ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Lab session 10

MIPS R2000 CACHE MEMORY
CODE CACHE

1. Introduction

In this lab session we will work with the MIPS R2000 cache. The concepts studied in previous labs are fundamental to understanding the cache memory operation and its influence on the programs execution time. The working tool is the MIPS R2000 processor simulator called PCSpim-Cache.

1.1. Goals

- To determine how cache memory helps to reduce the access time to information, in this lab we will concentrate on the instructions.
- To know how to interpret cache memory addresses issued by the processor.
- To analyze how the cache memory organization affects its hit rate.

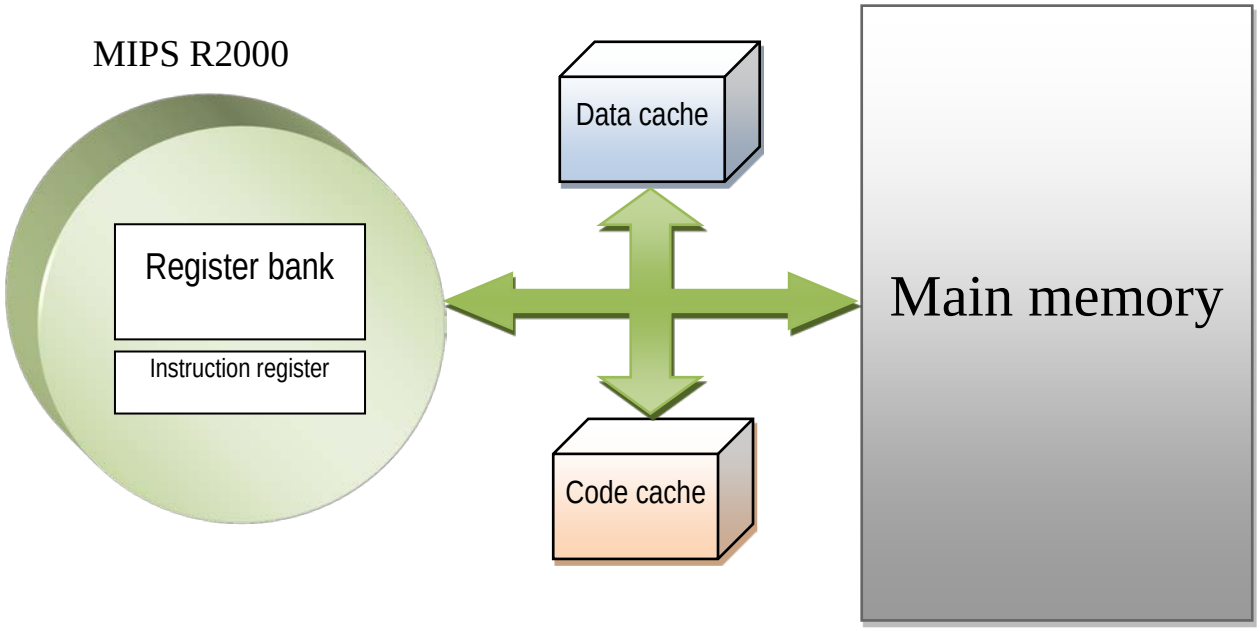
1.2. Material

The material required is available on PoliformaT folder: Resources -> Lab -> P10.

1.3. Configuration of the PCSpim-Cache simulator

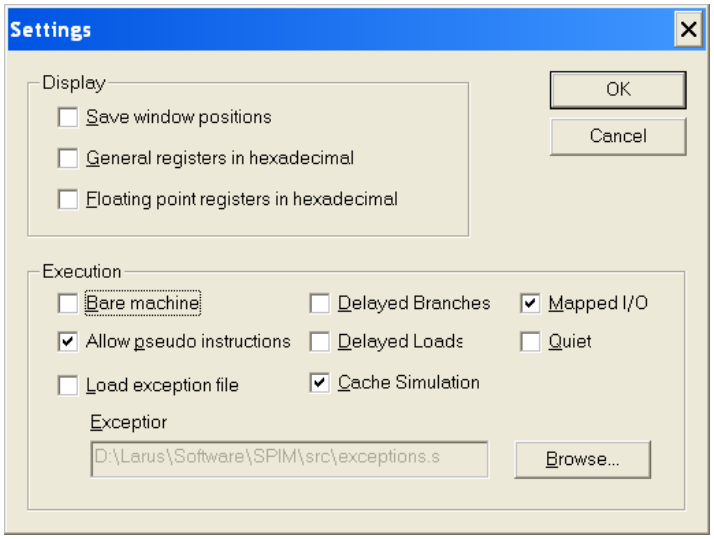
The simulator PCSpim-Cache is an extension of the basic PCSpim simulator that allows the execution of programs written in MIPS R2000 processor assembler taking into account the system cache. The MIPS R2000 processor is designed, in reality, to use a single system of first level cache (L1) that consists of an instruction cache and data cache (see Figure 1), both external to the processor. The data cache stores information located in the data segment, ie one that is read or written by the load and store instructions (lb, lbu, lh, lhu, lw, lwc1, sb, sh, sw, swc1), the code cache is used exclusively for storing code segment information, namely instructions, and therefore it is only accessed in read operations. In summary, the data cache is an intermediary between main memory and the register bank, while code cache is between main memory and the instruction register.

Figure 1. The cache MIPS R2000 processor



The simulator includes a first level cache memory organized in two memories, one for data and another for instructions. To enable simulation of cache memories you must select the option "Cache Simulation" within the simulator settings as shown in Figure 2. Also accessible through the menu Simulator | Settings

Figure 2. Enabling cache simulation



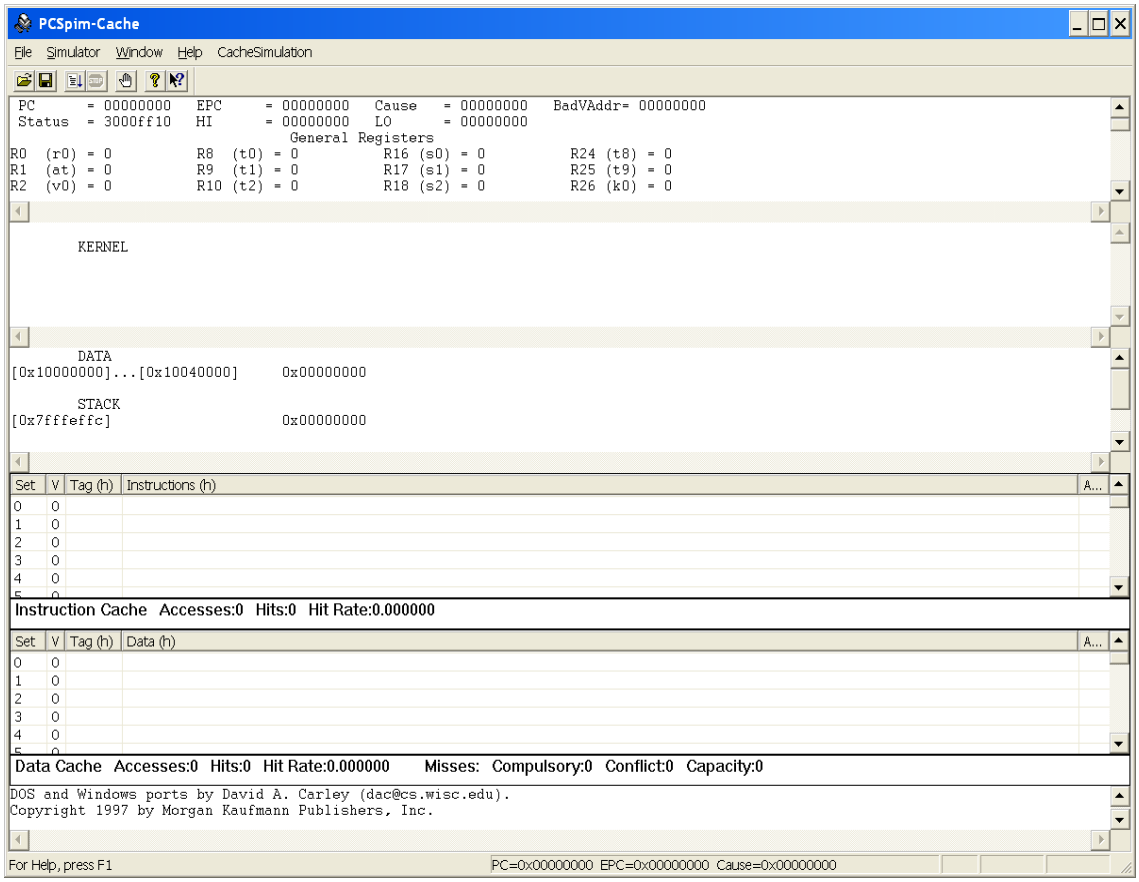
When this option is active, a new dropdown menu called Cache Simulation appears in the main window of the simulator. In this menu you choose the cache type to simulate: data, code or both (Cache Configuration option) and the organization and policies to be applied (Cache Settings option).

Before specifying the cache organization you must choose the type of cache to simulate: data or instructions. It is also possible to simulate two separate caches (instruction and data) simultaneously (Harvard architecture). For each of the cache memories to be simulated, an

independent frame appears as shown in Figure 3. At the foot of each of these frames is a line of text that displays statistics associated with each of the caches (number of accesses, hits, etc..).

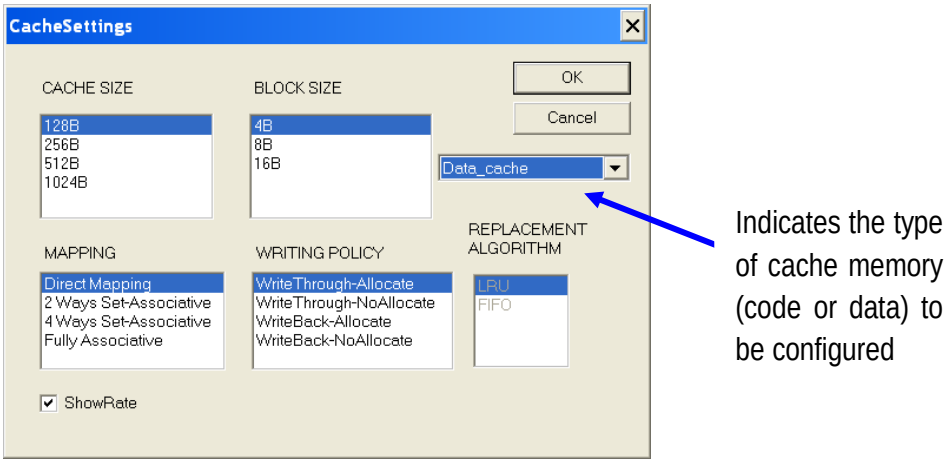
Regarding the graphic representation, each set of the cache is shown in a separate row. That is, the information corresponding to all channels of the same group (tags, valid bit, data, LRU value, etc.) is shown in the same row of the display. This arrangement only changes in fully associative cache correspondence, in which case each line represents a path, since in this cache type there is only one correspondence set.

Figure 3. Main window with a data cache and instruction



As mentioned previously, after selecting the type of cache memory to simulate the parameters that define its organization should be provided. Figure 4 shows the corresponding dialog. As shown, you have to specify the geometry of the cache (memory capacity, block size and number of tracks), writing policies, and the replacement algorithm. Also you should select the option ShowRate if statistics are to be displayed on screen.

Figure 4. Cache configuration dialog



Once the configuration is done you can proceed with loading the program written in assembly language and its execution. This process is performed in exactly the same way as in the basic PCSpim simulator.

2. Test program: product of a vector by a constant

The solution of many problems of numerical calculation requires the multiplication of the elements of a vector by a constant: Y = k * X Below is the code for an assembly program that performs this operation. Integers handled by the program are encoded in two's complement and are 32-bit long. The program assumes that the result of the product k * X [i] does not exceed 32 bits. This program is similar to the one studied in a previous practice with slight changes. Since this code is going to be modified, the program shown below will be referred to as the *original program*.

```
                ##########################################################
                # Data segment
                ##########################################################

                .data 0x10000000
A:              .word 0,1,2,3,4,5,6,7   # Vector A
                .data 0x10001000
B:              .space 32               # Vector B (result)
                .data 0x1000A030
k:              .word 7                 # Scalar constant
dim:            .word 8                 # Vector dimension

                ##########################################################
                # Code segment
                ##########################################################

                .text 0x00400000
                .globl __start

__start:        la $a0, A               # $a0 = A address
                la $a1, B               # $a1 = B address
                la $a2, k               # $a1 = k address
                la $a3, dim             # $a2 = dimension address
                jal sax                 # Subroutine call
```

```
                ######################################################
                # Execution ending with a system call
                ######################################################

                addi $v0, $zero, 10     # Exit code
                syscall                 # Execution end

                ######################################################
                # Subroutine that computes Y <- k*X
                # $a0 = Starting address of vector X
                # $a1 = Starting address of vector Y
                # $a2 = Address of scalar constant k
                # $a3 = Address of dimension
                ######################################################

sax:            lw $a2, 0($a2)          # $a3 = constant k
                lw $a3, 0($a3)          # $a3 = dimension
loop:           lw $t0, 0($a0)          # Reading X[i] into $t0
                mult $a2, $t0           # Computes k*X[i]
                mflo $t0                # $t0 <- k*X[i] (HI value is 0)
                sw $t0, 0($a1)          # Writing Y[i]
                addi $a0, $a0, 4        # Address of X[i+1]
                addi $a1, $a1, 4        # Address of Y[i+1]
                addi $a3, $a3, -1       # Decrements the number of elements
                bgtz $a3, loop          # Jumps if elements remain
                jr $ra                  # Subroutine return

                .end
```

Before seeing the relationship between the implementation of this program and the cache system, it is necessary to analyze its structure and behavior. There is no need to upload it yet to the simulator, just analyze it on paper and understand how it works. At this point it is important to be aware that, during program execution, each instruction executed has been read from the code segment and taken to the instruction register for decoding. Likewise, it should be noted that the vector A is accessed in read operations (load) while B is accessed by write operations (store).

1. How many elements have the vectors of the program? How many bytes occupy each element?

First we will determine the size occupied by program variables in the data segment and the size occupied by the program's instructions in the code segment.

2. Complete the following information in the data segment. Use the hexadecimal system to express memory addresses (do the same throughout the entire practice).

Starting address of vector A	
Bytes occupied by vector A	
Initial direction of vector B	
Bytes occupied by vector B	
Address of variable k	
Address of variable dim	

3. Complete the following information about the code segment. In this case do not forget to consider translating the program pseudoinstructions into machine instructions, since the latter are the only ones to consider. In this case it will be beneficial to load the program in the simulator (no need to run it) to see the address where the last instruction of the program is allocated.

Address of the first instruction	
Address of the last instruction	
Number of program instructions	
Bytes occupied by the program code (instructions)	

From this point we will take an interest in the dynamic aspects of the program and their relationship with memory. The most important thing now is to know the number of memory accesses made by the program. These accesses are made to both the code segment (remember that each instruction is read from memory to carry it to the instruction register) and the data segment (for reading or writing program variables by instructions load and store).

4. Determine the number of accesses to memory done by the program. These values are very important because they will help us later to know what is the total number of accesses served by the cache, that is, we can distinguish between accesses that are hits and accesses that are faults.

Accesses to data segment	
Accesses to code segment	

3. Code cache

We now consider a code cache with the following features:

Parameter	Value
Capacity	128 bytes
Correspondence	Direct
Block or line size	4 bytes

For the moment there is **no need to use the simulator**, we'll do that a little later. Remember that a code cache **receives only read operations**, since the processor is limited to read the instructions to execute. Also, note that a line can only store one instruction and that the entire program fits in the cache.

5. Considering the above features, indicate how many lines are in the cache memory.

6. Indicate what will be the interpretation that this cache will make regarding the receiving addresses (tag fields, line and offset).

7. The program instruction `jal sax` is stored in the address `0x0040001C` data segment. Indicate what cache line will allocate it and its tag.

The operation of cache memory is based on the control information stored in each of the lines. This control information includes a valid bit, tag bits and, as appropriate, the modified bits, counter bits to the replacement algorithm, etc.

8. Calculate, for this case, how many control bits are stored per line. Similarly, calculate the volume of the directory, that is, the total number of control bits contained in the code cache.

Control bits per line	
Directory volume (bytes)	

Now you are going to use the simulator. Define a cache system for both data and instructions (Harvard architecture). Do not worry about the data cache (to study later), we currently focus on the code cache. We will configure it with the features we have defined above (capacity of 128 bytes, direct correspondence and line size of 4 byte).

9. Load the *original program* and run it by selecting F10 (step by step) to follow in detail the effect on the code cache. Note that reading any instruction affects the code cache, but the execution of memory instructions (lw, sw, etc..) affects also the data cache not considered now. Also, note that the processing of the first 18 instructions only originates faults (*miss* message in the simulator) and that the first hit occurs in the nineteenth access to the instruction cache. Complete the following table:

Accesses to code segment	
Hits	
Faults	
Hit rate (H)	

10. Confirm that the `sax jal` instruction is stored in the expected line with the tag calculated above.

11. Suppose now that the main memory is implemented with MIPS R2000 modules with chips running at 50 MHz (20 ns period) and having as parameters tCL = 2 (CAS latency) and tRCD = 3 (time between RAS and CAS), both of them are expressed in terms of clock cycles. Suppose further, that the access time to the cache memory is 10 nanoseconds. Remember that this time can be calculated using the formula:

$$G = H \times T_{success} + (1-H) \times T_{failure}$$

In this context, obtain the average access time to the code segment experienced by the program.

3.1. Taking advantage of the locality principle

As you've seen, with the above cache configuration every cache line can contain only one instruction. If we take the locality principle of the program, in order to reduce memory access time, we can make the block size bigger. Thus we achieve as a result that a line fits several instructions.

12. Use the simulator and set the code cache with a block size of 16 bytes keeping all other parameters as before. Load and run now the original program and complete the following table:

Accesses to code segment	
Hits	
Faults	
Hit rate (H)	

13. As shown, the number of faults has been reduced considerably. What is the reason why?