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 use the MIPS simulator ***PCSpim-Cache***, a modified version of PCSpim that simulates cache.

1.1. Goals

- To observe how cache memory helps reduce the access time to information (instructions, in this lab).
- To determine the address structure from the cache point of view.
- To analyse how cache organisation affects the hit rate.

1.2. Material

The material required is available in the PoliformaT folder: *Resources →Lab →P10*.

1.3. Configuration of the PCSpim-Cache simulator

The simulator PCSpim-Cache is an extension of PCSpim that includes cache simulation. The MIPS R2000 processor was designed to use a dual L1 cache as a separate chip from the processor. The data cache stores information located in the data segment, i.e., the data that is read or written by the available load and store instructions (*lb*, *lbu*, *lh*, *lhu*, *lw*, *lwc1*, *sb*, *sh*, *sw*, *swc1*). In contrast, the instruction cache is used exclusively for storing code segment information, namely instructions. The instruction cache (or code cache) is only accessed for fetching (reading) instructions; hence, it is a read-only memory for user programs. The data cache is thus an intermediary between the main memory and the register file, and the code cache is located between main memory and the instruction register (IR).

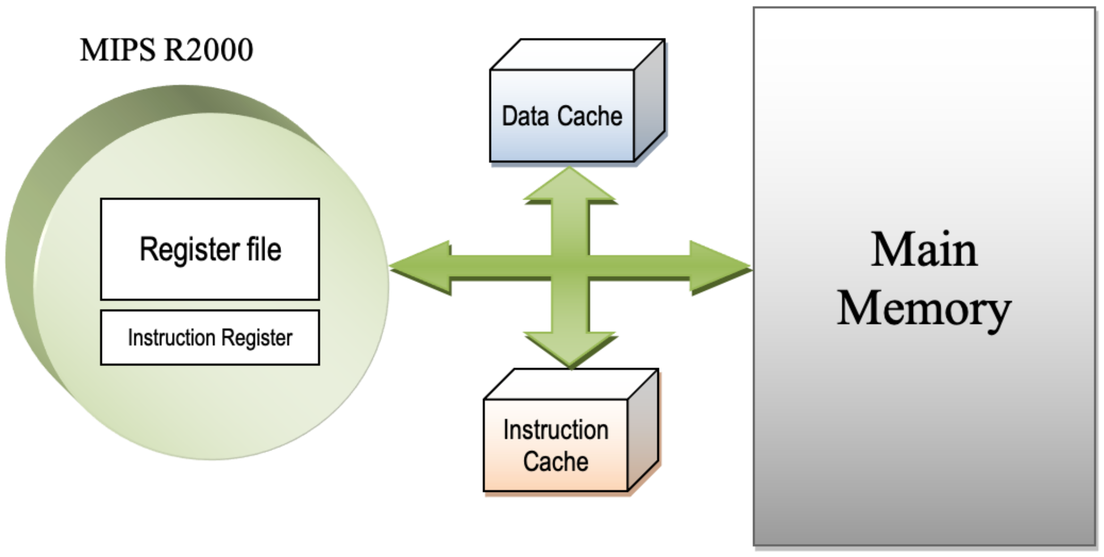
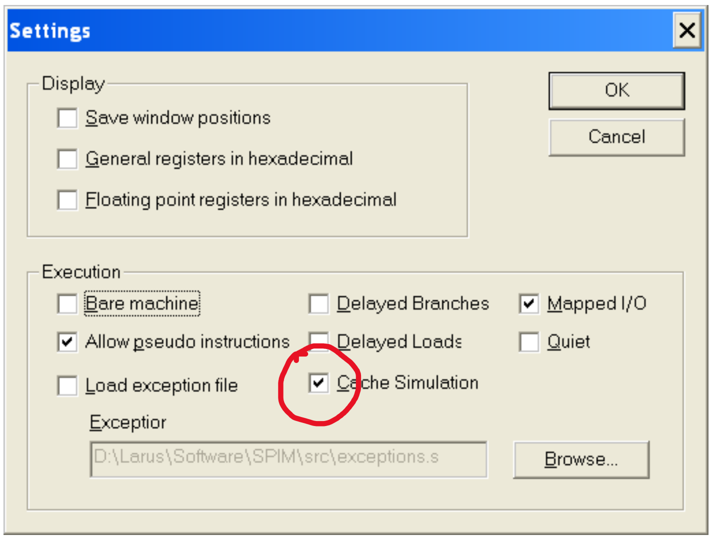


Figure 1. The dual cache of a MIPS R2000 processor

To enable the simulation of cache memories in PCSpim-Cache, you must tick the "Cache Simulation" checkbox in the *Simulator Settings* dialogue (see Figure 2). This is accessible from the simulator's top menu, Simulator | Settings.



Settings

Display
- [ ] Save window positions
- [ ] General registers in hexadecimal
- [ ] Floating point registers in hexadecimal

Execution
- [ ] Bare machine
- [✓] Allow pseudo instructions
- [ ] Load exception file
- [ ] Delayed Branches
- [ ] Delayed Loads
- [✓] Cache Simulation
- [✓] Mapped I/O
- [ ] Quiet

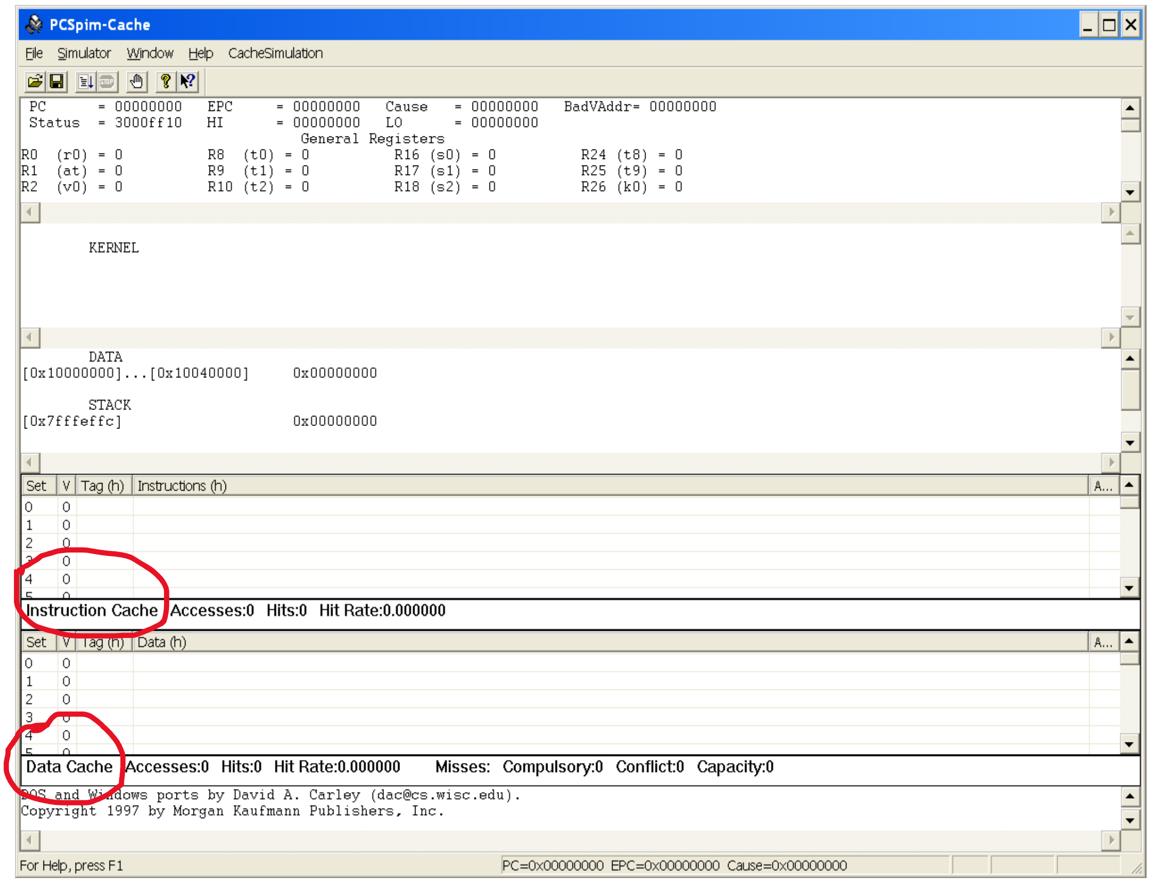
Exception: D:\Larus\Software\SPIM\src\exceptions.s

Figure 2. Enabling cache simulation in PCSpim-Cache

When this option is ticked, as highlighted in Figure 2, a new Cache Simulation dropdown menu will be available in the main simulator's window. From this menu, you can choose the cache type to simulate (***Cache Configuration*** submenu) and the organisation and policies to be applied (***Cache Settings*** submenu).

First, you use the *Cache Configuration* submenu to choose the type of cache to simulate: data only, instructions only, or a Harvard architecture, i.e., two separate caches, one for instructions and one for data. For each simulated cache, an independent frame appears in the main simulator's window, as shown in Figure 3. At the foot of each of these frames is a line of text that shows statistics associated with each cache (number of accesses, hits, etc..).

In PCSPim-Cache, each cache set is shown in a separate row with the value of control bits (tag, valid bit, etc.). This arrangement only changes with fully associative mapping, where each row represents a line since, in that case, there is only one set.



```
 PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                  General Registers
R0  (r0) = 0          R8  (t0) = 0          R16 (s0) = 0          R24 (t8) = 0
R1  (at) = 0          R9  (t1) = 0          R17 (s1) = 0          R25 (t9) = 0
R2  (v0) = 0          R10 (t2) = 0          R18 (s2) = 0          R26 (k0) = 0

        KERNEL

        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000
```

Set	V	Tag (h)	Instructions (h)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		

Instruction Cache Accesses:0 Hits:0 Hit Rate:0.000000

Set	V	Tag (h)	Data (h)
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		

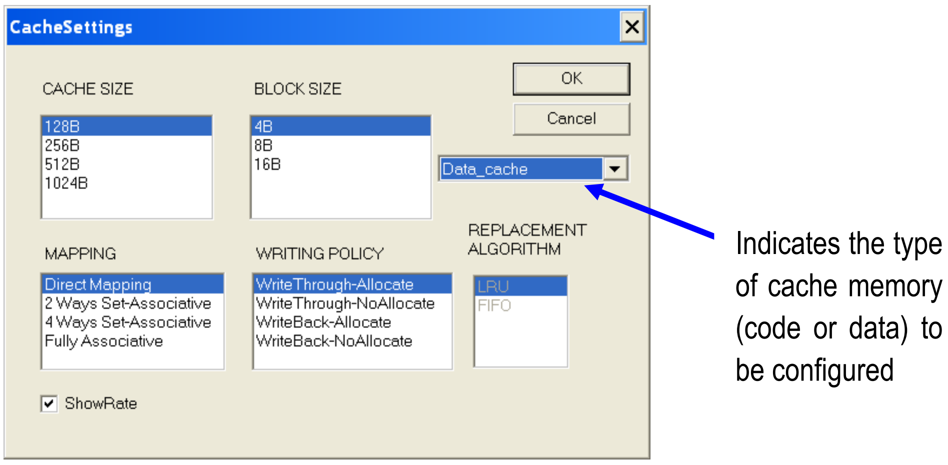
Data Cache Accesses:0 Hits:0 Hit Rate:0.000000 Misses: Compulsory:0 Conflict:0 Capacity:0

```
DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
```

PC=0x00000000 EPC=0x00000000 Cause=0x00000000

Figure 3. Main window with instruction and data cache (dual cache)

After selecting the type of cache memory to simulate, you must define the cache organisation using the *Cache Settings* submenu. Figure 4 shows the corresponding dialogue box. The parameters to specify are the cache geometry (capacity, block size and number of ways) and, when applicable, the write policies and the replacement algorithm. Remember to tick the checkbox *Show Rate* if you want cache statistics to be shown.


CacheSettings
OK
Cancel
Indicates the type of cache memory (code or data) to be configured

CACHE SIZE: 128B, 256B, 512B, 1024B (selected: 128B)

BLOCK SIZE: 4B, 8B, 16B (selected: 4B)

Data_cache

MAPPING: Direct Mapping, 2 Ways Set-Associative, 4 Ways Set-Associative, Fully Associative (selected: Direct Mapping)

WRITING POLICY: WriteThrough-Allocate, WriteThrough-NoAllocate, WriteBack-Allocate, WriteBack-NoAllocate (selected: WriteThrough-Allocate)

REPLACEMENT ALGORITHM: LRU, FIFO

[✓] ShowRate

Figure 4. Cache Configuration dialogue

Once you configure the cache settings, you can load and execute your program with the PCSpim simulator. The cache simulation results will be shown in their corresponding frames in the simulator's main window.

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

The solution to numerical calculation problems often requires the multiplication of a vector *A* by a constant *k*, i.e., $B = k \times A$. Below is the code of an assembly program that performs this operation using a given vector *A* of signed integers and an integer *k*. The program assumes that the result of each product $k \times A\ [i]$ does not exceed 32 bits. Since this code will be modified in later exercises, it 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 points to A
                la $a1, B               # $a1 points to B
                la $a2, k               # $a1 points to k
                la $a3, dim             # $a2 points to dim
                jal sax                 # Subroutine call

                # End program
                addi $v0, $zero, 10     # Exit code
                syscall                 # Call exit function
```

```
                 #######################################################
                 # 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)           # Read X[i] into $t0
                 mult $a2, $t0            # Compute k*X[i]
                 mflo $t0                 # $t0 <- k*X[i] (HI value is 0)
                 sw $t0, 0($a1)           # Write Y[i]
                 addi $a0, $a0, 4         # Address of X[i+1]
                 addi $a1, $a1, 4         # Address of Y[i+1]
                 addi $a3, $a3, -1        # Decrement the number of elements
                 bgtz $a3, loop           # Jump if elements remain
                 jr $ra                   # Subroutine return

                 .end
```

Before looking at the relationship between the implementation of this program and the cache system, observe its structure and behaviour. Do not load it yet on the simulator. For the moment, we will just analyse it on paper and try to understand how it works. Remember that, during program execution, each instruction executed has been read from the code segment (ideally from the instruction cache) into the instruction register. Regarding the data segment, vector A is accessed with load operations only, while B is accessed only by store instructions.

1. *Find the declarations of vectors A and B in this program. How many components do they contain? How many bytes per component?*

   8 components, 4 bytes per component

We will now 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 the exercises).*

Starting address of vector *A*	0x1000 0000
Bytes taken by vector *A*	32
Starting address of vector *B*	0x1000 1000
Bytes taken by vector *B*	32
Address of variable *k*	0x1000 A030
Address of variable *dim*	0x1000 A034

3. *Complete the following information about the **code segment**. You need to translate the pseudo-instructions in the program into machine instructions since 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 finding the instructions' addresses in the simulator.

Address of the first instruction	0x0040 0000
Address of the last instruction	0x0040 0050
Number of program instructions	21
Bytes taken by the program	21 * 4 = 84

Now that we know how the program's data and instructions are statically allocated to memory, we will investigate the program's interaction with memory at run time. We first want to find the number of accesses to memory made from this program, both for fetching instructions and loading or storing data. This will help us determine the achieved hit rates later.

4. *Determine the number of accesses to this program's data and code segments.*

Accesses to data segment	18
Accesses to code segment	77

3. Code cache

Consider a **code** cache with the following configuration parameters:

Parameter	Value
Capacity	128 bytes
Mapping	Direct
Line size	4 bytes

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

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

128 / 4 = 32 lines

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

tag: 25   line: 5 bits   offset: 2 bits

First, the offset is 2 bits, as block size is 4B. Then, as there are 32 lines, they occupy 5 bits. The rest of the bytes are for the tag.

7. *The program instruction* `jal sax` *is stored in address* `0x0040001C`*. Give its corresponding cache line and tag.*

```
Address = 0000 0000 0100 0000  0000 0000 0001 1100
          ------------------------------------------|--------|---
                         32768                           7      0
```

Cache operation is based on the control information stored in each line. This control information includes a valid bit, tag bits, and, when applicable, the dirty bit, additional bits for the replacement algorithm, etc.

8. *Calculate the number of control bits per cache line and the directory volume, i.e., the total number of control bits required by the code cache.*

Control bits per line	25 (tag) + 1 (valid bit)
Directory volume (bytes)	26 * 32 = 832 / 8 = 104

It's time to use the cache simulation part of PCSpim-Cache. Define a cache system for both data and instructions (Harvard architecture). Ensure you configure the instruction cache with the abovementioned parameters (capacity of 128 bytes, direct mapping, and line size of 4 bytes). Do not worry about the data cache; we focus on the code cache in this lab session.

9. *Load the original program and run it step by step (F10 key*[1]*) 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..) also involves the data cache (not considered today). Also, note that fetching the first 18 instructions results in cache misses and that the first hit occurs in the nineteenth access to the instruction cache. Complete the following table:*

Accesses to code (see question 4)	77
Hits	56
Misses	21
Hit rate (H)	0.727

10. *Double-check 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 SDRAM 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:*

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

Tm = 56/77 * 10 + 21/77 * ((3 + 2) * 20ns + 10ns) = 37.27ns

[1] Note that *PCSPim-Cache* requires you to press F10 three times to advance one instruction. This feature facilitates tracking the effects of every memory interaction on the cache.

3.1. Taking advantage of locality

A cache line can contain only one instruction with the above configuration (4-byte lines). Consequently, we are not exploiting the spatial locality of executing sequential code. A larger block size would bring more neighbouring instructions and increment the hit rate. Let’s evaluate the impact of increasing the block size.

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

Accesses to code segment	77
Hits	71
Faults	6
Hit rate (H)	0.922

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

For each miss, we now bring that instruction and 3 neighboring instructions, so when we access those instructions, a miss doesn't occur.