Lab 12: Synchronization by polling

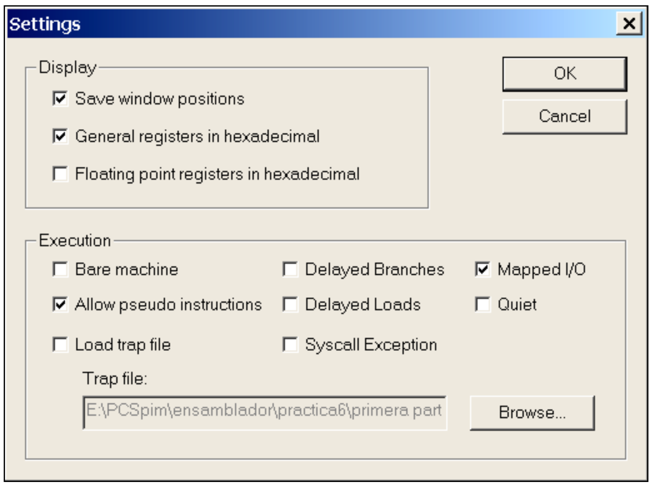
Objectives

Consolidating knowledge about I/O synchronization based on polling the peripherals. To achieve this goal the PCSpim simulator has been extended to include a keyboard and a console.

Material

In this lab session we will use the **PCSpim-ES** simulator and the source code files *wait.asm* and *eco.asm*. All is available at the lab session folder in PoliformaT.

PCSpim-ES is a *PCSpim* simulator version that includes two peripherals. It will also be used in the next lab session. Make sure the simulator you are using is the right one by opening the window *Help>AboutPCSpim* once the simulator is open. You should see the text “PCSpim Version 1.0 - adaptación para las asignaturas ETC2 y EC, etc...”. Fig. 1 shows the appropriate configuration in the simulator configuration window (*simulator>settings*). Notice that there is a new check box, *Syscall Exception*, which must remain unchecked.


Settings

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

Execution
- [ ] Bare machine
- [✓] Allow pseudo instructions
- [ ] Load trap file
- [ ] Delayed Branches
- [ ] Delayed Loads
- [ ] Syscall Exception
- [✓] Mapped I/O
- [ ] Quiet

Trap file: E:\PCSpim\ensamblador\practica6\primera part

Figure 1. Simulator configuration settings

I/O in PCSpim

Usually, user programs don’t access peripheral interfaces through memory access instructions. Furthermore, the operating system prevents user programs from accessing I/O devices directly for many reasons. Instead, programs must rely on input/output system functions that ensure safety and efficiency. These functions are the ones that ultimately access the registers in the interfaces by means of memory instructions (load/store).

The PCSpim environment is similar to an operating system but with some important differences:

- Predefined I/O functions (Appendix. PCSpim system functions) are simple and they are not prepared to support concurrency.
- User programs don’t have to use system calls in order to access peripherals (fig. 2): they are allowed to access them directly without restrictions.

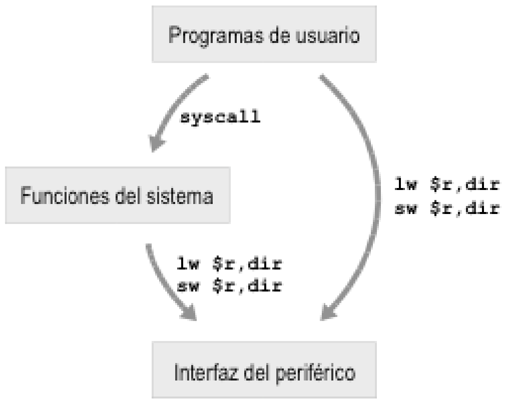


Figure 2. Peripheral Access in PCSpim

There are two peripherals available: keyboard and console (fig. 3). The keyboard and console are the usual PCSpim ones with the associated functions: `read_string` and `print_char`. Each peripheral has a simulated adapter that creates an interface you have to work with.

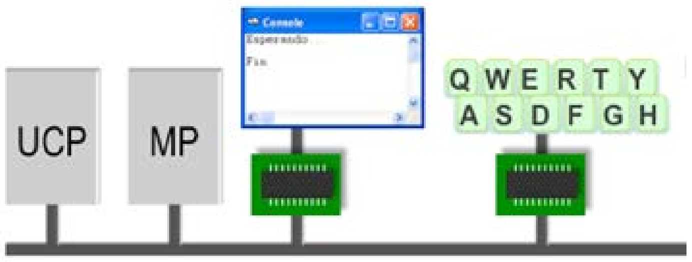


Figure 3. Complete PCSpim diagram including the keyboard and the console

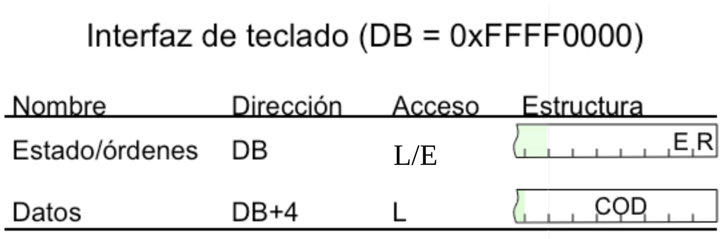
1. The keyboard interface

When PCSpim is running, the keyboard is an input device with the basic operation of reading one character. That input operation takes place when a key is pressed and the interface supplies the code for the typed character. The keyboard interface consists of two registers described in figure 4. Both registers are accessible in the MIPS addressing space from the base address DB = 0xFFFF0000 and they are 32-bit wide. The useful bits are the ones located at the least significant part of the registers content.

You can use any variant of the load instruction (`lw`, `lhu`, `lh`, `lbu` or `lb`) because the bits of interest reside in the least significant byte. Bit E will be used in the next lab session, so for now it should be kept as E = 0.

The keyboard ready bit

Bit R indicates that the keyboard is ready. Every time a key is pressed, this bit is set to 1. By polling this bit, a program becomes aware when there is a new character available for reading from the keyboard.



Interfaz de teclado (DB = 0xFFFF0000)

Nombre	Dirección	Acceso	Estructura
Estado/órdenes	DB	L/E	E R
Datos	DB+4	L	COD

Control and state register (Read/Write. Address = Base)

- R (bit 0, read only). Ready flag: R = 1 every time a key is typed. Reading the data register at DB+4 makes R = 0 again.
- E: (bit 1, read/write). Enable interrupt (while E = 1, R = 1 activates the device interrupt line)

Data register (Read only. Address = Base + 4)

- COD (bits 7...0). ASCII code of the typed character. Reading this register makes R = 0.

Figure 4. Keyboard interface description, in this lab session bit E must be 0

Activity 1. Understanding the polling loop

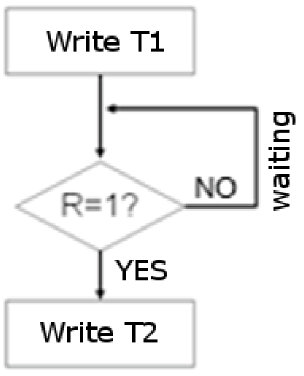
► Using a text editor, open the *wait.asm* file and look at the code. Identify its three main parts: writing the string T1 on the console by using a system call, the polling loop to be studied here and writing T2 on the console.

Figure 5 shows the program structure and the details of the polling loop, that waits for the ready bit to become 1. Pay attention to the following details on the polling loop:

- The interface's base address `0xFFFF0000` is loaded to `$t0`.
- The interface state register is read (offset 0 from the base address).
- Bit R is selected by means of an AND operation with the appropriate bit mask.
- The loop only ends when R=1.

► Open the program *wait.asm* in the simulator and execute it with *Simulator>Go* (F5). You will see the text T1 is immediately printed on the console and, after a key is typed, you'll see T2.

► Execute the program again without closing the simulator (for instance, with *Simulator>Reload* in the PCSpim menu). You will notice that there is no waiting: the reason for this behaviour is that the ready bit keeps the value R = 1 from the previous execution.



```
# Wait for key typing
      la $t0, 0xffff0000
wait: # Waits bit R == 1
      lw $t1,0($t0)
      andi $t1,$t1,1
      beqz $t1,wait
```

Figure 5. Diagram and code of *wait.asm*

► ***Question 1.***

If we changed the keyboard interface so that the R bit occupies position 5 in the control/status register, which instruction would you change and how?

Activity 2. Device cancellation

Cancellation is the mechanism that returns R back to 0 and leaves the keyboard ready for a new operation. In the keyboard interface, cancellation occurs as an effect of reading the data register.

► **Question 2.** Modify *wait.asm* by adding an instruction that reads the data register and leaves the read value in *$t2*, as shown in figure 6. Note that the address of this register is expressed in figure 4 as "DB+4".

► Try to run the modified program, several times without closing the simulator. You will notice that the program always waits for a key to be typed because the ready bit R is 0 at the end of the execution.

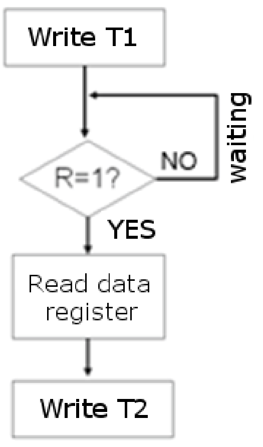


Figure 6. Diagram of modified *wait.asm*

2. Synchronization by polling

When a program waits until a device is ready by repeatedly reading its state register, we say that it is doing synchronization by polling. The general scheme for polling synchronization is shown in figure 7. The particular operation (treatment) performed when the device becomes ready depends on the device itself. Cancellation (making R=0) must always be enforced so that the polling program can detect R becoming 1 again.

► Write a program *ascii.asm* that repeatedly reads the keys that are typed on the keyboard and then writes their corresponding ASCII codes on the console. The program will finish when a chosen key is typed (return, point, etc.) The processing will be:

1. Read the data register from the keyboard interface.
2. Write on the console the value read by using the `print_int` system call.

The program pseudocode is:

```
Repeat
    wait until keyboard read (bit R = 1)
    Processing:
    read the keyboard data register
    write on the console the value read
until read_caracter == chosen_character
```

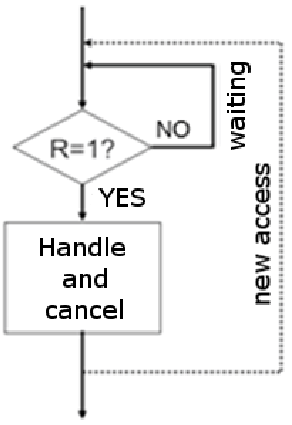


Figure 7. Common diagram for peripheral management by polling

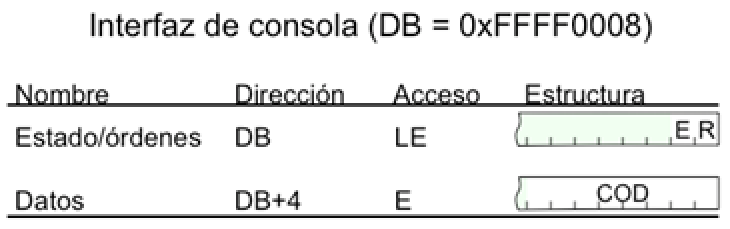
Since you can reuse many lines of the code in *wait.asm*, you'll save time by copying *wait.asm* into a file *ascii.asm* and then modify it. But do it with caution: Copy&Paste is a known source of programming typos.

► ***Question 3.*** Copy here the lines of code in charge of synchronization and reading the data register.

3. The console interface

Figure 8 shows the PCSpim console interface. A character is printed in the console when the program writes the character code in the data register of the console interface. Compared to the keyboard interface, you will find the following differences

- The base address is now `0xFFFF0008`.
- The data register is write-only, as dictated by the *output* nature of this device.


Interfaz de consola (DB = 0xFFFF0008)

Nombre	Dirección	Acceso	Estructura
Estado/órdenes	DB	LE	E R
Datos	DB+4	E	COD

Control and state register (Read/Write. Address = Base)

- R (bit 0, read only). Ready bit: R becomes 1 when the console is ready. R becomes 0 when a new code is written to the Data register, and goes back to 1 when the output is completed by the console.
- E: (bit 1, read/write). Enable interrupt (while E = 1, R = 1 activates the device interrupt line)

Data register (Read only. Address = Base + 4)

- COD (bits 7...0). ASCII code of the character to be written on the console. Writing in this

Figure 8. PCSpim console interface, in this lab session you must keep bit E = 0

The console ready bit

The console requires a certain amount of time to perform the output operation, along which the ready bit R is 0. Therefore, before writing a new character it is necessary to wait for R = 1. This is the synchronization requirement of this device.

Activity 3. Keyboard and console basic functions

We will now work with both peripherals and give support to two typical I/O functions:

- `void putchar(char c);` writes a character on the console
- `char getchar();` reads a character from the keyboard

These functions exist with the same name in the standard input/output C library `<stdio.h>`, and there are equivalent methods in Java like `TextIO.put(char c)` and `TextIO.getAnyChar()`. PCSpim offers them also as `read_char` and `print_char` (system functions 12 and 11, respectively – see appendix).

► Open *eco.asm* in a text editor and observe its structure. From the label `__start` you will find the main program. It first uses the `putchar` function to write the string "P12\n" in the console. Then comes a loop that repeatedly reads characters from the keyboard (by calling `getchar`) and writes them on the console (by calling `putchar`). Notice that the program will end when it reads the escape key (ASCII code 27).

► Complete the code of functions `getchar` and `putchar` after the corresponding labels. **Do it following the common convention for functions but without using PCSpim system calls**. That is:

- `getchar` has to synchronize with the keyboard by polling, reading the character from the data register and returning it in *$v0*.
- `putchar` has to synchronize with the console by polling, and write the contents of *$a0* to the data register.

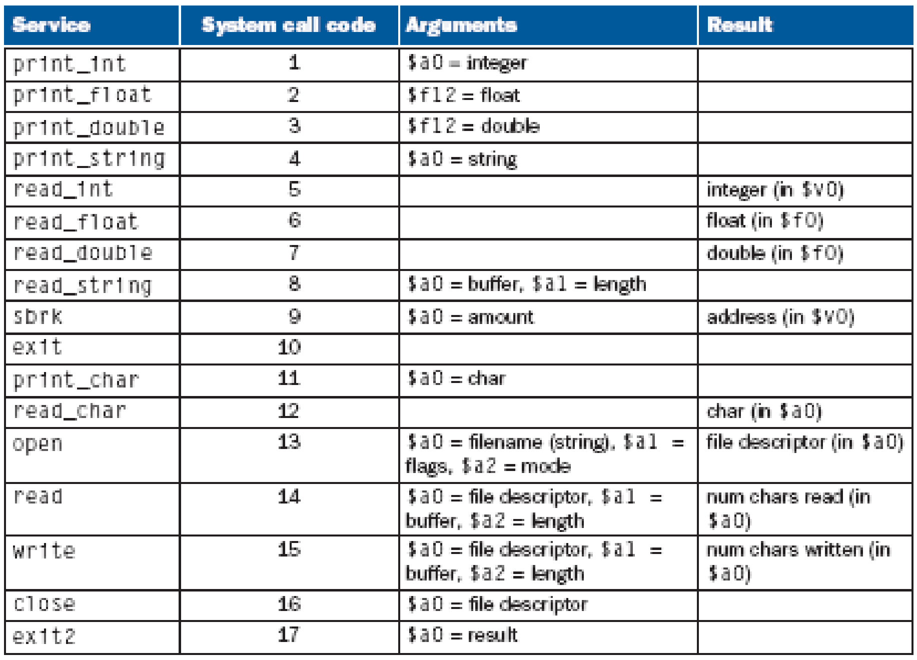
► ***Question 4.*** Write here the code for getchar and putchar.

► Run *eco.asm* (*Simulator* > *Run* or [F5]) and stop it with *Simulator* > *Break* while it is waiting for typing a key, after displaying text "P12" in the console. Then inspect the PC value and identify the instruction it is pointing to.

Instruction:	PC value (hex):

► ***Question 5.*** Can you explain why the program has stopped at that point?

Appendix. PCSpim system functions



Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	