Lab 13: Synchronization by interrupts

1. Introduction and objectives

The PCSpim simulator includes, in addition of I/O devices, a part of the registers and instructions that MIPS has to handle exceptions (see the annex at the end of this document), you can use it to perform input and output operations synchronized by interrupts.

Our goal is to manage two I/O devices by interruptions. For that, you will need to design the **initialization code** and the **interrupt system handler**.

This guide is organized in 7 progressive steps, with the intention of solving in every step a concrete design problem among the following ones:

- In the initialization code: enable interruptions globally and enable keyboard and clock interrupts selectively.
- In the interrupt handler: context preservation of the interrupted program, and returning to the appropriate point to resume its execution.
- In the interrupt handler: implementation of a particular keyboard and clock interrupt handlers.
- In the interrupt handler: the design of the decision tree that identifies every interrupt, and chooses the appropriate handler routine.

Tools

- PCSpim-ES simulator version.
- Source code files *nothing.asm*, *nothing.handler* and *loops.asm*.
- Appendix files: (1) system calls, (2) MIPS coprocessor and (3) PCSpim I/O devices.

2. Exceptions on PCSpim simulator

There are several exception sources (summarized on file “Appendix 2. Coprocessor”) on real MIPS processor. With the PCSpim Simulator that will be used, only two of them can be handled: peripheral interruptions and system calls performed with syscall. The appropriate simulator setting are shown in Figure 1.

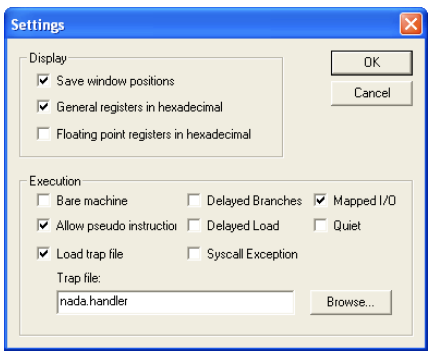

Figure 1. Configuration settings for interrupt handling. The handler is *nothing.handler*

Task 1. System observation with the simulator.

Along this lab session, you will work in parallel with two files:

- The *Trap file*, with extension *.handler*, that defines the *.kdata* and *.ktext* interrupt handler segments and a fragment of *.text* for starting and ending as required.
- The user program with extensions *.s* or *.asm*, describes the remaining *.data* and *.text*. segments

Every time you open (*File>Open*) or reload (*Simulator>Reload*) a user program file the simulator also load the interrupt handler specified as *Trap File* inside the simulator setting windows shown in figure 1 (*Simulator>Settings...*).

The combination of both files form a **system** composed by a user program, an interrupt handler and a starting code. Figure 2 shows the whole system.

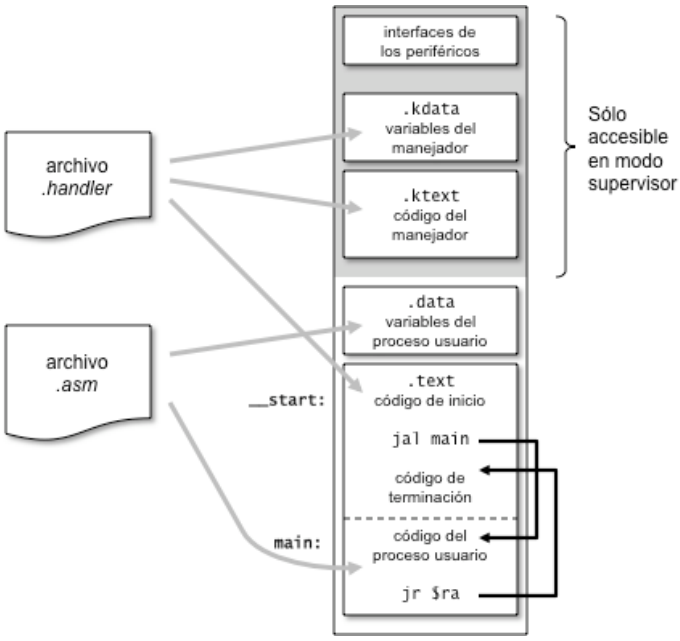

Figure 2. The two source code files and their contribution to system segments

► Look at figure 3 or open with a text editor the files *nothing.handler* and *nothing.asm* in order to check the following issues:

- File *nothing.handler* describes the content of segments `.kdata`, `.ktext` and `.text`.
- The execution starting point (tag `__start`) is located at segment `.text` on file *nothing.handler*. So, after that tag it goes the starting code (on *nothing.handler* this area is empty).
- On *nothing.handler* the initialization is followed by a jump to tag `main`. The entry point to the user program is marked by this tag on file *nothing.asm*.
- The user program must end with instruction `jr $ra`, to get the control back to the ending code located on file *nothing.handler*
- The ending code is a call to `exit`.

- The handler code is apparently outside the system execution flow. It will be executed only when an interrupt will happen.

Figure 3. System execution flow

►Select *nothing.handler* as *Trap File* and load *nothing.asm*. In the simulator window look for code, variables and messages, the elements marked in Figure 4 and enumerated next:

- In the state window: Exception coprocessor registers.
- In the code window: Segment content for code `.text` and `.ktext`. separated by the mark `KERNEL`. Notice that the initialization code is followed by the user program because both codes are allocated in the same code segment `.text`.
- In the variable window: the variables from the user data segment are followed by the stack (marked as `STACK`) and the handler variables (`KERNEL DATA`). Notice that *nothing.handler* defines a variable `var1` in segment `.kdata` which initial value is `0xFFFFFFFF`, while the user program defines `var2` in segment `.data` with content `0x77777777`; in the variable window they appear each one in its segment.
- In the message window: when a.*asm* file is loaded or reloaded it is shown what *.handler* has been loaded. If the code contains errors, it is indicated at the loading time.

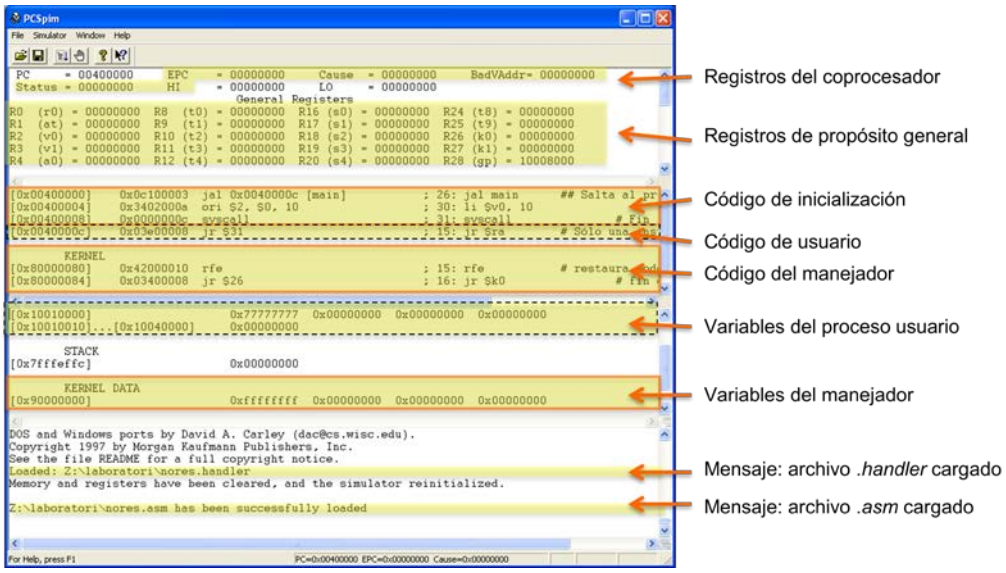

Figure 4. The simulator allows the inspection of the exception coprocessor registers, the handler code and variables together with the user ones

3. Interrupts handling.

PCSpim has three interrupt sources, corresponding to the three peripherals in the simulator, two of them studied in the previous lab, now we add the clock. The interfaces description is shown on file "Appendix 3. PCSPIM peripherals". Now each peripheral is connected to an interrupt line, as shown in Figure 5.

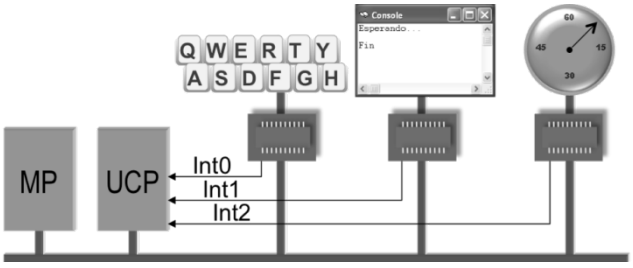

Figure 5. Connection of keyboard, console and clock to the MIPS interrupt lines. The console interrupt line (Int1) will not be used in this lab session.

Task 2. Preparing a test user program.

► Open file *loops.asm* with a text editor. Look at its structure: it is made of two nested loops designed just to get 5 to 10 execution seconds.

► Load *loops.asm* into the simulator (you can keep file *nothing.handler* as *Trap File*) and then execute it. If it runs too fast or too slow, set the initial value that controls the number of executions of the outer loop (instruction: `li $t0, 10`). **Don't forget that you can always stop program execution pressing *control-C.***

Task 3. System initialization.

In order to get processed an interrupts coming from a peripheral through a given interrupt line the following conditions must be met:

- Interrupt enabled in the peripheral interface (bit E = 1)
- The interrupt line is unmasked in the exception coprocessor state register

- Interrupts are enabled in the processor state register

In general, system startup is done in the following three steps sequence:

1. Configuring the available peripherals, enabling or disabling every interrupt in their interface control registers.
2. Configuring the exception coprocessor state register: interrupt mask and processor execution mode.
3. Transferring control to the user program.

Step 1: Enabling interrupts and setting execution mode

► With the text editor: Change *nothing.handler* and save the file with name *keyboard.handler*. You have to append the appropriate starting code (before `jal main`) so that the handler **only** will process the keyboard interrupt.

The initialization requires two operations:

- Enabling interrupts on the keyboard state/control register, and
- Writing the proper value on the coprocessor state word so the interrupt line *Int0** goes unmasked, the interrupts globally enabled and the user mode set. You will have to use the instruction `mtc0`.

► With the simulator: configure *keyboard.handler* as *Trap File* in *Simulator>Settings...*. Load and execute *loops.asm*. If the keyboard interrupt is enabled and unmasked correctly, the simulator will start giving messages like "`Bad address in data/stack read`" as soon as a key is pressed. Otherwise, the simulation will end normally.

► **Question 1.** Write the initialization code.

► **Question 2**. When are error messages appearing? Clue: Look at the program returning code that is located at the end of the handler (there is a comment `## Return to program`). Is the return to program code correct?

Task 4. Understanding the handler structure

We are going to write, step by step, the complete code for a basic interrupt handler. The available handler *nothing.handler* only contains comments marking the areas to write executable code, that will be developed along the lab session. For now, **the only interrupt source will be the keyboard**.

Step 2: Getting the return address to the user program

► With the text editor: Add the instruction that writes into *$k0* the return address to the handler end.

► **Question 3.** Write the instruction line append to the program.

► With the simulator: Load and execute again the user program *loops.asm*. If everything is fine, the program will end correctly even if a key is pressed.

Step 3: Temporary handling of the keyboard interrupt

The following **incorrect handling** will be applied, it writes an asterisk ("*") in the console every time an exception happens:

```
li $v0,11
li $a0, '*'
syscall
```

► With the text editor: Write the handling code on the interrupt handler, where indicated by "`identify and handle exceptions`").

► With the simulator: Load and execute the system, then press one key. If everything is fine asterisks will appear without stopping.

► **Question 4.** Why so many asterisks appear when pressing a key?

Step 4: Cancelling interrupt

The former incorrect behavior will be corrected by cancelling the keyboard interrupt during its handling.

► With the text editor: Write the code that will cancel the keyboard interrupt, something like this:

```
li $t0, base address for keyboard interface
lw $a0, keyboard data register on address ($t0 + 4)
```

► **Question 5.** Write the instruction lines added to cancel the keyboard interrupt.

► With the simulator: Load an execute the system. The user program will finish as soon as one key is pressed.

► **Question 6.** Why the user program ends before expected when a key is pressed? **Clue: The handler changes register *$t0* that is used inside the user program *loops.asm***.

Step 5: Managing the context

The handler has to save the user program execution context that has been interrupted in order to avoid interfering with it. By context it is meant: CPU register bank, program counter (PC) and any additional data inside the processor related to the program being executed. In this case, it's enough saving in the handler the CPU registers it is going to use, before any other action. We will call this operation *saving the context*. Then, just before returning to the interrupted program, the saved registers will be *restored*.

Figure 6. Handler structure including context management

► Annotate what registers uses the handler. At first sight they are: *$t0*, *$a0* and *$v0*, but register *$at* have to be included because it is implicitly used by `li` or `sw`. Even though program *loops.asm* uses few registers, this is a general issue: in order to allow correct execution of any program running with *keyboard.handler*, the handler must preserve the four registers that it uses.

> *An important point: we are going to forbid user programs to use registers $k0 and $k1. We will use $k1 as a pointer to the preserved context, as shown in figure 7. Register $k0 is already reserved for a short but very important role: it contains the returning address to the user program to allow instruction `jr $k0` (the last one on the handler) working correctly.*

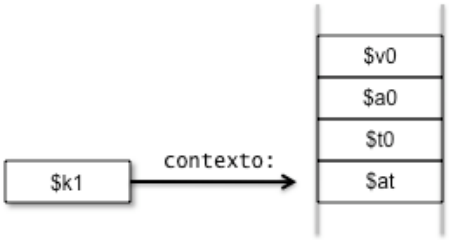

Figure 7. Register *$k1* points permanently to the memory area where the context is saved

► The code to save the context has to be added. There is an additional problem that we have to deal with: usually, the assembler considers register *$at* as reserved and it doesn't allows using it in the instructions. If we write the instruction:

```
add    $t1, $t0, $at
```

the assembler with give a message "`Register 1 is reserved for assembler on line` *xx* `of file` *yy*". An instruction like that has to be surrounded by two `.set` directives:

```
.set   noat
add    $t1, $t0, $at
.set   at
```

► **Question 7.** With the text editor, open *keyboard.handler*:

- On the handler data segment: Declare a variable named `context` with 4 words capacity.

- In the system starting code: Write into *$k1* the address of variable `context`

- At the beginning of the handler code, where it says “save the user program context”, write the instructions that save the four register former mentioned into `context`, as shown in figure 7.

- At the end of the handler code, where it says “restore the context”, write the code that reads the context content and copies it back to the registers.

► With the simulator: Check if the system works.

Summary

We have written the code required to make the system able to deal with only one interrupt source: the keyboard.

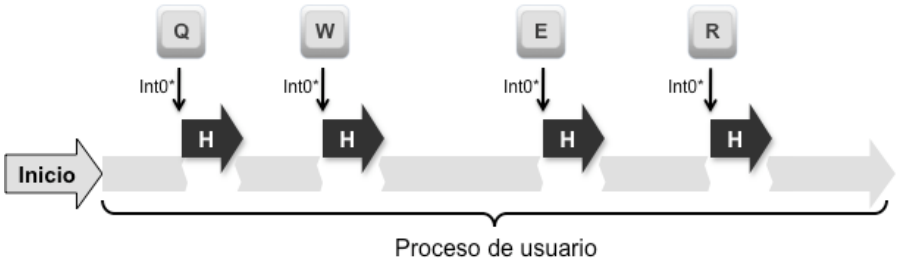

Figure 8. Handler *H* is executed every time the keyboard is ready.

Task 5. Handling several interrupts

Every time a new interrupt source is added, and generally when a new exception cause has to be handled, it is required that:

- On the handler data segment: Defining required **variables** to do the handling.
- On the handler code: Writing (1) the instructions that **identify** the new interrupt and (2) the corresponding **handling**.
- On the system starting code: **Initializing** the peripheral and the variables required to handle it.

Because up to now there was only one interrupt there no interrupt discrimination code.

Step 6: Enabling clock interrupt

The system clock handling has to be added. In this step the clock interrupt has to be enabled without changing the handler.

► **Question 8.** With the text editor open *keyboard.handler* and save it as *keyboard_and_clock.handler*.

- In the starting code add the instructions that enable the clock interrupt.

- Change the coprocessor state register to set the interrupt line *int2** unmasked.

► With the simulator: Check the system. If the clock interrupt has been enabled properly the console will show asterisks without stopping.

► **Question 9.** Explain why the system behaves as described previously. Clue: Is the clock interrupt cancelled somewhere on the handler?

Step 7: Analyzing the exception cause

It is clear that every interrupt (and in general every exception cause) requires an specific handler. In this step, an small decision tree is going to be added in order to associate every interrupt to its handler avoiding another exception cause to produce an inadequate response. The goal is to achieve the execution flow inside the exception handler shown in figure 9:

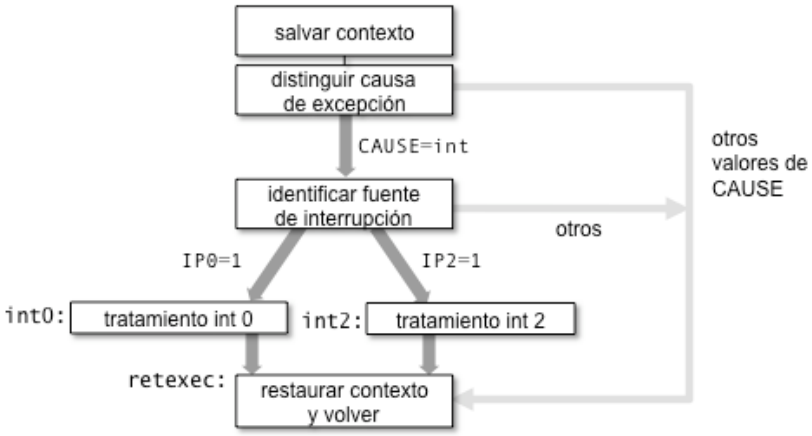

Figure 9. Exception handler execution flow handling two interrupts

► **Question 10**. With the text editor modify *keyboard_and_clock.handler* file.

- Add a new tag (in figure 9 appears as `retexc`) that will mark the point on the handler that restores the context and gives the control back to the interrupted program. From now, jumping to this tag inside the handler will be the same as *ending*.

- Add a new tag `int0` to apply the handling that from steps 4 and 5 are applied to the keyboard interrupt. At the end of the interrupt handling we have to add a jump to `retexc`.

- Add a tag `int2` followed by the clock interrupt handling that will be just cancelling the interrupt on the interface avoiding to unable the interrupt. Don't forget jumping to `retexc`.

- Add the instructions that will read and analyse the exception cause word from the exception coprocessor (the cause word explanation is on file "Appendix 2. Coprocessor") in the section marked with "`## Identify and handle exceptions`". If the cause is an interrupt the analysis will continue, otherwise a jump will be done to `retexc`.

- Once it has been identified that the cause of a given exception is an interrupt, what remains is to add the instructions that will analyse bits *IP0* and *IP2* from the state word in the exception coprocessor followed by a jump to tags `int0` e `int2`., respectively. If none of these bits is active then jump to `retexc`.

►Take care all the time over the registers used by the handler. In step 5 we have written the code that saves and restores registers *$at, $t0, $v0 and $a0*. You can restrict yourself to use only those four registers on the handler, or you can use other registers like (like *$t1*) if so you have to get aware that the context variable size should be increased to get space for those extra registers.

► With the simulator, check the system. If everything is fine, it will appear on the console one and only one "`*`" every time a key is pressed; the user program will smoothly end when the time comes.

ANNEX

Assembly for exceptions

New segments

`.kdata` Describes data located in the restricted memory areas (addresses from 80000000_h to $FFFFFFFF_h$). This is accessible only in kernel mode, so user programs are not allowed to read or write there. This is the directive required to define variables on the interrupt handler.

`.ktext` Describes instructions located inside the memory restricted area. They can only be executed in kernel mode, so user programs cannot jump there. This is the required directive to write the interrupt handler code.

New instructions

`mfc0` *rt, rs* Transfers the coprocessor register *rs* content to the general purpose register *rt*.

`mtc0` *rt, rs* Transfers the general purpose register *rt* content to the exception coprocessor register *rs*.

`rfe` Performs a two position right shift of execution mode and interrupt enable bits in the coprocessor state register, unmaking the effect of the last exception handling. Usually, this instruction does the going back to user mode at the end of the exception handler execution.