Lab 13: Interrupt Synchronisation

1. Introduction and goals

In addition to I/O devices, PCSpim also simulates some of the registers and instructions related to exception handling (see the Annex at the end of this document for a summary). In this session, you will use part of these resources to synchronise with I/O devices using interrupt synchronisation. To that end, you will need to implement the **system initialisation code** and the **interrupt handlers** themselves.

This session proposes seven progressive steps to achieve this goal and to end up with a functional exception handler to cater for keyboard and clock interrupts.

Tools and materials available

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

2. Exceptions in the PCSpim-ES simulator

A real MIPS processor can handle exceptions from different sources (see accompanying file "Appendix 2. Coprocessor"). The PCSpim simulator allows one to handle just two of them: peripheral interrupts and system calls performed with the syscall instruction. To that end, you must configure the simulator settings as shown in Figure 1.

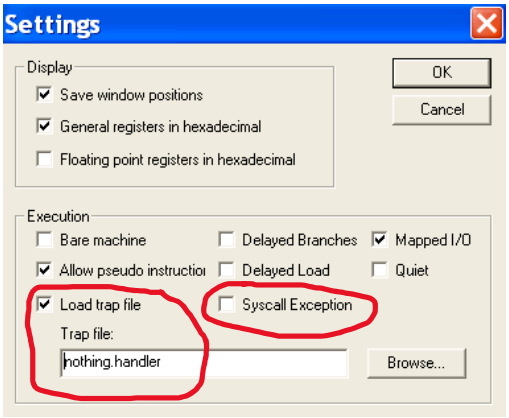

Figure 1. Configuration settings for interrupt handling. Check "*Load trap file*" and use *Browse* to select the handler file *nothing.handler*. Keep "*Syscall Exception*" unchecked.

Task 1. Observing the system with the simulator

In this lab session, you need to load two different files in PCSpim at the same time:

- The *Trap file*, with extension "*.handler"*. This file contains the *kdata* and *ktext* sections (kernel data and kernel code), plus the fragment of the *text* section containing the system initialisation, a call to the user program, and the termination code. This trap file must be selected from the *Simulator>Settings* window (see Figure 1).
- The *User-Program* file, with extension "*.s"* or "*.asm",* contains the user *data* and *text* sections.

With the configuration given in Figure 1, when you open (*File>Open*) or reload (*Simulator>Reload*) a user program file, the simulator will also load the given trap file.

Combining both files forms a **system** composed of a user program, an exception handler, and the system initialisation code. Figure 2 represents the whole system.

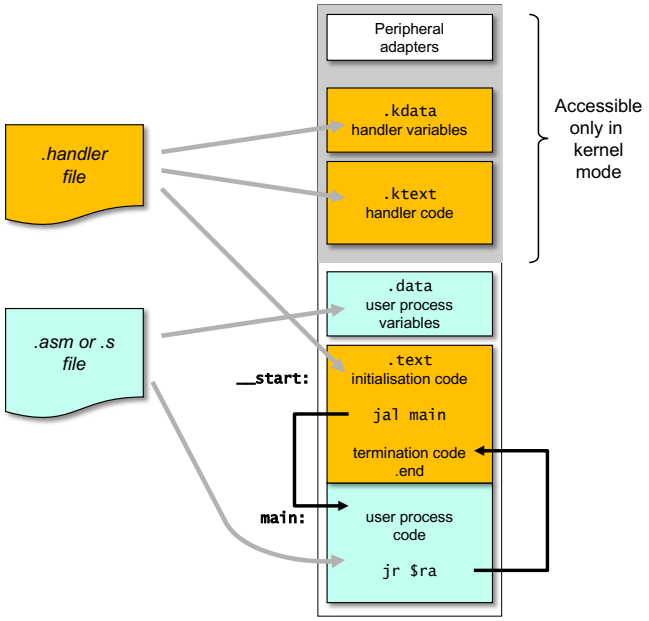

Figure 2. The two source code files and their contribution to system segments

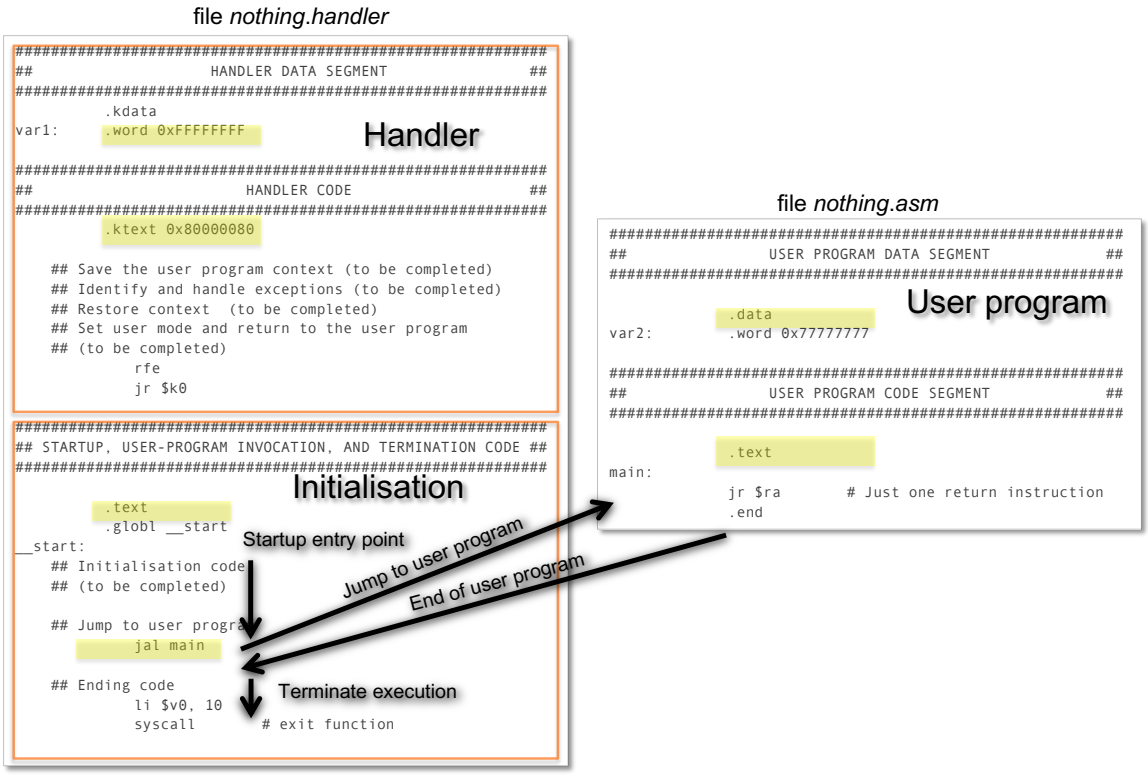

Figure 3. Execution flow of the whole system

► Observe Figure 3 and open the files *nothing.handler* and *nothing.asm* in a text editor for your reference. Check the following aspects:

- The file *nothing.handler* describes the segments *kdata*, *ktext* and a part of *text*.
- At start-up, the PC points to label *__start*. The instruction after this label is the first one to be executed. Note that the initialisation given in *nothing.handler* is incomplete.
- The initialisation code is followed by a jump-and-link to label *main*. This label points to the first instruction of the user program in file *nothing.asm*.
- The user program ends with the instruction jr $ra, which returns to the *ending code* in the text section of *nothing.handler*.
- The ending code is just a call to the system function exit, which terminates the program.
- The handler code in the *ktext* section is apparently out of the execution flow but note that it will be executed whenever an exception occurs.

►Select *nothing.handler* as the *Trap File* (as shown in Figure 1) and load *nothing.asm*. In the simulator window, locate the information highlighted in Figure 4:

- In the state area, the registers of the exception coprocessor (Coprocessor 0).
- The code area contains code sections *text* and *ktext*, separated by the mark KERNEL. Notice that the user program follows the initialisation code because both codes are allocated to the same code segment *text*.
- In the memory area: the variables from the user data segment are followed by the stack (marked as STACK) and then the handler variables (KERNEL DATA). Note that *nothing.handler* declares a variable `var1` in segment *kdata*, initialised to value `0xFFFFFFFF`, and the user program declares `var2` in segment *data*, with value `0x77777777`; in the variable window, they appear in their respective segment.
- In the bottom panel, messages indicate which *asm* file is loaded and which trap file is in place. Errors in either file are shown here if they exist. Pay attention to this often-neglected area of the simulator window!

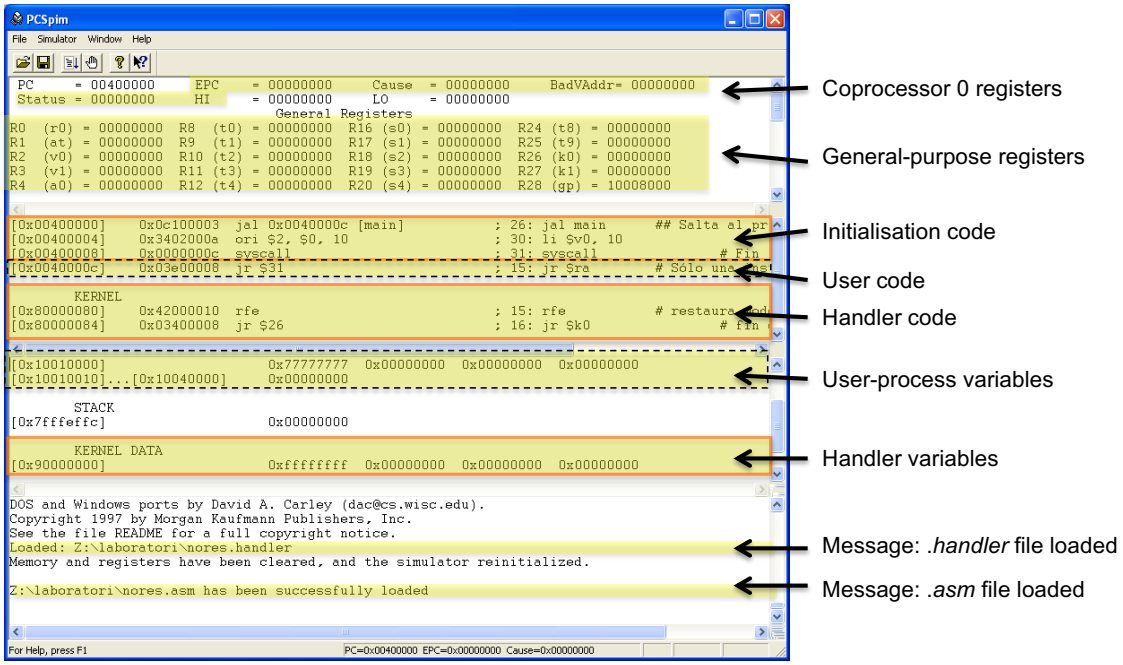

Figure 4. Location of relevant information about coprocessor 0 registers and the code and data of the user program and the kernel in PCSpim.

3. Interrupt handling

PCSpim simulates three interrupt sources corresponding to the three available peripherals. We used two of them in the previous lab session: the keyboard and the console. There is also a clock device. Their interface descriptions are given in the file "Appendix 3. PCSPIM peripherals". These peripherals are connected to the interrupt request lines shown in Figure 5.

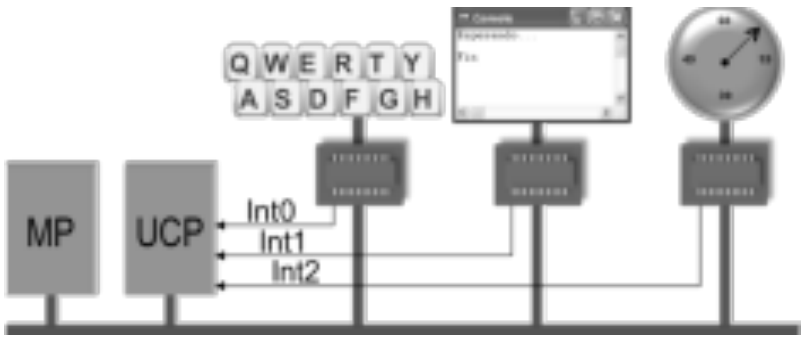

Figure 5. Connection of the keyboard, console and clock to the MIPS interrupt lines. In this lab session, you will be using and handling interrupt requests Int0 and Int2 only, but not console interrupts via Int1.

Task 2. Preparing a user program

The provided file *loops.asm* contains a user program with two nested loops, and its purpose is just to take some 5 to 10 seconds to execute. We will trigger keyboard interrupts during that time and see how the system reacts to them. You may open and inspect this file using a text editor.

► Load *loops.asm* into the simulator (using *nothing.handler* as the *Trap File*) and then run it. If it runs too fast or too slow, you can modify the number of iterations of the outer loop (i.e., the immediate value used in the instruction `li $t0, 10`). Remember that you can always stop a running program by typing *Control-C*, if needed.

Task 3. System initialisation

The purpose of the initialisation code is to enable interrupt requests before jumping to the user program. To enable requests coming from a device that is connected to some Int_*x*, you need to:

- In the peripheral adapter, enable the peripheral to send interrupt requests (set bit E = 1)
- In the $Status register (in coprocessor 0), unmask the interrupt request line (set $IM_x = 1$)
- In the $Status register as well, enable interrupts globally (set $IE_C = 1$)

In general, system start-up follows this sequence:

1. Configure the peripherals, enabling or disabling them to request interrupts using the *Enable* bit in their adapter.
2. Configure the $Status register with the proper interrupt mask and change the processor execution mode from *kernel* (default start-up mode) to *user* mode (set $KU_C = 1$).
3. Transfer control to the user program (in our case, using the instruction "jal main").

Step 1: Enabling interrupts and setting the execution mode

In this step, you will prepare the system to accept keyboard interrupts. In further steps, you will implement the required code to properly handle them.

► With the text editor, save *nothing.handler* with name *keyboard.handler*. Complete the initialisation code (before `jal main`) to enable **only** the keyboard interrupt and set the processor *User* mode before transferring control to the user program (main).

You need to take care of two operations:

- Enable interrupts in the keyboard state/control register, and
- Write the proper value to the $Status register (using `mtc0`) so that `Int0*` goes unmasked, interrupts are globally enabled, and the processor mode is set to *User*.

► With the simulator: configure *keyboard.handler* as the *Trap File* in *Simulator>Settings...* Load and execute *loops.asm*. If your initialisation code is correct, the simulator will start giving messages "Bad address in data/stack read" as soon as a key is pressed (remember the provided *nothing.handler* is incomplete, hence the error messages). If, however, the simulation ends successfully even when you press a key, that indicates that your code is not effectively enabling keyboard interrupts.

► **Question 1.** Copy here the initialisation code.

```
# Set Status IMx bit to 1, and IEc bit to 1 as well
# We also have to set the bit KUc to 1 to go from kernel to user mode
# We therefore have to load 0x103 in mtc0
li $t0, 0x103
mtc0 $t0, $12  # 12 is the Status register

# We now need to enable interruptions in the keyboard peripheral
li $t0, 0xFFFF0000
li $t1, 0x2
sw $t1, 0($t0)
```

► **Question 2**. Which one is the handler instruction causing the Bad address error? Clue: inspect the handler's return instruction after the comment ## Set user mode and return to the user program.

jr $k0
This instruction should be replaced by jr $ra

Task 4. A handler for keyboard interrupts

In the following steps, you will complete the implementation of an exception handler that can properly handle keyboard interrupts. In Task 5, and in future lab sessions, you will be extending the handler to also cater for other exception causes. The provided trap file *nothing.handler* contains comments marking the areas to write the required code.

Step 2: Using the proper return address to the user program

The "Bad address" errors that appeared in Step 1 were because the handler did not write the proper return address in register *$k0*. Hence the handler's return instruction jr $k0 tries to jump to the forbidden address 0 (the value PCSpim gives to uninitialised registers). The fact that the error is recurring has another direct reason that you will fix later.

► Edit the trap file to include the instruction that writes the exception return address to *$k0*. Remember that this address can be found in the register $EPC (*$14*) of coprocessor 0 when the handler starts.

► **Question 3.** Copy here the instruction that writes the exception return address to *$k0*.

mfc0 $k0, $14

► In the simulator, open and run the user program *loops.asm*, using the modified *keyboard.handler* as the trap file. The user program should now end correctly even when a key is pressed.

Step 3: Provisional handling of the keyboard interrupt

Now that the system can properly enter and leave the exception handler, use the following **incorrect-but-helpful provisional handler**[1]. It prints an asterisk to the console every time an exception occurs:

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

► With the text editor, write this handling code in the interrupt handler just after the comment "Identify and handle exceptions".

► With the simulator, open and run the system. While the user program is running, press a key. Now you should see asterisks appearing in the console repeatedly.

► **Question 4.** What is making the handler print asterisks repeatedly upon a single keypress?

Because we are not cancelling the ready bit of the keyboard when we handle the exception

[1] *The handler is incorrect because it should not make a syscall from within the exception handler; but since the option "Syscall Exception" is not checked (see Figure 1) PCSpim emulates syscall without executing the exception handler.*

Step 4: Cancelling an interrupt

You will now fix the handler to avoid the asterisk burst issue. To do this, you need the handler to cancel the keyboard interrupt so that the keyboard Ready bit is reset, which in turn resets the interrupt request and, ultimately, the bit IP_0 in coprocessor 0. From the keyboard description, the cancellation occurs implicitly when the adapter's Data register is read.

► Using a text editor, add the code that cancels the keyboard interrupt. You need to:

- *Load $t0 with the base address of the keyboard adapter.*
- *Read the keyboard data register from the base address + 4 (i.e., from 4($t0)).*

► **Question 5.** Write here the instructions that cancel the keyboard interrupt.

► Using the simulator, execute the system with the modified handler. Observe that the user program will now finish soon after a keypress, which is not intended, but at least you've got rid of the asterisk burst.

► **Question 6.** Why does the user program terminate just after a key press? *Clue: The handler now uses register $t0, which is also used as a loop counter by the user program in loops.asm.*

Because the handler modifies the register $t0, used in loops.asm, and doesn't restore its original value after handling the exception.

Step 5: Taking care of the user program context

The exception handler uses registers of the same register file as the interrupted user program (such as *$t0* in the previous step). Hence the registers used in the handler must be saved (to memory), used, and then retrieved before returning to the user program. This will preserve its execution *context* even in the presence of interrupts or other exceptions. The context of a user program includes the register file and also the value of the Program Counter (the return address). In the handler, we only need to save the registers that are used by the handler itself. Then, just before returning to the interrupted program, the saved registers will need to be *restored* to their saved values so that the user program does not see them changed by the handler.

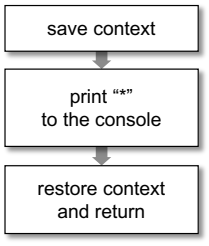

Figure 6. Handler structure, including context management

► Take note of which registers are used by the handler. At first sight, they are: $t0, $a0 and $v0, but remember, register **$at** must be included as well because it may be used by the assembler for expanding pseudo-instructions (such as la, li...). The context we need to preserve is comprised of these four registers. We are not modifying other registers in the handler, so we don't need to save/restore them. Even though the program *loops.asm* uses fewer registers, this is a general issue, and the handler must preserve all the registers it uses so that it is compatible with any user program

regardless of the registers it uses.

> *An important point here is that the use of registers $k0 and $k1 is forbidden to user programs, so they don't need to be saved as part of the user context. We will be using $k1 as a pointer to the context area in memory, as shown in Figure 7. The handler also uses register $k0 to preserve the return address to the user program, used at the end of the handler by instruction* jr $k0.

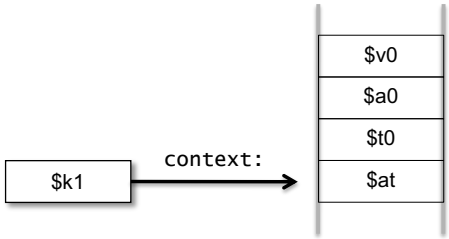

Figure 7. Register *$k1* points permanently to the memory area where the context is saved.

There is an additional detail regarding $at, which we need to save as part of the context. The register $at (*assembler temporary*) is reserved for the assembler and it is not directly usable from our programs. For example, if you write:

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

in your program, the assembler will raise an error message "Register 1 is reserved for assembler…". To go around this issue when you want to use $at directly, you need the *set* directive such as:

```
.set noat          # Instruct assembler not to complain about the use of $at
add $t1, $t0, $at  # Legal direct access to $at
.set at            # Back to reserved use of $at
```

► **Question 7.** Open *keyboard.handler* in a text editor.

• In the handler data segment (.kdata), declare a variable named context with 4-word capacity.

• In the system start-up code: load $k1 with the address of context

• At the beginning of the handler code, just after the comment "Save the user program context", write the instructions that save the four registers to context, (see Figure 7).

- After the comment "Restore context", write the code that restores the context registers.

► Use the simulator to check that the system works. The user program will take a few seconds to execute, and you should see just one asterisk printed on the console every time you press a key.

Summary

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

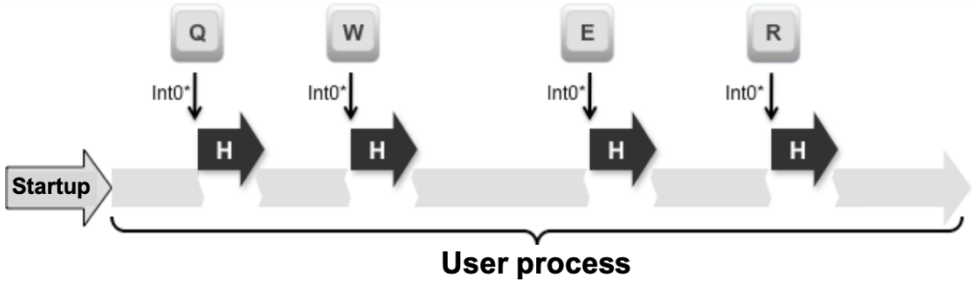

Figure 8. Handler *H* is executed once every time the keyboard is ready.

Task 5. Handling several interrupts

We will now extend the exception handler to also cover the handling of interrupts coming from the clock device simulated by PCSpim. Every time a new interrupt source is added, and generally, when a new interrupt cause must be handled, we need to:

- Define any **variables** that may be required for handling the interrupt in the *kdata* section.
- In the handler code, write the instructions needed to **identify** the new interrupt and to perform the corresponding **handling**.
- In the system start-up code, **initialise** the peripheral (enable it to interrupt).

Note that since we have been considering only one source of interrupts (*Int0*), the handler did not need to find what was the exact source of the interrupt. Since we now want to handle interrupts coming from either the keyboard or the clock, we will need the handler to discriminate between *Int0* and *Int2* to apply the proper handling.

Step 6: Enabling the clock interrupt

In this step, you will simply enable clock interrupts in the initialisation code without modifying the rest of the handler.

► **Question 8.** With a text editor, save *keyboard.handler* as *keyboard_and_clock.handler*. This will be your trap file from now on.

- In the start-up code, add the instructions that enable the clock to request interrupts.

- Change the coprocessor `$Status` register to unmask *Int2,* in addition to *Int0*.

► Run the system in the simulator. If the clock interrupt has been properly enabled, the console will be filled with asterisks (again!). You will fix this in later steps.

► **Question 9.** Explain why the system behaves this way. *Clue: Is the clock interrupt cancelled in the handler?*

Step 7: Determining the cause of an exception

Every interrupt (and generally, every exception cause) requires specific handling. In this step, you will build a small decision tree to associate each interrupt with its own handler. At this point, our handler always does the same handling (print an asterisk) regardless of the interrupt request received. The goal now is to implement the scheme shown in Figure 9 to discriminate the exception cause and give it adequate handling.

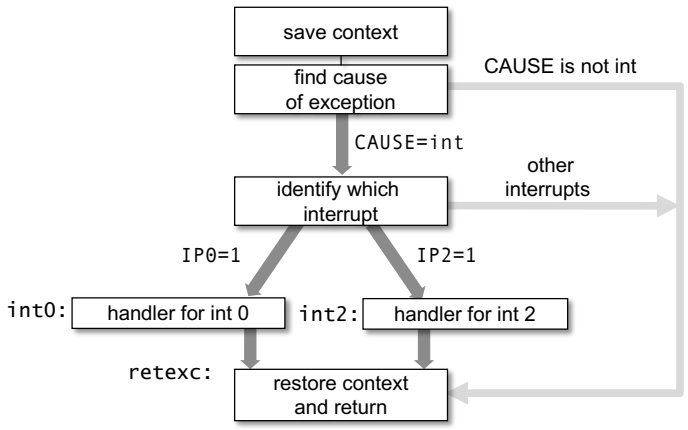

Figure 9. Decision on which handler to execute, int0 or int2

► **Question 10**. With the text editor, modify the *keyboard_and_clock.handler* file as follows:

- Add a new label (retexc, as in Figure 9) to mark the point in the handler that restores the context and gives the control back to the interrupted program. Jumping to retexc from any point in the exception handler will restore the context and return to the user program.
- Add a new label int0 and implement below it the handling that we've been applying to the keyboard interrupt since steps 4 and 5 (*print_char('*')*). At the end of the keyboard handler, don't forget to jump to retexc.

- Add a new label int2 followed by the clock interrupt handling. All we need to do now is cancel the clock interrupt in its adapter. See how to cancel the clock Ready bit in the peripheral description (Appendix 3). Do not forget to jump to retexc afterwards.

- Now that the two handlers are in place, add the instructions needed to read and analyse the exception cause from the $Cause register (see "Appendix 2. Coprocessor" for the description of this register). The place to do it is just after the comment "## Identify and handle exceptions". First, implement the part to detect that the cause of the exception is an interrupt. If it is, the analysis of which interrupt it was will continue in the next item. Otherwise, jump to retexc.

- Once it has been identified that the cause of the exception is an interrupt, you need to determine which one of the two possible interrupts was it, *Int0* or *Int2*, since both are enabled. You need to add instructions to analyse bits *IP0* and *IP2* from the $Status register and then jump to int0 or int2 accordingly. If none of these bits is set, then jump to retexc.

► Be aware of the registers used by the handler. In step 5, you implemented the saving and restoring of a context formed by registers *$at*, *$t0*, *$v0 and $a0*. You can restrict yourself to using only those four registers, and all will be fine. But if you want to use more registers in the handler (like *$t1*), make sure you include them in the context. Remember not only to save/restore any additional registers but also to resize the context area to make room for them.

► With the simulator, run the system to check that it works. If you successfully implemented this last step, you should see one (and only one) asterisk every time a key is pressed; the user program will smoothly start, execute, and terminate when the time comes. You will have the chance to implement more exciting clock interrupt handlers in future lab sessions.

ANNEX

Assembly for exceptions

New memory segments

.kdata — *Kernel data*. Defines the start of a kernel data segment. This segment contains the declaration of data located in the kernel memory area (addresses from 80000000_{16} to $FFFFFFFF_{16}$). This range of addresses is not accessible to user programs. Variables of the system exception handler need to be declared in a kernel data segment.

.ktext — *Kernel text.* Contains the instructions of the exception handler. These instructions can only be executed in kernel mode, so user programs cannot jump to this segment. This is the required directive to write the code of the system exception handler.

New instructions

mfc0 *rt, rs* — *Move from coprocessor 0.* Transfers the contents of coprocessor 0 register *rs* to the general-purpose register *rt*.

mtc0 *rt, rs* — *Move to coprocessor 0.* Transfers the contents of general-purpose register *rt* to the coprocessor 0 register *rs*.

rfe — *Restore from exception*. Copies the `$Status` register bits KU_P and IE_P to KU_C and IE_C, respectively. The effect is to restore the execution mode and the global interrupt-enable state of the program that was interrupted by the exception being handled.