Lab 12: Polling synchronisation

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 *echo.asm* are all available from the lab session folder in PoliformaT.

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

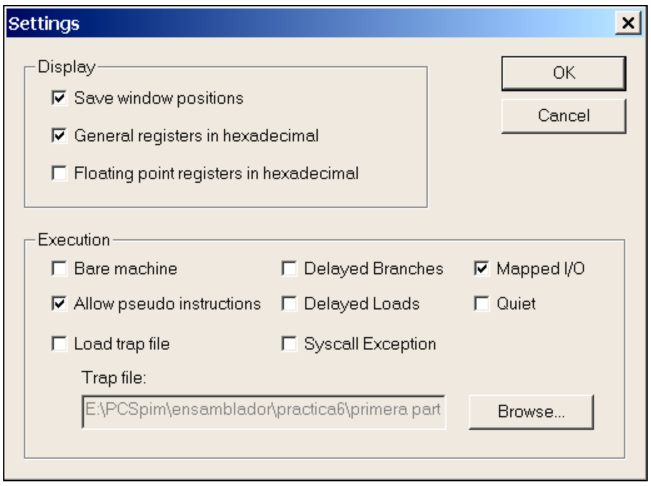


Figure 1. Simulator configuration settings for this lab session.

Input/Output in PCSpim

In real-life computers, user programs cannot access adapter registers directly. Operating systems *prevent* them from doing so, for good reasons. Instead, user programs must rely on system functions to perform all input/output operations. This approach ensures safety and efficiency since system software is optimised to properly access the peripherals, even when multiple user programs run concurrently. Hence, in most real computers, system functions are the ones that ultimately access adapter registers.

The PCSpim-ES environment, however, has important differences from actual operating systems, which is not strange considering it is an educational tool:

- Besides using system functions, user programs in PCSpim can access peripherals with memory instructions (*load*/*store*) to read or write adapter registers (see Figure 2).
- The predefined system functions (see Appendix) are not prepared to support concurrency. There can only be one user process under execution.

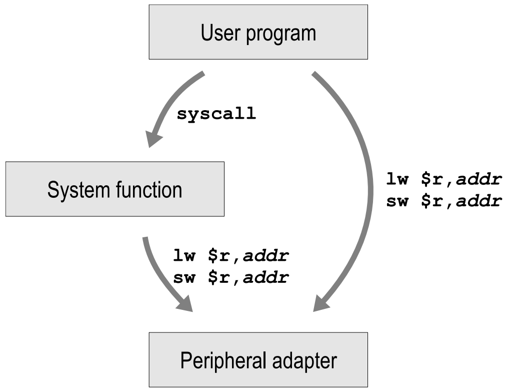


Figure 2. Two possible ways to access peripheral adapters in PCSpim-ES: using system functions (left) or directly accessing the peripheral's adapter with load/store instructions (right).

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

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

1. The keyboard adapter

The keyboard is a device with the basic purpose of providing user inputs to a program. Whenever a key is pressed in the keyboard, the ASCII code of the corresponding **character** is made available in a dedicated data register in the keyboard adapter. There are two registers in the adapter, whose description is given in Figure 4. Both are 32-bit wide, but all meaningful bits lay on their least significant byte. The base address of the adapter is `0xFFFF0000`.

Note that bit **E** (interrupt Enable) in the Status/Control register **must be kept at value E = 0 in this lab session**. It will be used in future lab sessions for interrupt synchronisation. We will only be using the ready bit (R) of this register in this lab.

The keyboard's 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* this bit to determine *when* a new character code can be read from the keyboard Data register. There is no cancellation bit in this keyboard: the Ready bit is reset by the hardware every time a program reads the Data register (i.e., the keyboard uses *implicit cancellation*).

Keyboard adapter
Base Address: BA = 0xFFFF0000

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

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

- R (bit 0, read only). Ready bit: R = 1 every time a key is pressed. Reading the data register at BA+4 resets the bit to R = 0.
- E: (bit 1, read/write). Enable interrupt. When 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.

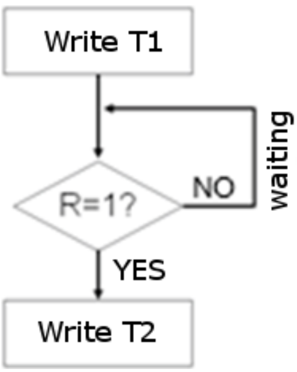
Figure 4. Registers of the Keyboard adapter. In this lab session, bit E must be kept at value E = 0.

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
```

- 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?

```
andi $t1, $t1, 32
```

- 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 keeps the value R = 1 from the previous execution because *wait.asm* does not clear it. We will tackle the 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 keyboard to detect a new key press event. In the keyboard adapter, cancellation is implicit, so there is no cancellation bit in the adapter. Instead, the Ready bit is cleared automatically when the Data register is read.

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

```
lw $t2, 4($t0)
```

- 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.

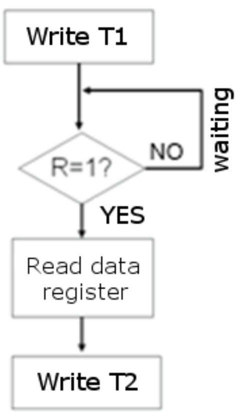


Figure 6. Diagram of a modified *wait.asm*

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.

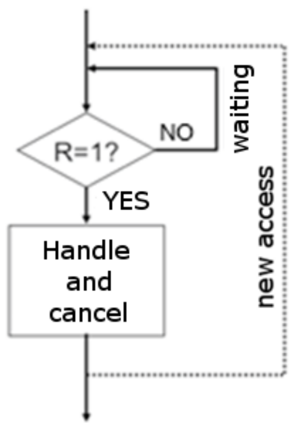


Figure 7. Peripheral handling using polling synchronization.

- 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 lines of code in charge of synchronisation and reading the data register.

```
getchar:      # $v0 = getchar()
  li $t2, 0xffff0000

  wait:  # Wait for bit R == 1
    lw $t1, 0($t2)
    andi $t1, $t1, 1
    beqz $t1, wait

  lw $v0, 4($t2)
  jr $ra      # return from getchar
```

3. The console adapter

In the previous exercise, we have 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 in this part of the lab. Figure 8 shows the details of the console adapter. When the console is ready, a character can be printed by writing its ASCII 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, which reflects the *output* nature 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 ready. R becomes 0 when a 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 (when E = 1, R = 1 activates the device interrupt line)

Data register (Write 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.

The console Ready bit

The console requires a certain amount of time to perform an output operation. During that time, bit R is 0. Before writing a new character code to the console Data register, you need to wait for R = 1. This is the synchronisation requirement of this device.

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 also offers them as system functions `read_char` and `print_char` (12 and 11, respectively – see appendix). You will now write your own versions of these functions, directly accessing the adapters of these peripherals.

- Open the file *echo.asm* in a text editor and observe its structure. Find the main program after the label `__start`. 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`). The program ends when it reads the *escape* key (ASCII code 27).
- Complete the code of functions `getchar` and `putchar` in *echo.asm*. You must implement these functions directly using the adapter, not the system functions 11 and 12. Follow the usual convention:
  - `getchar` must synchronise with the keyboard by polling, then read the character from the adapter's Data register and return the character code in `$v0`.
  - `putchar` must synchronise with the console by polling and then write the contents of `$a0` to the data register to have it printed in the console.
- **Question 4**. Write here the code for getchar and putchar.

```
getchar:      # $v0 = getchar()
   li $t2, 0xffff0000

   wait:   # Wait for bit R == 1
      lw $t1, 0($t2)
      andi $t1, $t1, 1
      beqz $t1, wait

   lw $v0, 4($t2)
   jr $ra      # return from getchar
```

```
putchar:      # putchar($a0)
   ### TO BE COMPLETED
   li $t2, 0xffff0008

   wait2:
      lw $t1, 0($t2)
      andi $t1, $t1, 1
      beqz $t1, wait2

   sw $a0, 4($t2)
   jr $ra      # return from putchar
```

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

Instruction: andi $9, $9, 1	PC value (hex): 0x0040050

- ***Question 5.*** Explain why the program has stopped at that point.

It's an instruction inside the polling loop. The program stopped there because it was running that instruction inside the loop, waiting until the ready bit becomes 1.

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	