ESTRUCTURA DE COMPUTADORES
Grado en Ingeniería Informática

Lab session 10

MIPS R2000 CACHE MEMORY
THE INSTRUCTION CACHE

1. Introduction

This is the first of two lab sessions about the MIPS R2000 cache. We will be using the MIPS simulator *PCSpim-Cache*, a modified version of PCSpim that simulates cache.

1.1. Goals

- To observe how cache memory helps reducing the access time to information (instructions, in this lab).
- To determine the address structure from the cache point of view.
- To analyze how the cache organization affects 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.

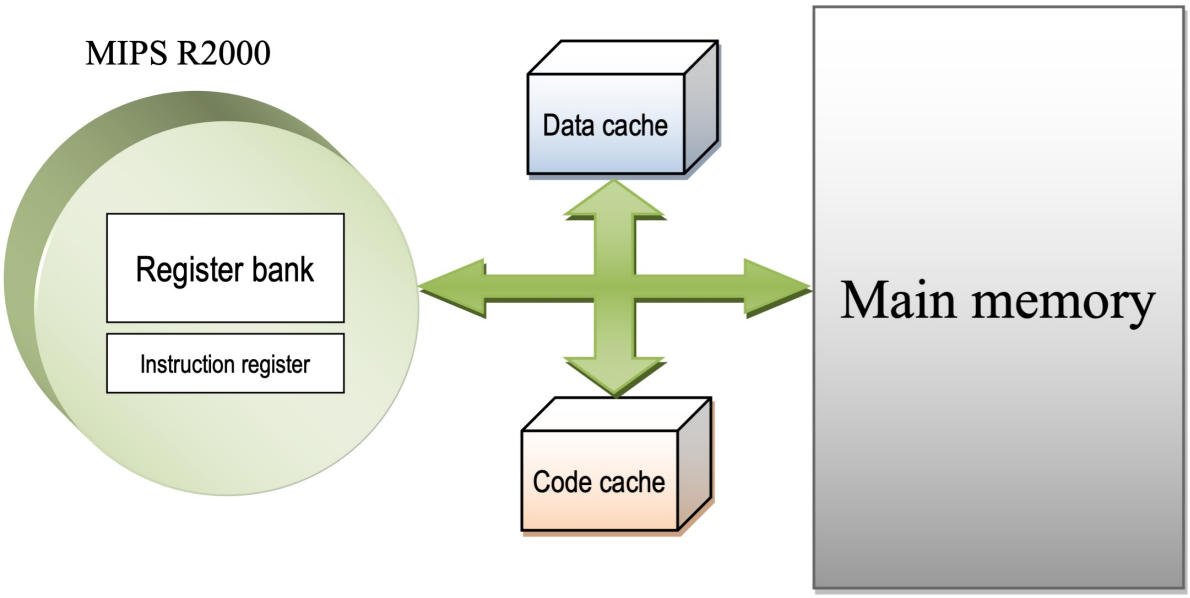

Figure 1. The cache MIPS R2000 processor

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

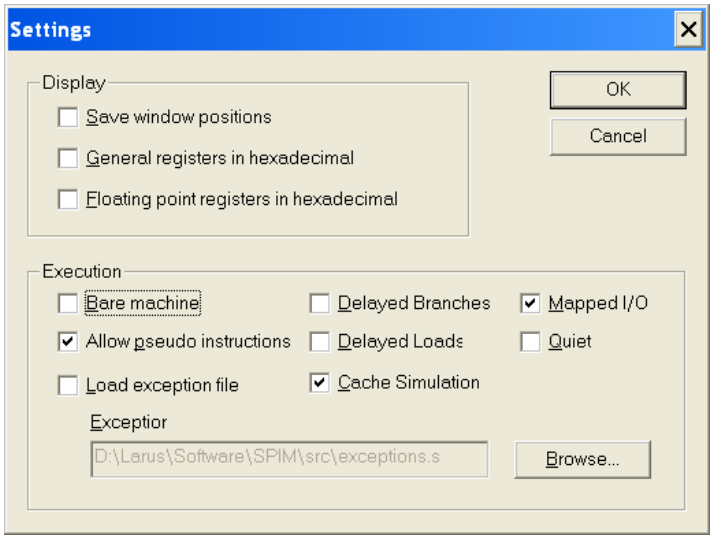

Figure 2. Enabling cache simulation

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.

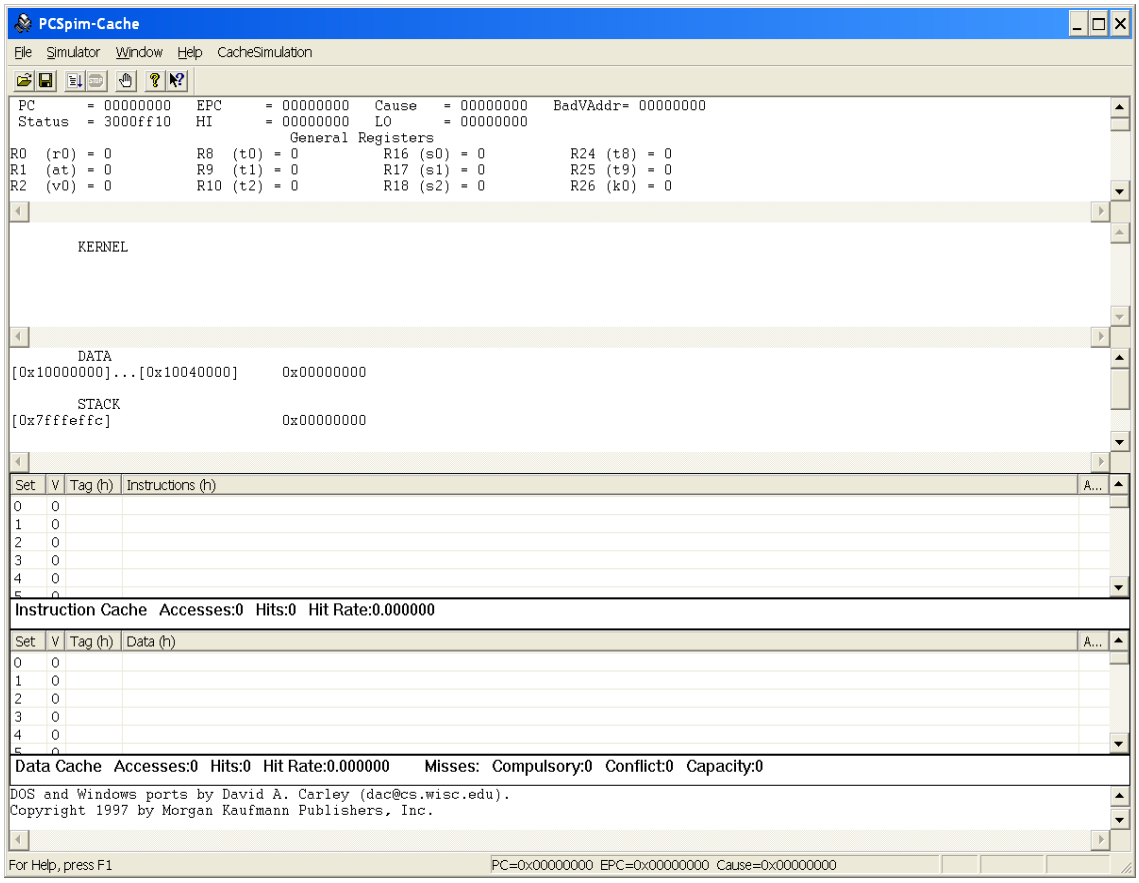

Figure 3. Main window with a data cache and instruction

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.

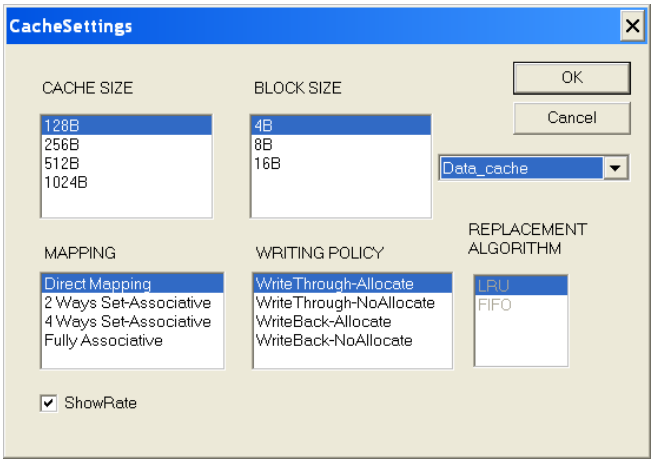

Figure 4. Cache configuration dialog

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 components can be stored in the vectors of the program? How many bytes per component?*

First, we will determine the amount of memory taken by the program variables in the data segment and the program instructions in the code segment.

> ***2.*** *Complete the following information regarding the* ***data segment****. Give the addresses in hexadecimal (here, and through the rest of exercises).*

Starting address of vector *A*	
Bytes taken by vector *A*	
Starting address of vector *B*	
Bytes taken by vector *B*	
Address of variable *k*	
Address of variable *dim*	

3. *Complete the following information about the code segment. You need to translate the pseudo-instructions in the program into machine instructions, since that is what the code memory contains: actual machine instructions. You can have PCSpim do the work by loading the program in the simulator (no need to run it yet) and inspect the code addresses there.*

Address of the first instruction	
Address of the last instruction	
Number of program instructions	
Bytes taken by the program	

Now that we know how data/instructions are allocated in memory for this program, we will now investigate the program interaction with memory at run time. The first thing we need to know is the number of memory accesses made by the program, both for fetching instructions and for loading or storing data.

4. *Determine the number of accesses to the data and code segments made by this program. Later on, these values will help us determine the achieved hit rate.*

Accesses to data segment	
Accesses to code segment	

3. Code cache

We now consider a **code** cache with the following parameters:

Parameter	Value
Capacity	128 bytes
Mapping	Direct
Line size	4 bytes

There is **no need to use the simulator** at the moment, we'll do that later. Remember that a code cache **receives only read operations**, since the processor is limited to read the instructions to execute. Note also that, with these parameters, a cache line can contain only one instruction and the entire program fits in the code cache.

5. *What is the number of lines in this cache?*

6. *Give the address structure form the cache point of view (tag, line, and offset fields).*

7. *The program instruction* `jal sax` *is stored in address* `0x0040001C`*. Give its corresponding cache line and 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 the number of control bits per cache line and the directory volume, that is, the total number of control bits required by the code cache.*

Control bits per line	
Directory volume (bytes)	

Time to use the PCSpim-Cache 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 mapping and line size of 4 bytes).

9. *Load the original program and run it step by step (F10 key) to follow in detail the effect on the code cache. Observe how instruction fetch affects the code cache, but the execution of memory instructions (lw, sw, etc..) affects also the data cache (not considered today). Also, note that fetching the first 18 instructions result in cache misses and that the first hit occurs in the nineteenth access to the instruction cache. Complete the following table:*

Accesses to code	
Hits	
Misses	
Hit rate (H)	

10. *Confirm that the instruction* `jal sax` *is stored in the expected line and with the tag you obtained in question 7 above.*

11. *Assume now that the main memory uses modules with DRAM chips running at 50 MHz (20 ns period) and with parameters CL = 2 cycles (CAS latency) and* t_{RCD} *= 3 cycles (time between RAS and CAS). The cache access time is 10 nanoseconds. Calculate the average access time to the code segment for this program. Remember that this time can be calculated as:*

$$T_m = H \times T_{hit} + (1-H) \times T_{miss}$$

3.1. Taking advantage of locality

With the above cache configuration, a cache line can contain only one instruction. As a consequence, we are not exploiting the spatial locality present in sequential code. For that purpose, we need to make the block size larger. Let's see the impact of this change.

12. *Use the simulator and configure a block size of 16 bytes, keeping all other parameters as before. Load and run now the original program and fill out the following table.*

Accesses to code segment	
Hits	
Faults	
Hit rate (H)	

13. The number of faults has been considerably reduced by enlarging the block size. How do you explain?