compulsivos(2 misses en total) pero al acceder a menos direcciones que ya se encuentran en caché con más frecuencia, la tasa de fallos resultó menor. En el programa prueba3.mem ocurre algo similar a prueba2.mem, con la diferencia de que no se trata de las mismas direcciones de memoria, pero sí están en el mismo bloque ya cargado en el caché, es decir se acceden a posiciones contiguas de memoria del primer bloque cargado. Para el caso 3b, si bien se accedes a posiciones contiguas de memoria, los datos no están en el mismo bloque por lo que se produces fallos en mayor número. Prueba4, Idem 3b. En el programa prueba5.mem tenemos dos comandos que no se pueden procesar. En el primer caso la dirección de memoria es mayor a la máxima(65535) y en el segundo caso se intenta escribir el valor 256 que es mayor a 8 bits(1 byte).

# 5. Código fuente

**Repositorio:github.com/romicasal/orga6620/tree/tp1_delpinor/tp2**

## 5.1. pruebas.sh

```
echo "Pruebas TP2" > test_result.txt
echo "prueba1.mem" >> test_result.txt
./tp2 prueba1.mem >> test_result.txt
echo "" >> test_result.txt
echo "prueba2.mem" >> test_result.txt
./tp2 prueba2.mem >> test_result.txt
echo "" >> test_result.txt
echo "prueba3.mem" >> test_result.txt
./tp2 prueba3.mem >> test_result.txt
echo "" >> test_result.txt
echo "prueba4.mem" >> test_result.txt
./tp2 prueba4.mem >> test_result.txt
echo "" >> test_result.txt
echo "prueba5.mem" >> test_result.txt
./tp2 prueba5.mem >> test_result.txt
```

## 5.2. main.c

```
/*
 * FILAS:
 * Cantidad de conjuntos = Bloques de cache / vias
 * Cantidad de conjuntos = 256 / 8 = 32
 * COLUMNAS:
 * FIFO = 1
 * Valid = 8x1
 * Dirty bit = 8x1
 * TAG =  8x1
 * Nro. de Block = 8x1
 * Datos = 64 * 8 = 512
 * Total = 1 + 8 + 8 + 8 + 8 + 512 = 545
 * |F|...|V|D|T|B|DA|...
 * |1|...|1|1|1|1|64|...  |1|1|1|1|64|....
 *  */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>

#define CONJUNTOS 32
#define FIFOINDEX 0
#define VALIDDIRTYTAG 4
#define ANCHO 545
#define BLOCKSIZE 64
#define MMSIZE 65536
#define OFFSET 6
#define VIAS 8
#define SET 5
#define TAG 5
#define BITS 16
#define FILENAME "prueba3b.mem"
float missRate = 0;
int accesosMemoria = 0;
float misses = 0;
float hits = 0;
unsigned int cache[CONJUNTOS][ANCHO];
unsigned int ram[MMSIZE];
```

```
void init();
unsigned int get_offset(unsigned int address);
unsigned int get_tag(unsigned int address);
unsigned int find_set(unsigned int address);
unsigned int select_oldest(unsigned int setnum);
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set);
unsigned int find_blockIndex(unsigned int way);
void write_tomem(unsigned int blocknum, unsigned int way, unsigned int set);
unsigned char read_byte(unsigned int address);
void write_byte(unsigned int address, unsigned char value);
float get_miss_rate();
unsigned int block_address(unsigned int address);
int main(void) {

    char *line_buf = NULL;
    char bloque1[7];
    char bloque2[7];
    size_t line_buf_size = 0;
    int line_count = 0;
    ssize_t line_size;

    if (!(argc > 1)) {
      fprintf(stderr, "Program filename argument missing\n");
      return EXIT_FAILURE;
    }

    FILE *fp = fopen(argv[1], "r");
    if (!fp) {
        fprintf(stderr, "Error opening file '%s'\n", argv[1]);
        return EXIT_FAILURE;
    }

    line_size = getline(&line_buf, &line_buf_size, fp);

    init();

    while (line_size >= 0) {
        line_count++;
        switch(line_buf[0]) {
        case 'F':
            //printf("FLUSH \n");
            init();
            break;
        case 'W':
            memcpy(bloque1,&line_buf[2], 7);
            int w;
            sscanf(bloque1, "%d", &w);

            char* token;
            char* rest = line_buf;
            int contar = 0;
            while ((token = strtok_r(rest, ",", &rest))) {
                if(contar == 1) {
                    memcpy(bloque2,token, 7);
                }
                contar++;
            }
            int v;
            sscanf(bloque2, "%d", &v );
            if(w >= MMSIZE || v > 255)
                printf("Direccion o valores no validos: W %s \n", bloque1);
            else
                write_byte(w,v);
            break;
        case 'R':
```

```
            memcpy(bloque1,&line_buf[2], 7);
            int r;
            sscanf(bloque1, "%d", &r);

            if(r > MMSIZE)
                printf("Direccion no valida: %d\n", r);
            else
                printf("Resultado: %d \n",read_byte(r));
            break;
        case 'M':
            printf("MR: %0.2f %%\n", get_miss_rate()*100);
            break;
        default:
            break;
        }


        line_size = getline(&line_buf, &line_buf_size, fp);
    }

    free(line_buf);
    line_buf = NULL;

    fclose(fp);

    return 0;
}
void init() {
    memset(ram, 0, sizeof(ram));
    memset(cache, 0, sizeof(cache[0][0]) * CONJUNTOS * ANCHO);
    missRate = 0;

}
unsigned int get_offset(unsigned int address) {
    return address %BLOCKSIZE;
}
unsigned int block_address(unsigned int address) {
    return address / BLOCKSIZE;
}
unsigned int get_tag(unsigned int address) {
    int bin[BITS] = { 0 };
    for (int i = 15; i >= 0; i--) {
        bin[i] = address % 2;
        address = address / 2;
    }
    int etiqueta = 0;
    for (int j = 0; j < 5; j++) {
        etiqueta = etiqueta + bin[4 - j] * pow(2, j);
    }
    return etiqueta;
}

unsigned int find_set(unsigned int address) {
    return block_address(address) %CONJUNTOS;
}
unsigned int select_oldest(unsigned int setnum) {
    return cache[setnum][FIFOINDEX];
}
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set) {
    /* llevo el bloque desde RAM hasta cache */
    for (int i = 0; i < BLOCKSIZE; i++) {
        cache[set][find_blockIndex(way) + VALIDDIRTYTAG + i] = ram[blocknum * BLOCKSIZE + i];
    }
    /* Valid = 1 */
    cache[set][find_blockIndex(way)] = 1;
```

```
    /* Guardo el # de bloque */
    cache[set][find_blockIndex(way)+3] = BLOCKSIZE*blocknum;
}
void write_tomem(unsigned int blocknum, unsigned int way, unsigned int set) {
    for (int i = 0; i < BLOCKSIZE; i++) {
        ram[blocknum + i] = cache[set][find_blockIndex(way) + VALIDDIRTYTAG + i];
    }

}
unsigned int find_blockIndex(unsigned int way) {
    return 1 + way * (BLOCKSIZE + VALIDDIRTYTAG);
}
unsigned char read_byte(unsigned int address) {
    accesosMemoria++;
    int hit = 0;
    int via = 0;
    int viaOriginal = 0;
    int valid = 0;
    int tag = 0;
    int set = find_set(address);
    while (!hit && via < VIAS) {
        valid = cache[set][find_blockIndex(via)];
        tag = cache[set][find_blockIndex(via) + 2];
        if ((get_tag(address) == tag) && (valid)) {
            hit = 1;
            hits++;
            break;
        }
        via++;

    };
    if (!hit) {
        via = cache[set][0];
        viaOriginal = via;
        /* Si Dirty bit = 1, escribo en memoria */
        if(cache[set][find_blockIndex(via) + 1] == 1) {
            // printf("M Block: %d",cache[set][find_blockIndex(via) + 3]);
            write_tomem(cache[set][find_blockIndex(via) + 3], via, set);
        }


        /* Llevo el bloque a la Cache */
        read_tocache(block_address(address), via, set);
        /* Actualizo TAG */
        cache[set][find_blockIndex(via) + 2] = get_tag(address);

        via++;
        // Si llego al final, empiezo de nuevo
        if (via >= VIAS) {
            via = 0;
        }

        cache[set][0] = via;
        misses++;
        return cache[set][find_blockIndex(viaOriginal) + VALIDDIRTYTAG + get_offset(add
    } else {
        return cache[set][find_blockIndex(via) + VALIDDIRTYTAG + get_offset(address)];
    }
}

void write_byte(unsigned int address, unsigned char value) {
    accesosMemoria++;
    int hit = 0;
    int via = 0;
    int valid = 0;
    int tag = 0;
```

```
    int set = find_set(address);
    while (!hit && via < VIAS) {
        valid = cache[set][find_blockIndex(via)];
        tag = cache[set][find_blockIndex(via) + 2];
        if ((get_tag(address) == tag) && (valid)) { // Hit?

            /* Dirty bit = 1 */
            cache[set][find_blockIndex(via) + 1] = 1;

            /* Actualizo cache */
            cache[set][find_blockIndex(via) + VALIDDIRTYTAG + get_offset(address)] = value;
            hit = 1;
            hits++;
            break;
        }
        via++;

    }
    if (!hit) {

        via = cache[set][0];
        /* Si Dirty bit = 1, escribo en memoria */
        if(cache[set][find_blockIndex(via) + 1] == 1) {
            write_tomem(cache[set][find_blockIndex(via) + 3], via, set);
        }
        /* Llevo el bloque a la Cache */
        read_tocache(block_address(address), via, set);

        /* Actualizo TAG */
        cache[set][find_blockIndex(via) + 2] = get_tag(address);
        /* Dirty bit = 1 */
        cache[set][find_blockIndex(via) + 1] = 1;

        /* Actualizo cache */
        cache[find_set(address)][find_blockIndex(via) + VALIDDIRTYTAG + get_offset(address)] = va
        // Recalcular FIFO
        via++;
        // Si llego al final, empiezo de nuevo
        if (via >= VIAS) {
            via = 0;
        }

        cache[set][0] = via;

        misses++;
    }

}
float get_miss_rate() {
    return misses / accesosMemoria;
}
```

# 6. Conclusión

Lo que podemos observar es que no todos los programas funcionan en forma eficiente para la misma arquitectura de caché que use nuestro hardware, lo que se desprende de los diferentes resultados obtenidos en la salida de los programas interpretados. Según los accesos que tenga durante el programa, tener un caché de tantas vías contribuye a reducir la cantidad de misses de conflicto, pero dependerá de cómo se distribuyan los accesos de memoria.
En la mayoría de casos en los que se obtuvo una tasa de MR menor es porque se aprovecha el principio de localidad espacial y temporal, es decir, cuando se acceden a posiciones de memoria contiguas o que se hayan accedido antes.

# Referencias

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

[2] Kernighan, Brian, and Ritchie, Dennis, The C Programming Language.

# 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[2], y correr al menos en Linux.

## 5. Introducción

La memoria a simular es una caché[1] asociativa por conjuntos de ocho vías, de 16KB de capacidad, bloques de 64 bytes, política de reemplazo FIFO y política de escritura 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 tag, el bit `V` y la información necesaria para implementar la política de reemplazo FIFO.

## 6. Programa

Se deben implementar las siguientes primitivas:

```
void init()
unsigned int get_offset (unsigned int address)
unsigned int find_set(unsigned int address)
unsigned int select_oldest(unsigned int setnum)
void read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)
void write_tocache(unsigned int address, unsigned char)
unsigned char read_byte(unsigned int address)
void write_byte(unsigned int address, unsigned char value)
float get_miss_rate()
```

- La función `init()` debe inicializar la memoria principal simulada en 0, los bloques de la caché como inválidos y la tasa de misses a 0.
- La función `get_offset(unsigned int address)` debe devolver el *offset* del byte del bloque de memoria al que mapea la dirección `address`.
- La función `find_set(unsigned int address)` debe devolver el conjunto de caché al que mapea la dirección `address`.
- La función `select_oldest()` debe devolver la vía en la que está el bloque más "viejo" dentro de un conjunto, utilizando el campo correspondiente de los metadatos de los bloques del conjunto.
- La función `read_tocache(unsigned int blocknum, unsigned int way, unsigned int set)` debe leer el bloque `blocknum` de memoria y guardarlo en el conjunto y vía indicados en la memoria caché.
- La función `read_byte(unsigned int address)` debe buscar el valor del byte correspondiente a la posición `address` en la caché; si éste no se encuentra en la caché debe cargar ese bloque. El valor de retorno siempre debe ser el valor del byte almacenado en la dirección indicada.
- La función `write_byte(unsigned int address, unsigned char value)` debe escribir el valor `value` en la posición `address` de memoria, y en la posición correcta del bloque que corresponde a `address`, si el bloque se encuentra en la caché. Si no se encuentra, debe escribir el valor solamente en la memoria.

- La función `get_miss_rate()` debe devolver el porcentaje de misses desde que se inicializó la caché.

Con estas primitivas, hacer un programa que llame a `init()` y luego lea de un archivo una serie de comandos y los ejecute. Los comandos tendrán la siguiente forma:

```
FLUSH
R ddddd
W ddddd, vvv
MR
```

- El comando "`FLUSH`" se ejecuta llamando a la función `init()`. Representa el vaciado del caché.
- Los comandos de la forma "`R ddddd`" se ejecutan llamando a la función `read_byte(ddddd)` e imprimiendo el resultado.
- Los comandos de la forma "`W ddddd, vvv`" se ejecutan llamando a la función `write_byte(unsigned int ddddd, char vvv)` e imprimiendo el resultado.
- El comando "`MR`" se ejecuta llamando a la función `get_miss_rate()` e imprimiendo el resultado.

El programa deberá chequear que las líneas del archivo correspondan a un comando con argumentos dentro del rango estipulado, o de lo contrario estar vacías. En caso de que una línea tenga otra cosa que espacios blancos y no tenga un comando válido, se deberá imprimir un mensaje de error informativo.

# 7. Mediciones

Se deberá incluir la salida que produzca el programa con los siguientes archivos de prueba:

- prueba1.mem
- prueba2.mem
- prueba3.mem
- prueba4.mem
- prueba5.mem

### 7.1. Documentación

Es necesario que el informe incluya una descripción detallada de las técnicas y procesos de medición empleados, y de todos los pasos involucrados en el mismo, ya que forman parte de los objetivos principales del trabajo.

## 8. Informe

El informe deberá incluir:

- Este enunciado;
- Documentación relevante al diseño e implementación del programa, incluyendo las estructuras de datos;
- Instrucciones de compilación;
- Resultados de las corridas de prueba;
- El código fuente completo del programa, en dos formatos: digital e impreso en papel.

## 9. Fecha de entrega

La fecha de entrega es el jueves 14 de Noviembre de 2019.

## Referencias

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

[2] Kernighan, Brian, and Ritchie, Dennis, The C Programming Language.