ESTRUCTURA DE COMPUTADORES

Lab 12: Synchronisation by Polling

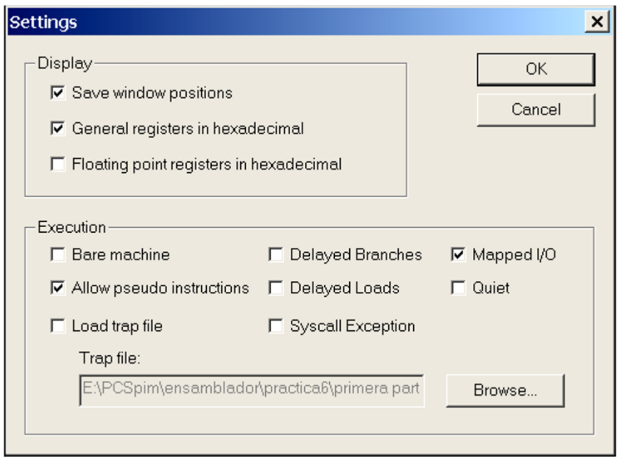
Goal

To develop programs that perform polling synchronisation with (emulated) peripherals.

Materials

Starting from this lab session, we will be using the **PCSpim-ES** simulator, which extends the original PCSPim to emulate various peripherals (a console and a keyboard, in this lab). The PCSpim-ES executable and the source files *wait.asm* and *eco.asm* are all available from the lab session folder in PoliformaT.

When you start using the simulator, open the window *Help > AboutPCSpim* and make sure that it shows "*PCSpim Version 1.0 - adaptación para las asignaturas ETC2 y EC, etc...*". You need to enforce the simulator configuration shown in Figure 1 (*Simulator > Settings*). Note that there is a new check box, *Syscall Exception*, that must remain unchecked for the moment.


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

Input/Ouput in PCSpim

Usually, user programs do not access adapter registers directly with memory instructions (*load* and *store*). Actually, operating systems *prevent* them from doing it, for many reasons. Instead, regular user programs must rely on system functions to perform input/output with peripherals. This approach ensures safety and efficiency since system software is optimized for those purposes. System functions are the ones that ultimately access adapter registers.

The PCSpim environment, however, has important differences with respect to actual operating systems. In the PCSpim model:

- Besides using system functions, user programs are allowed to access peripherals directly with memory instructions. Figure 2 shows the two possible ways in which user programs can access peripherals in PCSpim.
- Predefined I/O functions are simple, and they are not prepared to support concurrency. The available system functions are listed in the Appendix section at the end of this document.

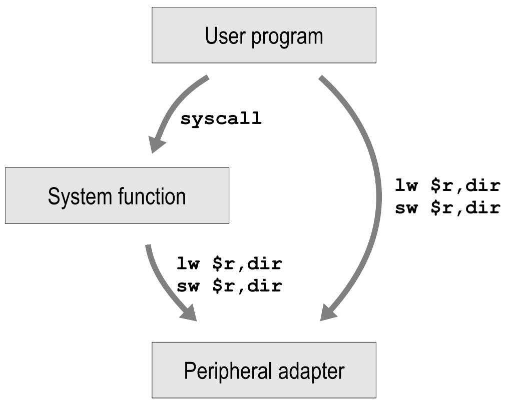


Figure 2. Two ways to access a peripheral adapter in PCSpim-ES

For the purposes of this lab session, there are two emulated peripherals available: keyboard and console (see Figure 3). Both are known from previous lab sessions, where we used them via system functions such as `read_string` or `print_char`. In PCSPim-ES, there is a simulated adapter for each peripheral that creates an interface that we will be using directly.

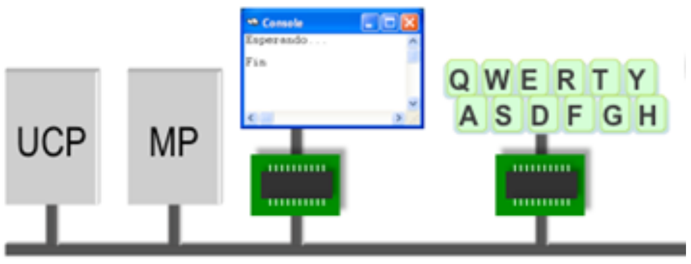


Figure 3. Complete PCSpim diagram including the keyboard and the console

1. The keyboard interface

The keyboard is an input device with the basic purpose of reading a **character** whenever a key is pressed. When this happens, the corresponding code for the pressed key can then be retrieved from a data register in the keyboard adapter. There are two registers in the adapter (see details in Figure 4). Both are 32-bit wide, but all meaningful bits are located in their least significant byte. The base address of the adapter is `0xFFFF0000`.

Note that bit E in the Status/Control register will be used in future lab sessions. For this lab session, it must be kept at value E = 0. In other words, we will only be using the ready bit (R) of the Status/Control register in this lab.

The keyboard ready bit

Every time a key is pressed, the keyboard hardware sets the Ready bit to R = 1. In this lab session, you will develop programs that detect this event by *polling* the value of the Ready bit, to determine *when* a new character code can be read from the keyboard Data register. The Ready bit is reset implicitly by the hardware every time a program reads the Data register.

Keyboard adapter
Base Address: BA = 0xFFFF0000

Register	Address	Access	Structure
Status/Control	BA	R/W	E R
Data	BA+4	R	COD

Status/Control register (Read/Write. Address = Base)

- R (bit 0, read only). Ready bit: R = 1 every time a key is pressed. Reading the data register resets the bit to R = 0.
- 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.

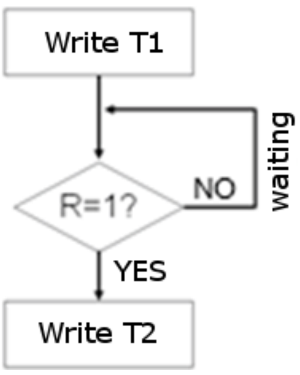
Figure 4. Keyboard interface description. In this lab session, bit E must be kept at value E = 0.

Activity 1. Understanding the polling loop

Using a text editor, open the file *wait.asm* and inspect the code. Identify the three main parts of this program: first it prints the string T1 on the console, using the system function print_string; then comes the polling loop, to wait for the Ready bit to be set; finally, it prints the string T2 on the console, once a key has been pressed, and then it exits.

Figure 5 shows the program structure and the details of the polling loop. Pay attention to the following details about this polling loop:

- The adapter base address `0xFFFF0000` is loaded to `$t0`.
- The interface Control/Status register is read (offset 0 from the base address).
- All bits except R are cleared using a bit mask.
- The loop only ends when R=1.



```
# Wait for key pressed

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

Figure 5. Diagram and polling loop code in *wait.asm*

➢ Open the file *wait.asm* in PCSpim-ES and use *Simulator* > *Go* to execute it. You should see the string T1 printed on the console and then, after a key press, string T2.

- **Question 1**. If we changed the keyboard adapter so that the Ready bit takes position 5 in the Control/Status register, how would you modify this code?

- Use *Simulator* > *Reload* to execute the program again. This second time, you will see T1 immediately followed by T2, with no waiting for a key press in between. The reason is that the ready bit is still keeping the value R = 1 from the previous execution, because wait.asm does not clear it. We tackle cancellation of the Ready bit in the next activity.

Activity 2. Device cancellation

Cancellation is the action of resetting the Ready bit, thus preparing the peripheral for a new operation. In the case of the keyboard, to detect a new key press event. In the keyboard adapter, cancellation occurs when the Data register is read.

- **Question 2.** Modify *wait.asm* to read the Data register to *$t2* after a key press (see Fig. 6).

- Try running the modified program several times without closing the simulator. Note that now the program always waits for a key press before displaying T2, because the ready bit R is conveniently cleared at the end of each execution.

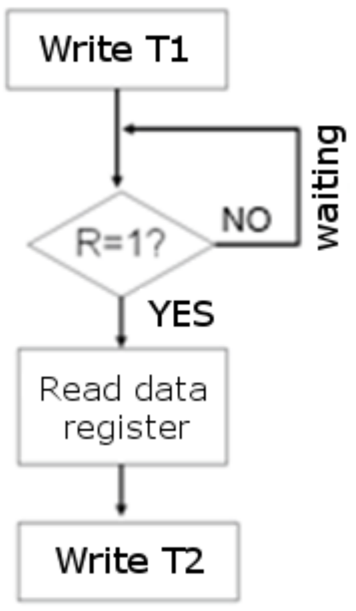


Figure 6. Diagram of a modified *wait.asm*

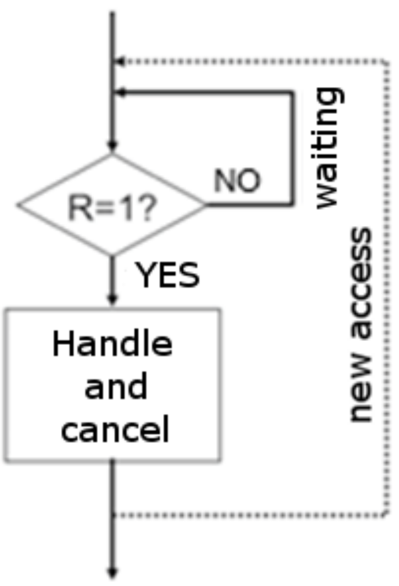


Figure 7. Diagram of polling and handling

2. Synchronisation by polling

The technique applied in *wait.asm* (repeatedly reading the state until the device is ready) is called *polling synchronisation*. The general scheme for polling synchronization is shown in Fig. 7. The particular operation (or ***handling***) performed when the device becomes ready depends on the device itself. Care must be taken to always cancel the ready bit so that the polling program can detect R becoming 1 again. We will now develop code for handling keyboard events in particular.

➢ Write a program that writes to the console the ASCII code of keys as they are pressed in the keyboard. The program must terminate when a key of your choice is pressed (return, point, etc.). The handling code must:

1. Read the data register from the keyboard interface. This effectively clears the R bit.
2. Print the read code to the console using the `print_int` system function.

The pseudocode for this program is:

```
Repeat
  Synchronisation:  wait until key pressed (bit R = 1)
         Handling:  read the keyboard data register
                    print the read code to the console using print_int
until read_code == your_chosen_key_code
```

Create a new file *ascii.asm* to write this program. You can reuse most of the code in *wait.asm* so, rather than starting from scratch, you can copy and paste *wait.asm* into *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 code in charge of synchronisation and reading the data register.

3. The console interface

In the previous exercise, we used the console via the system function print_int. Recall from Fig. 2 that the console adapter can also be accessed directly in PCSPim, and that's how we will use it this final part of the lab. Figure 8 gives the details of the console adapter. A character is printed to the console when the program writes the character code in the data register. Compared to the keyboard adapter, 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.

The console ready bit

The console requires a certain amount of time to perform an output operation (i.e., print a character). During that time, bit R is 0. Before trying to print a new, you need to wait for R = 1. This is the synchronisation requirement of this device.

Console adapter

Base Address: BA = 0xFFFF0008

Register	Address	Access	Structure
Status/Control	BA	R/W	E R
Data	BA+4	W	COD

Control/Status register (Read/Write. Address = Base)

- R (bit 0, read only). Ready bit: R = 1 when the console is prepared to print a character. R is cleared when a code is written to the Data register and it goes back to 1 when the output is completed.
- 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 character to print. Writing to this register makes R = 0.

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

Activity 3. Keyboard and console basic functions

We will now work directly with both peripherals, keyboard and console, to implement two common I/O functions:

- `void putchar(char c);` writes a character to 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). You will now write your own version, directly accessing the adapters of these peripherals.

➢ 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 (calling `getchar`) and writes them on the console (calling `putchar`). Note that the program ends when it reads the escape key (ASCII code 27).

➢ Complete the code of functions `getchar` and `putchar` in *eco.asm*. Note that you have to implement these functions directly using the adapter, not the available system functions 11 and 12. Follow the usual convention:

- `getchar` has to synchronise with the keyboard by polling, read the character from the data register and return the character code in *$v0.*
- `putchar` has to synchronise with the console by polling and write the contents of *$a0* to the data register to have it printed in the console.

➢ **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 a key press, 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.*** Explain why the program has stopped at that particular 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	